Dive into Deep Learning

Aston Zhang
Zachary C. Lipton
Mu Li
Alexander J. Smola

Mar 11, 2019

This draft is a testing version (draft date: March 11, 2019).
Visit https://d2l.ai to obtain a later or release version.

Contents

Preface

Just a few years ago, there were no legions of deep learning scientists developing intelligent products and services at major companies and startups. When the youngest of us (the authors) entered the field, machine learning didn't command headlines in daily newspapers. Our parents had no idea what machine learning was, let alone why we might prefer it to a career in medicine or law. Machine learning was a forward-looking academic discipline with a narrow set of real-world applications. And those applications, e.g. speech recognition and computer vision, required so much domain knowledge that they were often regarded as separate areas entirely for which machine learning was one small component. Neural networks, the antecedents of the deep learning models that we focus on in this book, were regarded as outmoded tools.

In just the past five years, deep learning has taken the world by surprise, driving rapid progress in fields as diverse as computer vision, natural language processing, automatic speech recognition, reinforcement learning, and statistical modeling. With these advances in hand, we can now build cars that drive themselves (with increasing autonomy), smart reply systems that anticipate mundane replies, helping people dig out from mountains of email, and software agents that dominate the world's best humans at board games like Go, a feat once deemed to be decades away. Already, these tools are exerting a widening impact, changing the way movies are made, diseases are diagnosed, and playing a growing role in basic sciences – from astrophysics to biology. This book represents our attempt to make deep learning approachable, teaching you both the *concepts*, the *context*, and the *code*.

About This Book

One Medium Combining Code, Math, and HTML

For any computing technology to reach its full impact, it must be well-understood, well-documented, and supported by mature, well-maintained tools. The key ideas should be clearly distilled, minimizing the onboarding time needing to bring new practitioners up to date. Mature libraries should automate common tasks, and exemplar code should make it easy for practitioners to modify, apply, and extend common applications to suit their needs. Take dynamic web applications as an example. Despite a large number of companies, like Amazon, developing successful database-driven web applications in the 1990s, the full potential of this technology to aid creative entrepreneurs has only been realized over the past ten years, owing to the development of powerful, well-documented frameworks.

Realizing deep learning presents unique challenges because any single application brings together various disciplines. Applying deep learning requires simultaneously understanding (i) the motivations for casting a problem in a particular way, (ii) the mathematics of a given modeling approach, (iii) the optimization algorithms for fitting the models to data, (iv) and the engineering required to train models efficiently, navigating the pitfalls of numerical computing and getting the most out of available hardware. Teaching both the critical thinking skills required to formulate problems, the mathematics to solve them, and the software tools to implement those solutions all in one place presents formidable challenges. Our goal in this book is to present a unified resource to bring would-be practitioners up to speed.

We started this book project in July 2017 when we needed to explain MXNet's (then new) Gluon interface to our users. At the time, there were no resources that were simultaneously (1) up to date, (2) covered the full breadth of modern machine learning with anything resembling of technical depth, and (3) interleaved the exposition one expects from an engaging textbook with the clean runnable code one seeks in hands-on tutorials. We found plenty of code examples for how to use a given deep learning framework (e.g. how to do basic numerical computing with matrices in TensorFlow) or for implementing particular techniques (e.g. code snippets for LeNet, AlexNet, ResNets, etc) in the form of blog posts or on GitHub. However, these examples typically focused on *how* to implement a given approach, but left out the discussion of *why* certain algorithmic decisions are made. While sporadic topics have been covered in blog posts, e.g. on the website Distill or personal blogs, they only covered selected topics in deep learning, and often lacked associated code. On the other hand, while several textbooks have emerged, most notably Goodfellow, Bengio and Courville, 2016, which offers an excellent survey of the concepts behind deep learning, these resources don't marry the descriptions to realizations of the concepts in code, sometimes leaving readers clueless as to how to implement them. Moreover, too many resources are hidden behind the paywalls of commercial course providers.

We set out to create a resource that could (1) be freely available for everyone, (2) offer sufficient technical depth to provide a starting point on the path to actually becoming an applied machine learning scientist, (3) include runnable code, showing readers *how* to solve problems in practice, and (4) that allowed for rapid updates, both by us, and also by the community at large, and (5) be complemented by a forum for interactive discussion of technical details and to answer questions.

These goals were often in conflict. Equations, theorems, and citations are best managed and laid out in LaTeX. Code is best described in Python. And webpages are native in HTML and JavaScript. Further-

more, we want the content to be accessible both as executable code, as a physical book, as a downloadable PDF, and on the internet as a website. At present there exist no tools and no workflow perfectly suited to these demands, so we had to assemble our own. We describe our approach in detail in the *appendix*. We settled on Github to share the source and to allow for edits, Jupyter notebooks for mixing code, equations and text, Sphinx as a rendering engine to generate multiple outputs, and Discourse for the forum. While our system is not yet perfect, these choices provide a good compromise among the competing concerns. We believe that this might be the first book published using such an integrated workflow.

Learning by Doing

Many textbooks teach a series of topics, each in exhaustive detail. For example, Chris Bishop's excellent textbook, Pattern Recognition and Machine Learning, teaches each topic so thoroughly, that getting to the chapter on linear regression requires a non-trivial amount of work. While experts love this book precisely for its thoroughness, for beginners, this property limits its usefulness as an introductory text.

In this book, we'll teach most concepts *just in time*. In other words, you'll learn concepts at the very moment that they are needed to accomplish some practical end. While we take some time at the outset to teach fundamental preliminaries, like linear algebra and probability. We want you to taste the satisfaction of training your first model before worrying about more esoteric probability distributions.

Aside from a few preliminary notebooks that provide a crash course in the basic mathematical background, each subsequent notebook introduces both a reasonable number of new concepts and provides a single self-contained working example – using a real dataset. This presents an organizational challenge. Some models might logically be grouped together in a single notebook. And some ideas might be best taught by executing several models in succession. On the other hand, there's a big advantage to adhering to a policy of *1 working example, 1 notebook*: This makes it as easy as possible for you to start your own research projects by leveraging our code. Just copy a notebook and start modifying it.

We will interleave the runnable code with background material as needed. In general, we will often err on the side of making tools available before explaining them fully (and we will follow up by explaining the background later). For instance, we might use *stochastic gradient descent* before fully explaining why it is useful or why it works. This helps to give practitioners the necessary ammunition to solve problems quickly, at the expense of requiring the reader to trust us with some curatorial decisions.

Throughout, we'll be working with the MXNet library, which has the rare property of being flexible enough for research while being fast enough for production. This book will teach deep learning concepts from scratch. Sometimes, we want to delve into fine details about the models that would typically be hidden from the user by `Gluon`'s advanced abstractions. This comes up especially in the basic tutorials, where we want you to understand everything that happens in a given layer or optimizer. In these cases, we'll often present two versions of the example: one where we implement everything from scratch, relying only on NDArray and automatic differentiation, and another, more practical example, where we write succinct code using `Gluon`. Once we've taught you how some component works, we can just use the `Gluon` version in subsequent tutorials.

Content and Structure

The book can be roughly divided into three sections:

- The first part covers prerequisites and basics. The first chapter offers an Introduction to Deep Learning. In *Crashcourse*, we'll quickly bring you up to speed on the prerequisites required for hands-on deep learning, such as how to acquire and run the codes covered in the book. Deep Learning Basics covers the most basic concepts and techniques of deep learning, such as multi-layer perceptrons and regularization.

- The next three chapters focus on modern deep learning techniques. *Deep Learning Computation* describes the various key components of deep learning calculations and lays the groundwork for the later implementation of more complex models. Next we explain *Convolutional Neural Networks*, powerful tools that form the backbone of most modern computer vision systems in recent years. Subsequently, we introduce *Recurrent Neural Networks*, models that exploit temporal or sequential structure in data, and are commonly used for natural language processing and time series prediction. These sections will get you up to speed on the basic tools behind most modern deep learning.

- Part three discusses scalability, efficiency and applications. First we discuss several common *Optimization Algorithms* used to train deep learning models. The next chapter, *Performance*, examines several important factors that affect the computational performance of your deep learning code. Chapters 9 and 10 illustrate major applications of deep learning in computer vision and natural language processing, respectively.

An outline of the book together with possible flows for navigating it is given below. The arrows provide a graph of prerequisites:
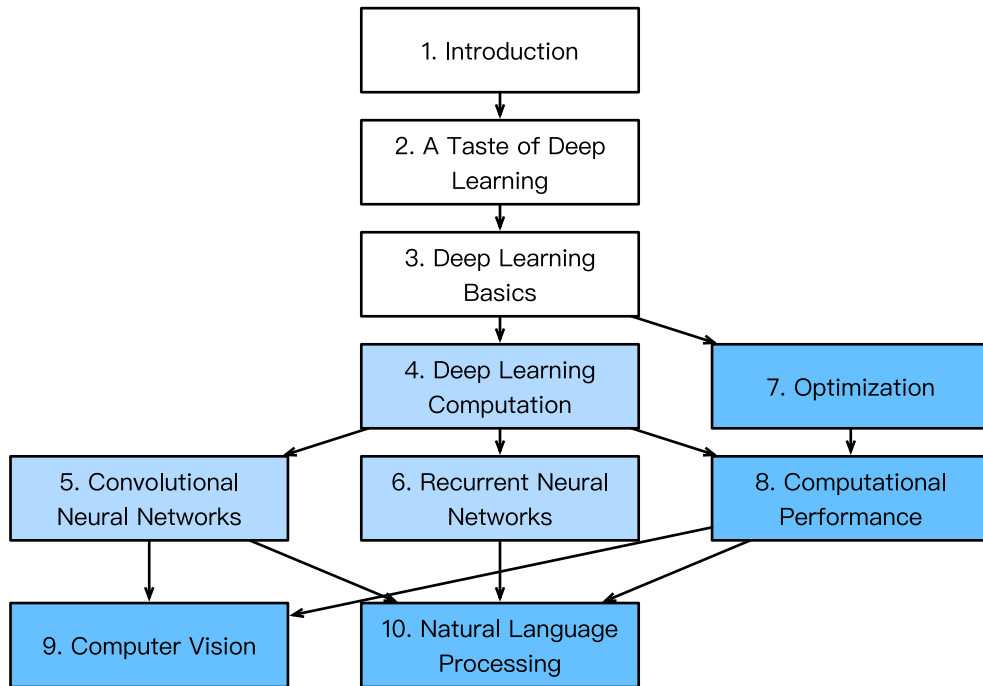
Fig. 1: Book structure

Code

Most sections of this book feature executable code. We recognize the importance of an interactive learning experience in deep learning. At present certain intuitions can only be developed through trial and error, tweaking the code in small ways and observing the results. Ideally, an elegant mathematical theory might tell us precisely how to tweak our code to achieve a desired result. Unfortunately, at present such elegant theories elude us. Despite our best attempts, our explanations for of various techniques might be lacking, sometimes on account of our shortcomings, and equally often on account of the nascent state of the science of deep learning. We are hopeful that as the theory of deep learning progresses, future editions of this book will be able to provide insights in places the present edition cannot.

Most of the code in this book is based on Apache MXNet. MXNet is an open-source framework for deep learning and the preferred choice of AWS (Amazon Cloud Services), as well as many colleges and companies. All of the code in this book has passed tests under MXNet 1.2.0. However, due to the rapid development of deep learning, some code *in the print edition* may not work properly in future versions of MXNet. However, we plan to keep the online version remain up-to-date. In case of such problems, please consult the section Installation and Running to update the code and runtime environment. At times, to avoid unnecessary repetition, we encapsulate the frequently-imported and referred-to functions, classes, etc. in this book in the `gluonbook` package, version number 1.0.0. We give a detailed overview of these functions and classes in the appendix gluonbook package index

---

Target Audience

This book is for students (undergraduate or graduate), engineers, and researchers, who seek a solid grasp of the practical techniques of deep learning. Because we explain every concept from scratch, no previous background in deep learning or machine learning is required. Fully explaining the methods of deep learning requires some mathematics and programming, but we'll only assume that you come in with some basics, including (the very basics of) linear algebra, calculus, probability, and Python programming. Moreover, this book's appendix provides a refresher on most of the mathematics covered in this book. Most of the time, we will prioritize intuition and ideas over mathematical rigor. There are many terrific books which can lead the interested reader further. For instance Linear Analysis by Bela Bollobas covers linear algebra and functional analysis in great depth. All of Statistics is a terrific guide to statistics. And if you have not used Python before, you may want to peruse the Python tutorial.

Forum

Associated with this book, we've launched a discussion forum, located at discuss.mxnet.io. When you have questions on any section of the book, you can find the associated discussion page by scanning the QR code at the end of the section to participate in its discussions. The authors of this book and broader MXNet developer community frequently participate in forum discussions.

Acknowledgments

Summary

- Deep learning has revolutionized pattern recognition, introducing technology that now powers a wide range of technologies, including computer vision, natural language processing, automatic speech recognition.

- To successfully apply deep learning, you must understand how to cast a problem, the mathematics of modeling, the algorithms for fitting your models to data, and the engineering techniques to implement it all.

- This book presents a comprehensive resource, including prose, figures, mathematics, and code, all in one place.

- To answer questions related to this book, visit our forum at https://discuss.mxnet.io/.

- Apache MXNet is a powerful library for coding up deep learning models and running them in parallel across GPU cores.

- Gluon is a high level library that makes it easy to code up deep learning models using Apache MXNet.

- Conda is a Python package manager that ensures that all software dependencies are met.

- All notebooks are available for download on GitHub and the conda configurations needed to run this book's code are expressed in the `environment.yml` file.

- If you plan to run this code on GPUs, don't forget to install the necessary drivers and update your configuration.

Exercises

1. Register an account on the discussion forum of this book discuss.mxnet.io.

2. Install Python on your computer.

3. Follow the links at the bottom of the section to the forum, where you'll be able to seek out help and discuss the book and find answers to your questions by engaging the authors and broader community.

4. Create an account on the forum and introduce yourself.

Scan the QR Code to Discuss

To get you up and running with hands-on experiences, we'll need you to set up with a Python environment, Jupyter's interactive notebooks, the relevant libraries, and the code needed to *run the book*.

Obtaining Code and Installing Running Environment

This book with code can be downloaded for free. For simplicity we recommend conda, a popular Python package manager to install all libraries. Windows users and Linux/macOS users can follow the instructions below respectively.

Windows Users

If it is your first time to run the code of this book, you need to complete the following 5 steps. Next time you can jump directly to Step 4 and Step 5.

Step 1 is to download and install Miniconda according to the operating system in use. During the installation, it is required to choose the option Add Anaconda to the system PATH environment variable.

Step 2 is to download the compressed file containing the code of this book. It is available at https://www.d2l.ai/d2l-en-1.0.zip. After downloading the zip file, create a folder `d2l-en` and extract the zip file into the folder. At the current folder, enter `cmd` on the address bar of File Explorer to enter the command line mode.

Step 3 is to create a virtual (running) environment using conda to install the libraries needed by this book. Here `environment.yml` is placed in the downloaded zip file. Open the file with a text editor to see

the libraries (such as MXNet and `d2lzh` package) and their version numbers on which running the code of the book is dependent.

```
conda env create -f environment.yml
```

Step 4 is to activate the environment that is created earlier. Activating this environment is a prerequisite for running the code of this book. To exit the environment, use the command `conda deactivate` (if the conda version is lower than 4.4, use the command `deactivate`).

```
# If the conda version is lower than 4.4, use the command `activate gluon`
conda activate gluon
```

Step 5 is to open the Jupyter Notebook.

```
jupyter notebook
```

At this point open http://localhost:8888 (usually automatically open) in the browser, then you can view and run the code in each section of the book.

Linux/macOS Users

Step 1 is to download and install Miniconda according to the operating system in use. It is a sh file. Open the Terminal application and enter the command to execute the sh file, such as

```
# The file name is subject to change, always use the one downloaded from the
# Miniconda website
sh Miniconda3-latest-Linux-x86_64.sh
```

The terms of use will be displayed during installation. Press to continue reading, press Q to exit reading. After that, answer the following questions:

```
Do you accept the license terms? [yes|no]
[no] >>> yes
Do you wish the installer to prepend the Miniconda3 install location
to PATH in your /home/your_name/your_file ? [yes|no]
[no] >>> yes
```

After the installation is complete, conda should be made to take effect. Linux users need to run `source ~/.bashrc` or restart the command line application; macOS users need to run `source ~/.bash_profile` or restart the command line application.

Step 2 is to download the compressed file containing the code of this book, and extract it into the folder. Run the following commands. For Linux users who do not install `unzip`, they can run the command `sudo apt install unzip` to install it.

```
mkdir d2l-en && cd d2l-en
curl https://www.d2l.ai/d2l-en-1.0.zip -o d2l-en.zip
unzip d2l-en.zip && rm d2l-en.zip
```

For Step 3 to Step 5, refer to the such steps for Windows users as described earlier. If the conda version is lower than 4.4, replace the command in Step 4 with `source activate gluon` and exit the virtual environment using the command `source deactivate`.

Updating Code and Running Environment

Since deep learning and MXNet grow fast, this open source book will be updated and released regularly. To update the open source content of this book (e.g., code) with corresponding running environment (e.g., MXNet of a later version), follow the steps below.

Step 1 is to re-download the latest compressed file containing the code of this book. It is available at https://www.d2l.ai/d2l-en.zip. After extracting the zip file, enter the folder `d2l-en`.

Step 2 is to update the running environment with the command

```
conda env update -f environment.yml
```

The subsequent steps for activating the environment and running Jupyter are the same as those described earlier.

GPU Support

By default MXNet is installed without GPU support to ensure that it will run on any computer (including most laptops). Part of this book requires or recommends running with GPU. If your computer has NVIDIA graphics cards and has installed CUDA, you should modify the conda environment to download the CUDA enabled build.

Step 1 is to uninstall MXNet without GPU support. If you have installed the virtual environment for running the book, you need to activate this environment then uninstall MXNet without GPU support:

```
pip uninstall mxnet
```

Then exit the virtual environment.

Step 2 is to update the environment description in `environment.yml`. Likely, you'll want to replace `mxnet` by `mxnet-cu90`. The number following the hyphen (90 above) corresponds to the version of CUDA you installed). For instance, if you're on CUDA 8.0, you need to replace `mxnet-cu90` with `mxnet-cu80`. You should do this *before* creating the conda environment. Otherwise you will need to rebuild it later.

Step 3 is to update the virtual environment. Run the command

```
conda env update -f environment.yml
```

Then we only need to activate the virtual environment to use MXNet with GPU support to run the book. Note that you need to repeat these 3 steps to use MXNet with GPU support if you download the updated code later.

Exercises

1. Download the code for the book and install the runtime environment.

Scan the QR Code to Discuss

1

Introduction

Until recently, nearly all of the computer programs that we interacted with every day were coded by software developers from first principles. Say that we wanted to write an application to manage an e-commerce platform. After huddling around a whiteboard for a few hours to ponder the problem, we would come up with the broad strokes of a working solution that would probably look something like this: (i) users would interact with the application through an interface running in a web browser or mobile application (ii) our application would rely on a commerical database engine to keep track of each user's state and maintain records of all historical transactions (ii) at the heart of our application, running in parallel across many servers, the *business logic* (you might say, the *brains*) would map out in methodical details the appropriate action to take in every conceivable circumstance.

To build the *brains* of our application, we'd have to step through every possible corner case that we anticipate encountering, devising appropriate rules. Each time a customer clicks to add an item to their shopping cart, we add an entry to the shopping cart database table, associating that user's ID with the requested product's ID. While few developers ever get it completely right the first time (it might take some test runs to work out the kinks), for the most part, we could write such a program from first principles and confidently launch it *before ever seeing a real customer*. Our ability to design automated systems from first principles that drive functioning products and systems, often in novel situations, is a remarkable cognitive feat. And when you're able to devise solutions that work 100% of the time. *you should not be using machine learning*.

Fortunatelyfor the growing community of ML scientistsmany problems in automation don't bend so easily to human ingenuity. Imagine huddling around the whiteboard with the smartest minds you know, but this time you are tackling any of the following problems: * Write a program that predicts tomorrow's weather given geographic information, satellite images, and a trailing window of past weather. * Write a program

that takes in a question, expressed in free-form text, and answers it correctly. * Write a program that given an image can identify all the people it contains, drawing outlines around each. * Write a program that presents users with products that they are likely to enjoy but unlikely, in the natural course of browsing, to encounter.

In each of these cases, even elite programmers are incapable of coding up solutions from scratch. The reasons for this can vary. Sometimes the program that we are looking for follows a pattern that changes over time, and we need our programs to adapt. In other cases, the relationship (say between pixels, and abstract categories) may be too complicated, requiring thousands or millions of computations that are beyond our conscious understanding (even if our eyes manage the task effortlessly). Machine learning (ML) is the study of powerful techniques that can *learn behavior* from *experience*. As ML algorithm accumulates more experience, typically in the form of observational data or interactions with an environment, their performance improves. Contrast this with our deterministic e-commerce platform, which performs according to the same business logic, no matter how much experience accrues, until the developers themselves *learn* and decide that it's time to update the software. In this book, we will teach you the fundamentals of machine learning, and focus in particular on deep learning, a powerful set of techniques driving innovations in areas as diverse as computer vision, natural language processing, healthcare, and genomics.

1.1 A Motivating Example

Before we could begin writing, the authors of this book, like much of the work force, had to become caffeinated. We hopped in the car and started driving. Using an iPhone, Alex called out Hey Siri', awakening the phone's voice recognition system. Then Mu commanded directions to Blue Bottle coffee shop'. The phone quickly displayed the transcription of his command. It also recognized that we were asking for directions and launched the Maps application to fulfill our request. Once launched, the Maps app identified a number of routes. Next to each route, the phone displayed a predicted transit time. While we fabricated this story for pedagogical convenience, it demonstrates that in the span of just a few seconds, our everyday interactions with a smartphone can engage several machine learning models.

Imagine just writing a program to respond to a *wake word* like Alexa', Okay, Google' or Siri'. Try coding it up in a room by yourself with nothing but a computer and a code editor. How would you write such a program from first principles? Think about it the problem is hard. Every second, the microphone will collect roughly 44,000 samples. What rule could map reliably from a snippet of raw audio to confident predictions `{yes, no}` on whether the snippet contains the wake word? If you're stuck, don't worry. We don't know how to write such a program from scratch either. That's why we use ML.
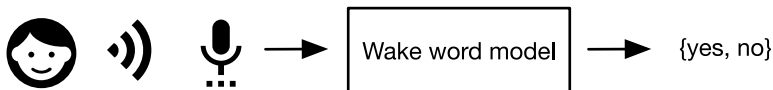
Here's the trick. Often, even when we don't know how to tell a computer explicitly how to map from inputs to outputs, we are nonetheless capable of performing the cognitive feat ourselves. In other words, even if you don't know *how to program a computer* to recognize the word Alexa', you yourself *are*

able to recognize the word Alexa'. Armed with this ability, we can collect a huge *dataset* containing examples of audio and label those that *do* and that *do not* contain the wake word. In the ML approach, we do not design a system *explicitly* to recognize wake words. Instead, we define a flexible program whose behavior is determined by a number of *parameters*. Then we use the dataset to determine the best possible set of parameters, those that improve the performance of our program with respect to some measure of performance on the task of interest.

You can think of the parameters as knobs that we can turn, manipulating the behavior of the program. Fixing the parameters, we call the program a *model*. The set of all distinct programs (input-output mappings) that we can produce just by manipulating the parameters is called a *family* of models. And the *meta-program* that uses our dataset to choose the parameters is called a *learning algorithm*.

Before we can go ahead and engage the learning algorithm, we have to define the problem precisely, pinning down the exact nature of the inputs and outputs, and choosing an appropriate model family. In this case, our model receives a snippet of audio as *input*, and it generates a selection among {yes, no} as *output*which, if all goes according to plan, will closely approximate whether (or not) the snippet contains the wake word.

If we choose the right family of models, then there should exist one setting of the knobs such that the model fires yes every time it hears the word Alexa'. Because the exact choice of the wake word is arbitrary, we'll probably need a model family capable, via another setting of the knobs, of firing yes on the word Apricot'. We expect that the same model should apply to Alexa' recognition and Apricot' recognition because these are similar tasks. However, we might need a different family of models entirely if we want to deal with fundamentally different inputs or outputs, say if we wanted to map from images to captions, or from English sentences to Chinese sentences.

As you might guess, if we just set all of the knobs randomly, it's not likely that our model will recognize Alexa', Apricot', or any other English word. In deep learning, the *learning* is the process by which we discover the right setting of the knobs coercing the desired behaviour from our model.

The training process usually looks like this:

1. Start off with a randomly initialized model that can't do anything useful.

2. Grab some of your labeled data (e.g. audio snippets and corresponding {yes,no} labels)

3. Tweak the knobs so the model sucks less with respect to those examples

4. Repeat until the model is awesome.
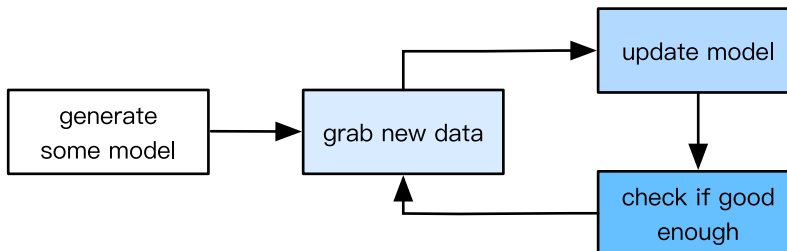
To summarize, rather than code up a wake word recognizer, we code up a program that can *learn* to recognize wake words, *if we present it with a large labeled dataset*. You can think of this act of determining a program's behavior by presenting it with a dataset as *programming with data*. We can program' a cat detector by providing our machine learning system with many examples of cats and dogs, such as the images below:

| cat | cat | dog | dog |

This way the detector will eventually learn to emit a very large positive number if it's a cat, a very large negative number if it's a dog, and something closer to zero if it isn't sure, and this barely scratches the surface of what ML can do.

Deep learning is just one among many popular frameworks for solving machine learning problems. While thus far, we've only talked about machine learning broadly and not deep learning, there's a couple points worth sneaking in here: First, the problems that we've discussed thus far: learning from raw audio signal, directly from the pixels in images, and mapping between sentences of arbitrary lengths and across languages are problems where deep learning excels and traditional ML tools faltered. Deep models are *deep* in precisely the sense that they learn many *layers* of computation. It turns out that these many-layered (or hierarchical) models are capable of addressing low-level perceptual data in a way that previous tools could not. In bygone days, the crucial part of applying ML to these problems consisted of coming up with manually engineered ways of transforming the data into some form amenable to *shallow* models. One key advantage of deep learning is that it replaces not only the *shallow* models at the end of traditional learning pipelines, but also the labor-intensive feature engineering. Secondly, by replacing much of the *domain-specific preprocessing*, deep learning has eliminated many of the boundaries that previously separated computer vision, speech recognition, natural language processing, medical informatics, and other application areas, offering a unified set of tools for tackling diverse problems.

1.2 The Key Components: Data, Models, and Algorithms

In our *wake-word* example, we described a dataset consisting of audio snippets and binary labels gave a hand-wavy sense of how we might *train* a model to approximate a mapping from snippets to classifications. This sort of problem, where we try to predict a designated unknown *label* given known *inputs* (also

called *features* or *covariates*), and examples of both is called *supervised learning*, and it's just one among many *kinds* of machine learning problems. In the next section, we'll take a deep dive into the different ML problems. First, we'd like to shed more light on some core components that will follow us around, no matter what kind of ML problem we take on:

1. The **data** that we can learn from

2. A **model** of how to transform the data

3. A **loss** function that quantifies the *badness* of our model

4. An **algorithm** to adjust the model's parameters to minimize the loss

1.2.1 Data

It might go without saying that you cannot do data science without data. We could lose hundreds of pages pondering the precise nature of data but for now we'll err on the practical side and focus on the key properties to be concerned with. Generally we are concerned with a collection of *examples* (also called *data points*, *samples*, or *instances*). In order to work with data usefully, we typically need to come up with a suitable numerical representation. Each *example* typically consists of a collection of numerical attributes called *features* or *covariates*.

If we were working with image data, each individual photograph might constitute an *example*, each represented by an ordered list of numerical values corresponding to the brightness of each pixel. A 200×200 color photograph would consist of $200 \times 200 \times 3 = 120000$ numerical values, corresponding to the brightness of the red, green, and blue channels corresponding to each spatial location. In a more traditional task, we might try to predict whether or not a patient will survive, given a standard set of features such as age, vital signs, diagnoses, etc.

When every example is characterized by the same number of numerical values, we say that the data consists of *fixed-length* vectors and we describe the (constant) length of the vectors as the *dimensionality* of the data. As you might imagine, fixed length can be a convenient property. If we wanted to train a model to recognize cancer in microscopy images, fixed-length inputs means we have one less thing to worry about.

However, not all data can easily be represented as fixed length vectors. While we might expect micro-scrope images to come from standard equipment, we can't expect images mined from the internet to all show up in the same size. While we might imagine cropping images to a standard size, text data resists fixed-length representations even more stubbornly. Consider the product reviews left on e-commerce sites like Amazon or TripAdvisor. Some are short: it stinks!. Others ramble for pages. One major advantage of deep learning over traditional methods is the comparative grace with which modern models can handle *varying-length* data.

Generally, the more data we have, the easier our job becomes. When we have more data, we can train more powerful models, and rely less heavily on pre-conceived assumptions. The regime change from (comparatively small) to big data is a major contributor to the success of modern deep learning. To drive the point home, many of the most exciting models in deep learning either don't work without large data sets. Some others work in the low-data regime, but no better than traditional approaches.

Finally it's not enough to have lots of data and to process it cleverly. We need the *right* data. If the data is full of mistakes, or if the chosen features are not predictive of the target quantity of interest, learning is going to fail. The situation is well captured by the cliché: *garbage in, garbage out*. Moreover, poor predictive performance isn't the only potential consequence. In sensitive applications of machine learning, like predictive policing, resumé screening, and risk models used for lending, we must be especially alert to the consequences of garbage data. One common failure mode occurs in datasets where some groups of people are unrepresented in the training data. Imagine applying a skin cancer recognition system in the wild that had never seen black skin before. Failure can also occur when the data doesn't merely under-represent some groups, but reflects societal prejudices. For example if past hiring decisions are used to train a predictive model that will be used to screen resumes, then machine learning models could inadvertantly capture and automate historical injustices. Note that this can all happen without the data scientist being complicit, or even aware.

1.2.2 Models

Most machine learning involves *transforming* the data in some sense. We might want to build a system that ingests photos and predicts *smiley-ness*. Alternatively, we might want to ingest a set of sensor readings and predict how *normal* vs *anomalous* the readings are. By *model*, we denote the computational machinery for ingesting data of one type, and spitting out predictions of a possibly different type. In particular, we are interested in statistical models that can be estimated from data. While simple models are perfectly capable of addressing appropriately simple problems the problems that we focus on in this book stretch the limits of classical methods. Deep learning is differentiated from classical approaches principally by the set of powerful models that it focuses on. These models consist of many successive transformations of the data that are chained together top to bottom, thus the name *deep learning*. On our way to discussing deep neural networks, we'll discuss some more traditional methods.

1.2.3 Objective functions

Earlier, we introduced machine learning as learning behavior from experience. By *learning* here, we mean *improving* at some task over time. But who is to say what constitutes an improvement? You might imagine that we could propose to update our model, and some people might disagree on whether the proposed update constitued an improvement or a decline.

In order to develop a formal mathematical system of learning machines, we need to have formal measures of how good (or bad) our models are. In machine learning, and optimization more generally, we call these objective functions. By convention, we usually define objective funcitons so that *lower* is *better*. This is merely a convention. You can take any function f for which higher is better, and turn it into a new function f' that is qualitatively identical but for which lower is better by setting $f' = -f$. Because lower is better, these functions are sometimes called *loss functions* or *cost functions*.

When trying to predict numerical values, the most common objective function is squared error $(y - \hat{y})^2$. For classification, the most common objective is to minimize error rate, i.e., the fraction of instances on which our predictions disagree with the ground truth. Some objectives (like squared error) are easy to

optimize. Others (like error rate) are difficult to optimize directly, owing to non-differentiability or other complications. In these cases, it's common to optimize a surrogate objective.

Typically, the loss function is defined with respect to the models parameters and depends upon the dataset. The best values of our model's parameters are learned by minimizing the loss incurred on a *training set* consisting of some number of *examples* collected for training. However, doing well on the training data doesn't guarantee that we will do well on (unseen) test data. So we'll typically want to split the available data into two partitions: the training data (for fitting model parameters) and the test data (which is held out for evaluation), reporting the following two quantities:

- **Training Error:** The error on that data on which the model was trained. You could think of this as being like a student's scores on practice exams used to prepare for some real exam. Even if the results are encouraging, that does not guarantee success on the final exam.

- **Test Error:** This is the error incurred on an unseen test set. This can deviate significantly from the training error. When a model fails to generalize to unseen data, we say that it is *overfitting*. In real-life terms, this is like flunking the real exam despite doing well on practice exams.

1.2.4 Optimization algorithms

Once we've got some data source and representation, a model, and a well-defined objective function, we need an algorithm capable of searching for the best possible parameters for minimizing the loss function. The most popular optimization algorithms for neural networks follow an approach called gradient descent. In short, at each step, they check to see, for each parameter, which way the training set loss would move if you perturbed that parameter just a small amount. They then update the parameter in the direction that reduces the loss.

1.3 Kinds of Machine Learning

In the following sections, we will discuss a few types of machine learning in some more detail. We begin with a list of *objectives*, i.e. a list of things that machine learning can do. Note that the objectives are complemented with a set of techniques of *how* to accomplish them, i.e. training, types of data, etc. The list below is really only sufficient to whet the readers' appetite and to give us a common language when we talk about problems. We will introduce a larger number of such problems as we go along.

1.3.1 Supervised learning

Supervised learning addresses the task of predicting *targets* given input data. The targets, also commonly called *labels*, are generally denoted y. The input data points, also commonly called *examples* or *instances*, are typically denoted x. The goal is to produce a model f_θ that maps an input x to a prediction $f_\theta(x)$

To ground this description in a concrete example, if we were working in healthcare, then we might want to predict whether or not a patient would have a heart attack. This observation, *heart attack* or *no heart*

attack, would be our label y. The input data x might be vital signs such as heart rate, diastolic and systolic blood pressure, etc.

The supervision comes into play because for choosing the parameters θ, we (the supervisors) provide the model with a collection of *labeled examples* (x_i, y_i), where each example x_i is matched up against its correct label.

In probabilistic terms, we typically are interested in estimating the conditional probability $P(y|x)$. While it's just one among several approaches to machine learning, supervised learning accounts for the majority of machine learning in practice. Partly, that's because many important tasks can be described crisply as estimating the probability of some unknown given some available evidence:

- Predict cancer vs not cancer, given a CT image.

- Predict the correct translation in French, given a sentence in English.

- Predict the price of a stock next month based on this month's financial reporting data.

Even with the simple description predict targets from inputs' supervised learning can take a great many forms and require a great many modeling decisions, depending on the type, size, and the number of inputs and outputs. For example, we use different models to process sequences (like strings of text or time series data) and for processing fixed-length vector representations. We'll visit many of these problems in depth throughout the first 9 parts of this book.

Put plainly, the learning process looks something like this. Grab a big pile of example inputs, selecting them randomly. Acquire the ground truth labels for each. Together, these inputs and corresponding labels (the desired outputs) comprise the training set. We feed the training dataset into a supervised learning algorithm. So here the *supervised learning algorithm* is a function that takes as input a dataset, and outputs another function, *the learned model*. Then, given a learned model, we can take a new previously unseen input, and predict the corresponding label.
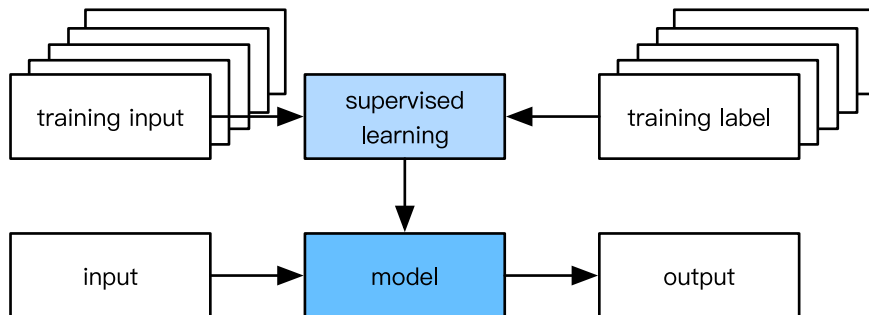
Regression

Perhaps the simplest supervised learning task to wrap your head around is Regression. Consider, for example a set of data harvested from a database of home sales. We might construct a table, where each row corresponds to a different house, and each column corresponds to some relevant attribute, such as the square footage of a house, the number of bedrooms, the number of bathrooms, and the number of minutes

(walking) to the center of town. Formally, we call one row in this dataset a *feature vector*, and the object (e.g. a house) it's associated with an *example*.

If you live in New York or San Francisco, and you are not the CEO of Amazon, Google, Microsoft, or Facebook, the (sq. footage, no. of bedrooms, no. of bathrooms, walking distance) feature vector for your home might look something like: $[100, 0, .5, 60]$. However, if you live in Pittsburgh, it might look more like $[3000, 4, 3, 10]$. Feature vectors like this are essential for all the classic machine learning problems. We'll typically denote the feature vector for any one example $\mathbf{x_i}$ and the set of feature vectors for all our examples X.

What makes a problem a *regression* is actually the outputs. Say that you're in the market for a new home, you might want to estimate the fair market value of a house, given some features like these. The target value, the price of sale, is a *real number*. We denote any individual target y_i (corresponding to example $\mathbf{x_i}$) and the set of all targets $\mathbf{y}$ (corresponding to all examples X). When our targets take on arbitrary real values in some range, we call this a regression problem. The goal of our model is to produce predictions (guesses of the price, in our example) that closely approximate the actual target values. We denote these predictions $\hat{y}_i$ and if the notation seems unfamiliar, then just ignore it for now. We'll unpack it more thoroughly in the subsequent chapters.

Lots of practical problems are well-described regression problems. Predicting the rating that a user will assign to a movie is a regression problem, and if you designed a great algorithm to accomplish this feat in 2009, you might have won the $1 million Netflix prize. Predicting the length of stay for patients in the hospital is also a regression problem. A good rule of thumb is that any *How much?* or *How many?* problem should suggest regression.

- How many hours will this surgery take?' - *regression*

- How many dogs are in this photo?' - *regression*.

However, if you can easily pose your problem as Is this a _ ?', then it's likely, classification, a different fundamental problem type that we'll cover next. Even if you've never worked with machine learning before, you've probably worked through a regression problem informally. Imagine, for example, that you had your drains repaired and that your contractor spent $x_1 = 3$ hours removing gunk from your sewage pipes. Then she sent you a bill of $y_1 = \$350$. Now imagine that your friend hired the same contractor for $x_2 = 2$ hours and that she received a bill of $y_2 = \$250$. If someone then asked you how much to expect on their upcoming gunk-removal invoice you might make some reasonable assumptions, such as more hours worked costs more dollars. You might also assume that there's some base charge and that the contractor then charges per hour. If these assumptions held true, then given these two data points, you could already identify the contractor's pricing structure: \$100 per hour plus \$50 to show up at your house. If you followed that much then you already understand the high-level idea behind linear regression (and you just implicitly designed a linear model with bias).

In this case, we could produce the parameters that exactly matched the contractor's prices. Sometimes that's not possible, e.g., if some of the variance owes to some factors besides your two features. In these cases, we'll try to learn models that minimize the distance between our predictions and the observed

values. In most of our chapters, we'll focus on one of two very common losses, the L1 loss where

$$l(y, y') = \sum_i |y_i - y'_i|$$

and the least mean squares loss, aka L2 loss where

$$l(y, y') = \sum_i (y_i - y'_i)^2.$$

As we will see later, the L_2 loss corresponds to the assumption that our data was corrupted by Gaussian noise, whereas the L_1 loss corresponds to an assumption of noise from a Laplace distribution.

Classification

While regression models are great for addressing *how many?* questions, lots of problems don't bend comfortably to this template. For example, a bank wants to add check scanning to their mobile app. This would involve the customer snapping a photo of a check with their smartphone's camera and the machine learning model would need to be able to automatically understand text seen in the image. It would also need to understand hand-written text to be even more robust. This kind of system is referred to as optical character recognition (OCR), and the kind of problem it solves is called a classification. It's treated with a distinct set of algorithms than those that are used for regression.

In classification, we want to look at a feature vector, like the pixel values in an image, and then predict which category (formally called *classes*), among some set of options, an example belongs. For hand-written digits, we might have 10 classes, corresponding to the digits 0 through 9. The simplest form of classification is when there are only two classes, a problem which we call binary classification. For example, our dataset X could consist of images of animals and our *labels Y* might be the classes $\{\mathrm{cat}, \mathrm{dog}\}$. While in regression, we sought a *regressor* to output a real value $\hat{y}$, in classification, we seek a *classifier*, whose output $\hat{y}$ is the predicted class assignment.

For reasons that we'll get into as the book gets more technical, it's pretty hard to optimize a model that can only output a hard categorical assignment, e.g. either *cat* or *dog*. It's a lot easier instead to express the model in the language of probabilities. Given an example x, the model assigns a probability $\hat{y}_k$ to each label k. Because these are probabilities, they need to be positive numbers and add up to 1. This means that we only need $K - 1$ numbers to give the probabilities of K categories. This is easy to see for binary classification. If there's a 0.6 (60%) probability that an unfair coin comes up heads, then there's a 0.4 (40%) probability that it comes up tails. Returning to our animal classification example, a classifier might see an image and output the probability that the image is a cat $\Pr(y = \mathrm{cat}|x) = 0.9$. We can interpret this number by saying that the classifier is 90% sure that the image depicts a cat. The magnitude of the probability for the predicted class is one notion of confidence. It's not the only notion of confidence and we'll discuss different notions of uncertainty in more advanced chapters.

When we have more than two possible classes, we call the problem *multiclass classification*. Common examples include hand-written character recognition `[0, 1, 2, 3 ... 9, a, b, c, ...]`. While we attacked regression problems by trying to minimize the L1 or L2 loss functions, the common

loss function for classification problems is called cross-entropy. In MXNet Gluon, the corresponding loss function can be found here.

Note that the most likely class is not necessarily the one that you're going to use for your decision. Assume that you find this beautiful mushroom in your backyard:

Death cap - do not eat!

Now, assume that you built a classifier and trained it to predict if a mushroom is poisonous based on a photograph. Say our poison-detection classifier outputs $\Pr(y = \text{deathcap}|\text{image}) = 0.2$. In other words, the classifier is 80% confident that our mushroom *is not* a death cap. Still, you'd have to be a fool to eat it. That's because the certain benefit of a delicious dinner isn't worth a 20% risk of dying from it. In other words, the effect of the *uncertain risk* by far outweighs the benefit. Let's look at this in math. Basically, we need to compute the expected risk that we incur, i.e. we need to multiply the probability of the outcome with the benefit (or harm) associated with it:

$$L(\text{action}|x) = \mathbf{E}_{y \sim p(y|x)}[\text{loss}(\text{action}, y)]$$

Hence, the loss L incurred by eating the mushroom is $L(a = \text{eat}|x) = 0.2 * \infty + 0.8 * 0 = \infty$, whereas the cost of discarding it is $L(a = \text{discard}|x) = 0.2 * 0 + 0.8 * 1 = 0.8$.

Our caution was justified: as any mycologist would tell us, the above mushroom actually *is* a death cap. Classification can get much more complicated than just binary, multiclass, of even multi-label classification. For instance, there are some variants of classification for addressing hierarchies. Hierarchies assume that there exist some relationships among the many classes. So not all errors are equal - we prefer to mis-classify to a related class than to a distant class. Usually, this is referred to as *hierarchical classification*. One early example is due to Linnaeus, who organized the animals in a hierarchy.

Extant Shark Orders
Recognisable external characteristics
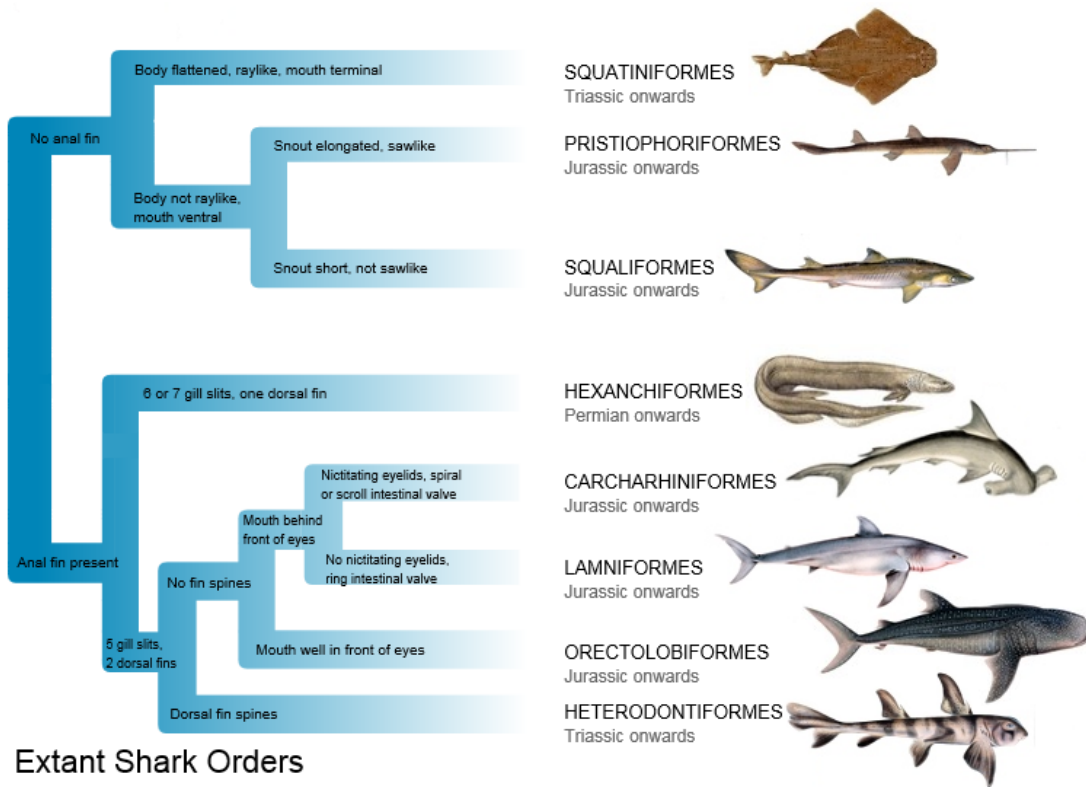
In the case of animal classification, it might not be so bad to mistake a poodle for a schnauzer, but our model would pay a huge penalty if it confused a poodle for a dinosaur. Which hierarchy is relevant might depend on how you plan to use the model. For example, rattle snakes and garter snakes might be close on the phylogenetic tree, but mistaking a rattler for a garter could be deadly.

Tagging

Some classification problems don't fit neatly into the binary or multiclass classification setups. For example, we could train a normal binary classifier to distinguish cats from dogs. Given the current state of computer vision, we can do this easily, with off-the-shelf tools. Nonetheless, no matter how accurate our model gets, we might find ourselves in trouble when the classifier encounters an image of the Town Musicians of Bremen.

As you can see, there's a cat in the picture, and a rooster, a dog and a donkey, with some trees in the background. Depending on what we want to do with our model ultimately, treating this as a binary classification problem might not make a lot of sense. Instead, we might want to give the model the option of saying the image depicts a cat *and* a dog *and* a donkey *and* a rooster.

The problem of learning to predict classes that are *not mutually exclusive* is called multi-label classifica-

tion. Auto-tagging problems are typically best described as multi-label classification problems. Think of the tags people might apply to posts on a tech blog, e.g., machine learning', technology', gadgets', programming languages', linux', cloud computing', AWS'. A typical article might have 5-10 tags applied because these concepts are correlated. Posts about cloud computing' are likely to mention AWS' and posts about machine learning' could also deal with programming languages'.

We also have to deal with this kind of problem when dealing with the biomedical literature, where correctly tagging articles is important because it allows researchers to do exhaustive reviews of the literature. At the National Library of Medicine, a number of professional annotators go over each article that gets indexed in PubMed to associate it with the relevant terms from MeSH, a collection of roughly 28k tags. This is a time-consuming process and the annotators typically have a one year lag between archiving and tagging. Machine learning can be used here to provide provisional tags until each article can have a proper manual review. Indeed, for several years, the BioASQ organization has hosted a competition to do precisely this.

Search and ranking

Sometimes we don't just want to assign each example to a bucket or to a real value. In the field of information retrieval, we want to impose a ranking on a set of items. Take web search for example, the goal is less to determine whether a particular page is relevant for a query, but rather, which one of the plethora of search results should be displayed for the user. We really care about the ordering of the relevant search results and our learning algorithm needs to produce ordered subsets of elements from a larger set. In other words, if we are asked to produce the first 5 letters from the alphabet, there is a difference between returning `A B C D E` and `C A B E D`. Even if the result set is the same, the ordering within the set matters nonetheless.

One possible solution to this problem is to score every element in the set of possible sets along with a corresponding relevance score and then to retrieve the top-rated elements. PageRank is an early example of such a relevance score. One of the peculiarities is that it didn't depend on the actual query. Instead, it simply helped to order the results that contained the query terms. Nowadays search engines use machine learning and behavioral models to obtain query-dependent relevance scores. There are entire conferences devoted to this subject.

Recommender systems

Recommender systems are another problem setting that is related to search and ranking. The problems are similar insofar as the goal is to display a set of relevant items to the user. The main difference is the emphasis on *personalization* to specific users in the context of recommender systems. For instance, for movie recommendations, the results page for a SciFi fan and the results page for a connoisseur of Woody Allen comedies might differ significantly.

Such problems occur, e.g. for movie, product or music recommendation. In some cases, customers will provide explicit details about how much they liked the product (e.g. Amazon product reviews). In some other cases, they might simply provide feedback if they are dissatisfied with the result (skipping titles

on a playlist). Generally, such systems strive to estimate some score y_{ij}, such as an estimated rating or probability of purchase, given a user u_i and product p_j.

Given such a model, then for any given user, we could retrieve the set of objects with the largest scores y_{ij}, which are then used as a recommendation. Production systems are considerably more advanced and take detailed user activity and item characteristics into account when computing such scores. The following image is an example of deep learning books recommended by Amazon based on personalization algorithms tuned to the author's preferences.
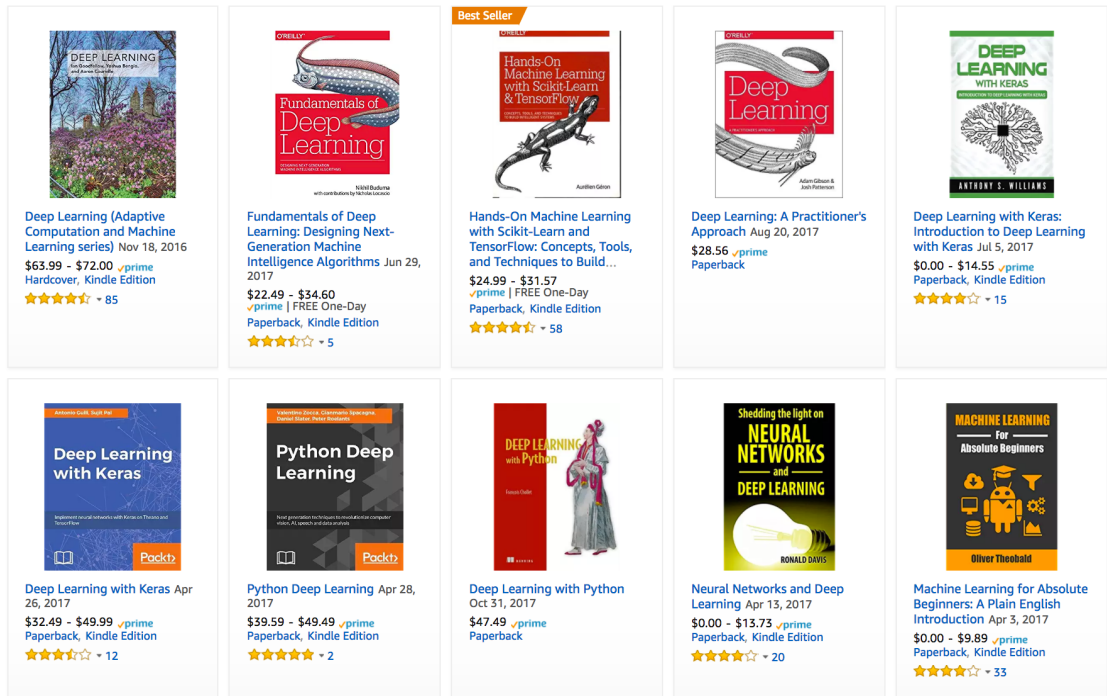
Sequence Learning

So far we've looked at problems where we have some fixed number of inputs and produce a fixed number of outputs. Before we considered predicting home prices from a fixed set of features: square footage, number of bedrooms, number of bathrooms, walking time to downtown. We also discussed mapping from an image (of fixed dimension), to the predicted probabilities that it belongs to each of a fixed number of classes, or taking a user ID and a product ID, and predicting a star rating. In these cases, once we feed our fixed-length input into the model to generate an output, the model immediately forgets what it just saw.

This might be fine if our inputs truly all have the same dimensions and if successive inputs truly have nothing to do with each other. But how would we deal with video snippets? In this case, each snippet might consist of a different number of frames. And our guess of what's going on in each frame might be much stronger if we take into account the previous or succeeding frames. Same goes for language.

One popular deep learning problem is machine translation: the task of ingesting sentences in some source language and predicting their translation in another language.

These problems also occur in medicine. We might want a model to monitor patients in the intensive care unit and to fire off alerts if their risk of death in the next 24 hours exceeds some threshold. We definitely wouldn't want this model to throw away everything it knows about the patient history each hour, and just make its predictions based on the most recent measurements.

These problems are among the more exciting applications of machine learning and they are instances of *sequence learning*. They require a model to either ingest sequences of inputs or to emit sequences of outputs (or both!). These latter problems are sometimes referred to as `seq2seq` problems. Language translation is a `seq2seq` problem. Transcribing text from spoken speech is also a `seq2seq` problem. While it is impossible to consider all types of sequence transformations, a number of special cases are worth mentioning:

Tagging and Parsing

This involves annotating a text sequence with attributes. In other words, the number of inputs and outputs is essentially the same. For instance, we might want to know where the verbs and subjects are. Alternatively, we might want to know which words are the named entities. In general, the goal is to decompose and annotate text based on structural and grammatical assumptions to get some annotation. This sounds more complex than it actually is. Below is a very simple example of annotating a sentence with tags indicating which words refer to named entities.

Tom	has	dinner	in	Washington	with	Sally.
Ent	•	•	•	Ent	•	Ent

Automatic Speech Recognition

With speech recognition, the input sequence x is the sound of a speaker, and the output y is the textual transcript of what the speaker said. The challenge is that there are many more audio frames (sound is typically sampled at 8kHz or 16kHz) than text, i.e. there is no 1:1 correspondence between audio and text, since thousands of samples correspond to a single spoken word. These are `seq2seq` problems where the output is much shorter than the input.

```
-D-e-e-p- L-ea-r-ni-ng-
```
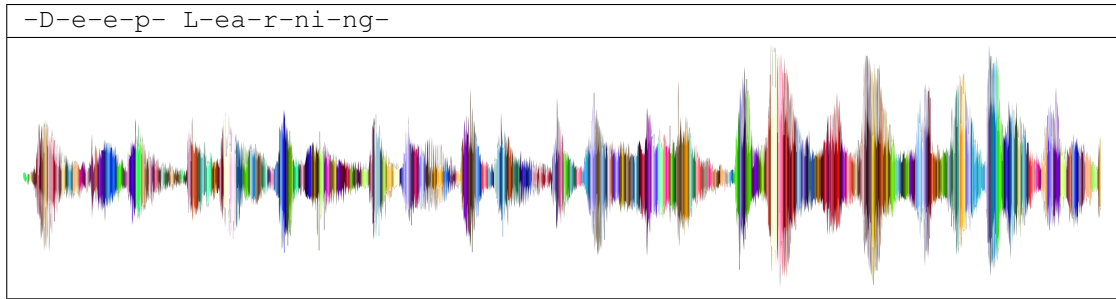
Text to Speech

Text-to-Speech (TTS) is the inverse of speech recognition. In other words, the input x is text and the output y is an audio file. In this case, the output is *much longer* than the input. While it is easy for *humans* to recognize a bad audio file, this isn't quite so trivial for computers.

Machine Translation

Unlike the case of speech recognition, where corresponding inputs and outputs occur in the same order (after alignment), in machine translation, order inversion can be vital. In other words, while we are still converting one sequence into another, neither the number of inputs and outputs nor the order of corresponding data points are assumed to be the same. Consider the following illustrative example of the obnoxious tendency of Germans (*Alex writing here*) to place the verbs at the end of sentences.

German	Haben Sie sich schon dieses grossartige Lehrwerk angeschaut?
English	Did you already check out this excellent tutorial?
Wrong alignment	Did you yourself already this excellent tutorial looked-at?

A number of related problems exist. For instance, determining the order in which a user reads a webpage is a two-dimensional layout analysis problem. Likewise, for dialogue problems, we need to take world-knowledge and prior state into account. This is an active area of research.

1.3.2 Unsupervised learning

All the examples so far were related to *Supervised Learning*, i.e. situations where we feed the model a bunch of examples and a bunch of *corresponding target values*. You could think of supervised learning as having an extremely specialized job and an extremely anal boss. The boss stands over your shoulder and tells you exactly what to do in every situation until you learn to map from situations to actions. Working for such a boss sounds pretty lame. On the other hand, it's easy to please this boss. You just recognize the pattern as quickly as possible and imitate their actions.

In a completely opposite way, it could be frustrating to work for a boss who has no idea what they want you to do. However, if you plan to be a data scientist, you'd better get used to it. The boss might just hand you a giant dump of data and tell you to *do some data science with it!* This sounds vague because it is. We call this class of problems *unsupervised learning*, and the type and number of questions we could ask is limited only by our creativity. We will address a number of unsupervised learning techniques in later chapters. To whet your appetite for now, we describe a few of the questions you might ask:

- Can we find a small number of prototypes that accurately summarize the data? Given a set of photos, can we group them into landscape photos, pictures of dogs, babies, cats, mountain peaks, etc.? Likewise, given a collection of users' browsing activity, can we group them into users with similar behavior? This problem is typically known as **clustering**.

- Can we find a small number of parameters that accurately capture the relevant properties of the data? The trajectories of a ball are quite well described by velocity, diameter, and mass of the ball. Tailors have developed a small number of parameters that describe human body shape fairly accurately for the purpose of fitting clothes. These problems are referred to as **subspace estimation** problems. If the dependence is linear, it is called **principal component analysis**.

- Is there a representation of (arbitrarily structured) objects in Euclidean space (i.e. the space of vectors in $\mathbb{R}^n$) such that symbolic properties can be well matched? This is called **representation learning** and it is used to describe entities and their relations, such as Rome - Italy + France = Paris.

- Is there a description of the root causes of much of the data that we observe? For instance, if we have demographic data about house prices, pollution, crime, location, education, salaries, etc., can we discover how they are related simply based on empirical data? The field of **directed graphical models** and **causality** deals with this.

- An important and exciting recent development is **generative adversarial networks**. They are basically a procedural way of synthesizing data. The underlying statistical mechanisms are tests to check whether real and fake data are the same. We will devote a few notebooks to them.

1.3.3 Interacting with an Environment

So far, we haven't discussed where data actually comes from, or what actually *happens* when a machine learning model generates an output. That's because supervised learning and unsupervised learning do not address these issues in a very sophisticated way. In either case, we grab a big pile of data up front, then do our pattern recognition without ever interacting with the environment again. Because all of the learning takes place after the algorithm is disconnected from the environment, this is called *offline learning*. For supervised learning, the process looks like this:
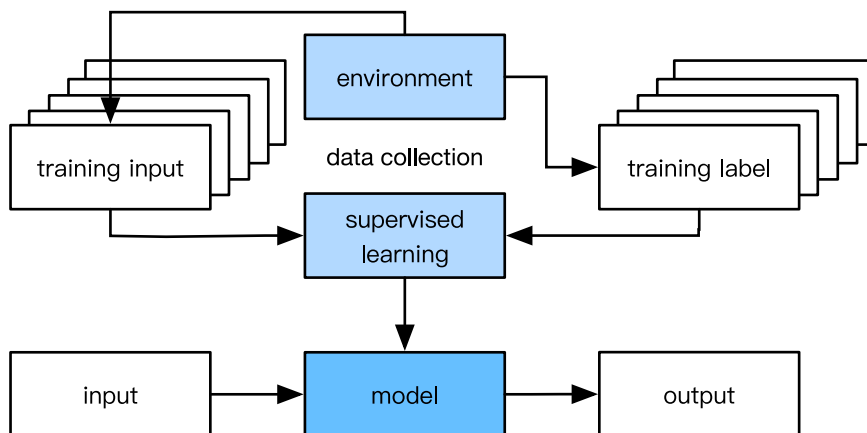
This simplicity of offline learning has its charms. The upside is we can worry about pattern recognition in isolation without these other problems to deal with, but the downside is that the problem formulation is quite limiting. If you are more ambitious, or if you grew up reading Asimov's Robot Series, then you might imagine artificially intelligent bots capable not only of making predictions, but of taking actions in the world. We want to think about intelligent *agents*, not just predictive *models*. That means we need to think about choosing *actions*, not just making *predictions*. Moreover, unlike predictions, actions actually impact the environment. If we want to train an intelligent agent, we must account for the way its actions might impact the future observations of the agent.

Considering the interaction with an environment opens a whole set of new modeling questions. Does the environment:

- remember what we did previously?

- want to help us, e.g. a user reading text into a speech recognizer?

- want to beat us, i.e. an adversarial setting like spam filtering (against spammers) or playing a game (vs an opponent)?

- not care (as in most cases)?

- have shifting dynamics (steady vs. shifting over time)?

This last question raises the problem of *covariate shift*, (when training and test data are different). It's a problem that most of us have experienced when taking exams written by a lecturer, while the homeworks were composed by his TAs. We'll briefly describe reinforcement learning and adversarial learning, two settings that explicitly consider interaction with an environment.

1.3.4 Reinforcement learning

If you're interested in using machine learning to develop an agent that interacts with an environment and takes actions, then you're probably going to wind up focusing on *reinforcement learning* (RL). This might include applications to robotics, to dialogue systems, and even to developing AI for video games.

Deep reinforcement learning (DRL), which applies deep neural networks to RL problems, has surged in popularity. The breakthrough deep Q-network that beat humans at Atari games using only the visual input , and the AlphaGo program that dethroned the world champion at the board game Go are two prominent examples.

Reinforcement learning gives a very general statement of a problem, in which an agent interacts with an environment over a series of *time steps*. At each time step t, the agent receives some observation o_t from the environment, and must choose an action a_t which is then transmitted back to the environment. Finally, the agent receives a reward r_t from the environment. The agent then receives a subsequent observation, and chooses a subsequent action, and so on. The behavior of an RL agent is governed by a *policy*. In short, a *policy* is just a function that maps from observations (of the environment) to actions. The goal of reinforcement learning is to produce a good policy.
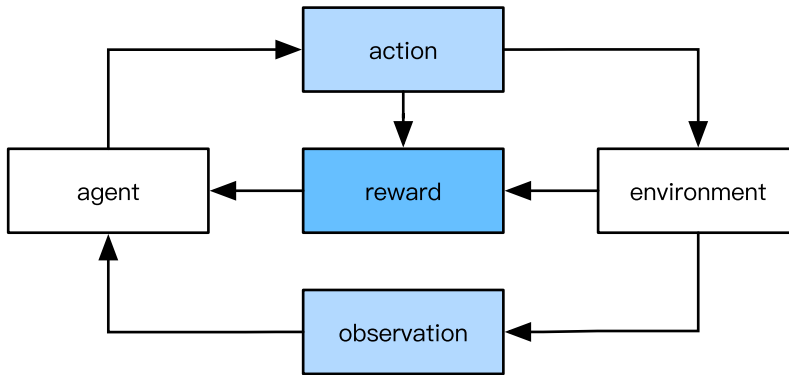
It's hard to overstate the generality of the RL framework. For example, we can cast any supervised learning problem as an RL problem. Say we had a classification problem. We could create an RL agent with one *action* corresponding to each class. We could then create an environment which gave a reward that was exactly equal to the loss function from the original supervised problem.

That being said, RL can also address many problems that supervised learning cannot. For example, in supervised learning we always expect that the training input comes associated with the correct label. But in RL, we don't assume that for each observation, the environment tells us the optimal action. In general, we just get some reward. Moreover, the environment may not even tell us which actions led to the reward.

Consider for example the game of chess. The only real reward signal comes at the end of the game when we either win, which we might assign a reward of 1, or when we lose, which we could assign a reward of -1. So reinforcement learners must deal with the *credit assignment problem*. The same goes for an employee who gets a promotion on October 11. That promotion likely reflects a large number of well-chosen actions over the previous year. Getting more promotions in the future requires figuring out what actions along the way led to the promotion.

Reinforcement learners may also have to deal with the problem of partial observability. That is, the current observation might not tell you everything about your current state. Say a cleaning robot found itself trapped in one of many identical closets in a house. Inferring the precise location (and thus state) of the robot might require considering its previous observations before entering the closet.

Finally, at any given point, reinforcement learners might know of one good policy, but there might be many other better policies that the agent has never tried. The reinforcement learner must constantly choose whether to *exploit* the best currently-known strategy as a policy, or to *explore* the space of strategies, potentially giving up some short-run reward in exchange for knowledge.

MDPs, bandits, and friends

The general reinforcement learning problem is a very general setting. Actions affect subsequent observations. Rewards are only observed corresponding to the chosen actions. The environment may be either fully or partially observed. Accounting for all this complexity at once may ask too much of researchers. Moreover not every practical problem exhibits all this complexity. As a result, researchers have studied a number of *special cases* of reinforcement learning problems.

When the environment is fully observed, we call the RL problem a *Markov Decision Process* (MDP). When the state does not depend on the previous actions, we call the problem a *contextual bandit problem*. When there is no state, just a set of available actions with initially unknown rewards, this problem is the classic *multi-armed bandit problem*.

1.4 Roots

Although deep learning is a recent invention, humans have held the desire to analyze data and to predict future outcomes for centuries. In fact, much of natural science has its roots in this. For instance, the Bernoulli distribution is named after Jacob Bernoulli (1655-1705), and the Gaussian distribution was discovered by Carl Friedrich Gauss (1777-1855). He invented for instance the least mean squares algorithm, which is still used today for a range of problems from insurance calculations to medical diagnostics. These tools gave rise to an experimental approach in natural sciences - for instance, Ohm's law relating current and voltage in a resistor is perfectly described by a linear model.

Even in the middle ages mathematicians had a keen intuition of estimates. For instance, the geometry book of Jacob Köbel (1460-1533) illustrates averaging the length of 16 adult men's feet to obtain the average foot length.

Fig. 1.1: Estimating the length of a foot

Figure 1.1 illustrates how this estimator works. 16 adult men were asked to line up in a row, when leaving church. Their aggregate length was then divided by 16 to obtain an estimate for what now amounts to 1 foot. This algorithm' was later improved to deal with misshapen feet - the 2 men with the shortest and longest feet respectively were sent away, averaging only over the remainder. This is one of the earliest examples of the trimmed mean estimate.

Statistics really took off with the collection and availability of data. One of its titans, Ronald Fisher (1890-1962), contributed significantly to its theory and also its applications in genetics. Many of his algorithms (such as Linear Discriminant Analysis) and formulae (such as the Fisher Information Matrix) are still in frequent use today (even the Iris dataset that he released in 1936 is still used sometimes to illustrate machine learning algorithms).

A second influence for machine learning came from Information Theory (Claude Shannon, 1916-2001) and the Theory of computation via Alan Turing (1912-1954). Turing posed the question can machines

think? in his famous paper Computing machinery and intelligence (Mind, October 1950). In what he described as the Turing test, a machine can be considered intelligent if it is difficult for a human evaluator to distinguish between the replies from a machine and a human being through textual interactions. To this day, the development of intelligent machines is changing rapidly and continuously.

Another influence can be found in neuroscience and psychology. After all, humans clearly exhibit intelligent behavior. It is thus only reasonable to ask whether one could explain and possibly reverse engineer these insights. One of the oldest algorithms to accomplish this was formulated by Donald Hebb (1904-1985).

In his groundbreaking book The Organization of Behavior (John Wiley & Sons, 1949) he posited that neurons learn by positive reinforcement. This became known as the Hebbian learning rule. It is the prototype of Rosenblatt's perceptron learning algorithm and it laid the foundations of many stochastic gradient descent algorithms that underpin deep learning today: reinforce desirable behavior and diminish undesirable behavior to obtain good weights in a neural network.

Biological inspiration is what gave Neural Networks its name. For over a century (dating back to the models of Alexander Bain, 1873 and James Sherrington, 1890) researchers have tried to assemble computational circuits that resemble networks of interacting neurons. Over time the interpretation of biology became more loose but the name stuck. At its heart lie a few key principles that can be found in most networks today:

- The alternation of linear and nonlinear processing units, often referred to as layers'.

- The use of the chain rule (aka backpropagation) for adjusting parameters in the entire network at once.

After initial rapid progress, research in Neural Networks languished from around 1995 until 2005. This was due to a number of reasons. Training a network is computationally very expensive. While RAM was plentiful at the end of the past century, computational power was scarce. Secondly, datasets were relatively small. In fact, Fisher's Iris dataset' from 1932 was a popular tool for testing the efficacy of algorithms. MNIST with its 60,000 handwritten digits was considered huge.

Given the scarcity of data and computation, strong statistical tools such as Kernel Methods, Decision Trees and Graphical Models proved empirically superior. Unlike Neural Networks they did not require weeks to train and provided predictable results with strong theoretical guarantees.

1.5 The Road to Deep Learning

Much of this changed with the ready availability of large amounts of data, due to the World Wide Web, the advent of companies serving hundreds of millions of users online, a dissemination of cheap, high quality sensors, cheap data storage (Kryder's law), and cheap computation (Moore's law), in particular in the form of GPUs, originally engineered for computer gaming. Suddenly algorithms and models that seemed computationally infeasible became relevant (and vice versa). This is best illustrated in the table below:

Decade	Dataset	Memory	Floating Point Calculations per Second
1970	100 (Iris)	1 KB	100 KF (Intel 8080)
1980	1 K (House prices in Boston)	100 KB	1 MF (Intel 80186)
1990	10 K (optical character recognition)	10 MB	10 MF (Intel 80486)
2000	10 M (web pages)	100 MB	1 GF (Intel Core)
2010	10 G (advertising)	1 GB	1 TF (Nvidia C2050)
2020	1 T (social network)	100 GB	1 PF (Nvidia DGX-2)

It is quite evident that RAM has not kept pace with the growth in data. At the same time, the increase in computational power has outpaced that of the data available. This means that statistical models needed to become more memory efficient (this is typically achieved by adding nonlinearities) while simultaneously being able to spend more time on optimizing these parameters, due to an increased compute budget. Consequently the sweet spot in machine learning and statistics moved from (generalized) linear models and kernel methods to deep networks. This is also one of the reasons why many of the mainstays of deep learning, such as Multilayer Perceptrons (e.g. McCulloch & Pitts, 1943), Convolutional Neural Networks (Le Cun, 1992), Long Short Term Memory (Hochreiter & Schmidhuber, 1997), Q-Learning (Watkins, 1989), were essentially rediscovered' in the past decade, after laying dormant for considerable time.

The recent progress in statistical models, applications, and algorithms, has sometimes been likened to the Cambrian Explosion: a moment of rapid progress in the evolution of species. Indeed, the state of the art is not just a mere consequence of available resources, applied to decades old algorithms. Note that the list below barely scratches the surface of the ideas that have helped researchers achieve tremendous progress over the past decade.

- Novel methods for capacity control, such as Dropout [3] allowed for training of relatively large networks without the danger of overfitting, i.e. without the danger of merely memorizing large parts of the training data. This was achieved by applying noise injection [4] throughout the network, replacing weights by random variables for training purposes.

- Attention mechanisms solved a second problem that had plagued statistics for over a century: how to increase the memory and complexity of a system without increasing the number of learnable parameters. [5] found an elegant solution by using what can only be viewed as a learnable pointer structure. That is, rather than having to remember an entire sentence, e.g. for machine translation in a fixed-dimensional representation, all that needed to be stored was a pointer to the intermediate state of the translation process. This allowed for significantly increased accuracy for long sentences, since the model no longer needed to remember the entire sentence before beginning to generate sentences.

- Multi-stage designs, e.g. via the Memory Networks [6] and the Neural Programmer-Interpreter [7] allowed statistical modelers to describe iterative approaches to reasoning. These tools allow for an internal state of the deep network to be modified repeatedly, thus carrying out subsequent steps in a chain of reasoning, similar to how a processor can modify memory for a computation.

- Another key development was the invention of Generative Adversarial Networks [8]. Traditionally

statistical methods for density estimation and generative models focused on finding proper probability distributions and (often approximate) algorithms for sampling from them. As a result, these algorithms were largely limited by the lack of flexibility inherent in the statistical models. The crucial innovation in GANs was to replace the sampler by an arbitrary algorithm with differentiable parameters. These are then adjusted in such a way that the discriminator (effectively a two-sample test) cannot distinguish fake from real data. Through the ability to use arbitrary algorithms to generate data it opened up density estimation to a wide variety of techniques. Examples of galloping Zebras [9] and of fake celebrity faces [10] are both testimony to this progress.

- In many cases a single GPU is insufficient to process the large amounts of data available for training. Over the past decade the ability to build parallel distributed training algorithms has improved significantly. One of the key challenges in designing scalable algorithms is that the workhorse of deep learning optimization, stochastic gradient descent, relies on relatively small minibatches of data to be processed. At the same time, small batches limit the efficiency of GPUs. Hence, training on 1024 GPUs with a minibatch size of, say 32 images per batch amounts to an aggregate minibatch of 32k images. Recent work, first by Li [11], and subsequently by You et al. [12] and Jia et al. [13] pushed the size up to 64k observations, reducing training time for ResNet50 on ImageNet to less than 7 minutes. For comparison - initially training times were measured in the order of days.

- The ability to parallelize computation has also contributed quite crucially to progress in reinforcement learning, at least whenever simulation is an option. This has led to significant progress in computers achieving superhuman performance in Go, Atari games, Starcraft, and in physics simulations (e.g. using MuJoCo). See e.g. Silver et al. [18] for a description of how to achieve this in AlphaGo. In a nutshell, reinforcement learning works best if plenty of (state, action, reward) triples are available, i.e. whenever it is possible to try out lots of things to learn how they relate to each other. Simulation provides such an avenue.

- Deep Learning frameworks have played a crucial role in disseminating ideas. The first generation of frameworks allowing for easy modeling encompassed Caffe, Torch, and Theano. Many seminal papers were written using these tools. By now they have been superseded by TensorFlow, often used via its high level API Keras, CNTK, Caffe 2, and Apache MxNet. The third generation of tools, namely imperative tools for deep learning, was arguably spearheaded by Chainer, which used a syntax similar to Python NumPy to describe models. This idea was adopted by PyTorch and the Gluon API of MxNet. It is the latter that this course uses to teach Deep Learning.

The division of labor between systems researchers building better tools for training and statistical modelers building better networks has greatly simplified things. For instance, training a linear logistic regression model used to be a nontrivial homework problem, worthy to give to new Machine Learning PhD students at Carnegie Mellon University in 2014. By now, this task can be accomplished with less than 10 lines of code, putting it firmly into the grasp of programmers.

1.6 Success Stories

Artificial Intelligence has a long history of delivering results that would be difficult to accomplish otherwise. For instance, mail is sorted using optical character recognition. These systems have been deployed

since the 90s (this is, after all, the source of the famous MNIST and USPS sets of handwritten digits). The same applies to reading checks for bank deposits and scoring creditworthiness of applicants. Financial transactions are checked for fraud automatically. This forms the backbone of many e-commerce payment systems, such as PayPal, Stripe, AliPay, WeChat, Apple, Visa, MasterCard. Computer programs for chess have been competitive for decades. Machine learning feeds search, recommendation, personalization and ranking on the internet. In other words, artificial intelligence and machine learning are pervasive, albeit often hidden from sight.

It is only recently that AI has been in the limelight, mostly due to solutions to problems that were considered intractable previously.

- Intelligent assistants, such as Apple's Siri, Amazon's Alexa, or Google's assistant are able to answer spoken questions with a reasonable degree of accuracy. This includes menial tasks such as turning on light switches (a boon to the disabled) up to making barber's appointments and offering phone support dialog. This is likely the most noticeable sign that AI is affecting our lives.

- A key ingredient in digital assistants is the ability to recognize speech accurately. Gradually the accuracy of such systems has increased to the point where they reach human parity [14] for certain applications.

- Object recognition likewise has come a long way. Estimating the object in a picture was a fairly challenging task in 2010. On the ImageNet benchmark Lin et al. [15] achieved a top-5 error rate of 28%. By 2017 Hu et al. [16] reduced this error rate to 2.25%. Similarly stunning results have been achieved for identifying birds, or diagnosing skin cancer.

- Games used to be a bastion of human intelligence. Starting from TDGammon [23], a program for playing Backgammon using temporal difference (TD) reinforcement learning, algorithmic and computational progress has led to algorithms for a wide range of applications. Unlike Backgammon, chess has a much more complex state space and set of actions. DeepBlue beat Gary Kasparov, Campbell et al. [17], using massive parallelism, special purpose hardware and efficient search through the game tree. Go is more difficult still, due to its huge state space. AlphaGo reached human parity in 2015, Silver et al. [18] using Deep Learning combined with Monte Carlo tree sampling. The challenge in Poker was that the state space is large and it is not fully observed (we don't know the opponents' cards). Libratus exceeded human performance in Poker using efficiently structured strategies; Brown and Sandholm [19]. This illustrates the impressive progress in games and the fact that advanced algorithms played a crucial part in them.

- Another indication of progress in AI is the advent of self-driving cars and trucks. While full autonomy is not quite within reach yet, excellent progress has been made in this direction, with companies such as Momenta, Tesla, NVIDIA, MobilEye and Waymo shipping products that enable at least partial autonomy. What makes full autonomy so challenging is that proper driving requires the ability to perceive, to reason and to incorporate rules into a system. At present, Deep Learning is used primarily in the computer vision aspect of these problems. The rest is heavily tuned by engineers.

Again, the above list barely scratches the surface of what is considered intelligent and where machine learning has led to impressive progress in a field. For instance, robotics, logistics, computational biology, particle physics and astronomy owe some of their most impressive recent advances at least in parts to

machine learning. ML is thus becoming a ubiquitous tool for engineers and scientists.

Frequently the question of the AI apocalypse, or the AI singularity has been raised in non-technical articles on AI. The fear is that somehow machine learning systems will become sentient and decide independently from their programmers (and masters) about things that directly affect the livelihood of humans. To some extent AI already affects the livelihood of humans in an immediate way - creditworthiness is assessed automatically, autopilots mostly navigate cars safely, decisions about whether to grant bail use statistical data as input. More frivolously, we can ask Alexa to switch on the coffee machine and she will happily oblige, provided that the appliance is internet enabled.

Fortunately we are far from a sentient AI system that is ready to enslave its human creators (or burn their coffee). Firstly, AI systems are engineered, trained and deployed in a specific, goal oriented manner. While their behavior might give the illusion of general intelligence, it is a combination of rules, heuristics and statistical models that underlie the design. Second, at present tools for general Artificial Intelligence simply do not exist that are able to improve themselves, reason about themselves, and that are able to modify, extend and improve their own architecture while trying to solve general tasks.

A much more realistic concern is how AI is being used in our daily lives. It is likely that many menial tasks fulfilled by truck drivers and shop assistants can and will be automated. Farm robots will likely reduce the cost for organic farming but they will also automate harvesting operations. This phase of the industrial revolution will have profound consequences on large swaths of society (truck drivers and shop assistants are some of the most common jobs in many states). Furthermore, statistical models, when applied without care can lead to racial, gender or age bias. It is important to ensure that these algorithms are used with great care. This is a much bigger concern than to worry about a potentially malevolent superintelligence intent on destroying humanity.

Summary

- Machine learning studies how computer systems can use data to improve performance. It combines ideas from statistics, data mining, artificial intelligence and optimization. Often it is used as a means of implementing artificially intelligent solutions.

- As a class of machine learning, representational learning focuses on how to automatically find the appropriate way to represent data. This is often accomplished by a progression of learned transformations.

- Much of the recent progress has been triggered by an abundance of data arising from cheap sensors and internet scale applications, and by significant progress in computation, mostly through GPUs.

- Whole system optimization is a key component in obtaining good performance. The availability of efficient deep learning frameworks has made design and implementation of this significantly easier.

Exercises

1. Which parts of code that you are currently writing could be learned', i.e. improved by learning and automatically determining design choices that are made in your code? Does your code include heuristic design choices?

2. Which problems that you encounter have many examples for how to solve them, yet no specific way to automate them? These may be prime candidates for using Deep Learning.

3. Viewing the development of Artificial Intelligence as a new industrial revolution, what is the relationship between algorithms and data? Is it similar to steam engines and coal (what is the fundamental difference)?

4. Where else can you apply the end-to-end training approach? Physics? Engineering? Econometrics?

References

[1] Turing, A. M. (1950). Computing machinery and intelligence. Mind, 59(236), 433.

[2] Hebb, D. O. (1949). The organization of behavior; a neuropsychological theory. A Wiley Book in Clinical Psychology. 62-78.

[3] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The Journal of Machine Learning Research, 15(1), 1929-1958.

[4] Bishop, C. M. (1995). Training with noise is equivalent to Tikhonov regularization. Neural computation, 7(1), 108-116.

[5] Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473.

[6] Sukhbaatar, S., Weston, J., & Fergus, R. (2015). End-to-end memory networks. In Advances in neural information processing systems (pp. 2440-2448).

[7] Reed, S., & De Freitas, N. (2015). Neural programmer-interpreters. arXiv preprint arXiv:1511.06279.

[8] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., & Bengio, Y. (2014). Generative adversarial nets. In Advances in neural information processing systems (pp. 2672-2680).

[9] Zhu, J. Y., Park, T., Isola, P., & Efros, A. A. (2017). Unpaired image-to-image translation using cycle-consistent adversarial networks. arXiv preprint.

[10] Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2017). Progressive growing of gans for improved quality, stability, and variation. arXiv preprint arXiv:1710.10196.

[11] Li, M. (2017). Scaling Distributed Machine Learning with System and Algorithm Co-design (Doctoral dissertation, PhD thesis, Intel).

[12] You, Y., Gitman, I., & Ginsburg, B. Large batch training of convolutional networks. ArXiv e-prints.

[13] Jia, X., Song, S., He, W., Wang, Y., Rong, H., Zhou, F., & Chen, T. (2018). Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes. arXiv preprint arXiv:1807.11205.

[14] Xiong, W., Droppo, J., Huang, X., Seide, F., Seltzer, M., Stolcke, A., & Zweig, G. (2017, March). The Microsoft 2016 conversational speech recognition system. In Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on (pp. 5255-5259). IEEE.

[15] Lin, Y., Lv, F., Zhu, S., Yang, M., Cour, T., Yu, K., & Huang, T. (2010). Imagenet classification: fast descriptor coding and large-scale svm training. Large scale visual recognition challenge.

[16] Hu, J., Shen, L., & Sun, G. (2017). Squeeze-and-excitation networks. arXiv preprint arXiv:1709.01507, 7.

[17] Campbell, M., Hoane Jr, A. J., & Hsu, F. H. (2002). Deep blue. Artificial intelligence, 134 (1-2), 57-83.

[18] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., & Dieleman, S. (2016). Mastering the game of Go with deep neural networks and tree search. Nature, 529 (7587), 484.

[19] Brown, N., & Sandholm, T. (2017, August). Libratus: The superhuman ai for no-limit poker. In Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence.

[20] Canny, J. (1986). A computational approach to edge detection. IEEE Transactions on pattern analysis and machine intelligence, (6), 679-698.

[21] Lowe, D. G. (2004). Distinctive image features from scale-invariant keypoints. International journal of computer vision, 60(2), 91-110.

[22] Salton, G., & McGill, M. J. (1986). Introduction to modern information retrieval.

[23] Tesauro, G. (1995), Transactions of the ACM, (38) 3, 58-68

Scan the QR Code to Discuss

2

Crashcourse in the Preliminaries

If you are in a hurry, this chapter has all the details you need to get a taste of deep learning. If you like to take the scenic route, we still recommend that you read it, since it will explain how to set up Apache MXNet, how to use automatic differentiation and how to manipulate data and memory. We also provide quick primers on linear algebra and statistics, designed to help readers to get up to speed on the basics.

2.1 Data Manipulation

It's impossible to get anything done if we can't manipulate data. Generally, there are two important things we need to do with data: (i) acquire it and (ii) process it once it's inside the computer. There's no point in trying to acquire data if we don't even know how to store it, so let's get our hands dirty first by playing with synthetic data.

We'll start by introducing NDArrays, MXNet's primary tool for storing and transforming data. If you've worked with NumPy before, you'll notice that NDArrays are, by design, similar to NumPy's multi-dimensional array. However, they confer a few key advantages. First, NDArrays support asynchronous computation on CPU, GPU, and distributed cloud architectures. Second, they provide support for automatic differentiation. These properties make NDArray an ideal ingredient for machine learning.

2.1.1 Getting Started

In this chapter, we'll get you going with the basic functionality. Don't worry if you don't understand any of the basic math, like element-wise operations or normal distributions. In the next two chapters we'll take another pass at NDArray, teaching you both the math you'll need and how to realize it in code. For even more math, see the *Math* section in the appendix.

We begin by importing MXNet and the `ndarray` module from MXNet. Here, `nd` is short for `ndarray`.

```
In [1]: import mxnet as mx
        from mxnet import nd
```

The simplest object we can create is a vector. `arange` creates a row vector of 12 consecutive integers.

```
In [2]: x = nd.arange(12)
        x
Out[2]:
        [ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9. 10. 11.]
        <NDArray 12 @cpu(0)>
```

From the property `<NDArray 12 @cpu(0)>` shown when printing `x` we can see that it is a one-dimensional array of length 12 and that it resides in CPU main memory. The 0 in '@cpu(0)'' has no special meaning and does not represent a specific core.

We can get the NDArray instance shape through the `shape` property.

```
In [3]: x.shape
Out[3]: (12,)
```

We can also get the total number of elements in the NDArray instance through the `size` property. Since we are dealing with a vector, both are identical.

```
In [4]: x.size
Out[4]: 12
```

In the following, we use the `reshape` function to change the shape of the line vector `x` to (3, 4), which is a matrix of 3 rows and 4 columns. Except for the shape change, the elements in `x` (and also its size) remain unchanged.

```
In [5]: x = x.reshape((3, 4))
        x
Out[5]:
        [[ 0.  1.  2.  3.]
         [ 4.  5.  6.  7.]
         [ 8.  9. 10. 11.]]
        <NDArray 3x4 @cpu(0)>
```

It can be awkward to reshape a matrix in the way described above. After all, if we want a matrix with 3 rows we also need to know that it should have 4 columns in order to make up 12 elements. Or we might want to request NDArray to figure out automatically how to give us a matrix with 4 columns and whatever number of rows that are needed to take care of all elements. This is precisely what the entry −1 does in

any one of the fields. That is, in our case `x.reshape((3, 4))` is equivalent to `x.reshape((-1, 4))` and `x.reshape((3, -1))`.

```
In [6]: nd.empty((3, 4))
```
```
Out[6]:
        [[ 9.8411094e-17  4.5762204e-41 -1.5210272e-14  3.0801942e-41]
         [ 0.0000000e+00  0.0000000e+00  0.0000000e+00  0.0000000e+00]
         [ 0.0000000e+00  0.0000000e+00  0.0000000e+00  0.0000000e+00]]
        <NDArray 3x4 @cpu(0)>
```

The `empty` method just grabs some memory and hands us back a matrix without setting the values of any of its entries. This is very efficient but it means that the entries can have any form of values, including very big ones! But typically, we'll want our matrices initialized.

Commonly, we want one of all zeros. For objects with more than two dimensions mathematicians don't have special names - they simply call them tensors. To create one with all elements set to 0 and a shape of (2, 3, 4) we use

```
In [7]: nd.zeros((2, 3, 4))
```
```
Out[7]:
        [[[0. 0. 0. 0.]
          [0. 0. 0. 0.]
          [0. 0. 0. 0.]]

         [[0. 0. 0. 0.]
          [0. 0. 0. 0.]
          [0. 0. 0. 0.]]]
        <NDArray 2x3x4 @cpu(0)>
```

Just like in NumPy, creating tensors with each element being 1 works via

```
In [8]: nd.ones((2, 3, 4))
```
```
Out[8]:
        [[[1. 1. 1. 1.]
          [1. 1. 1. 1.]
          [1. 1. 1. 1.]]

         [[1. 1. 1. 1.]
          [1. 1. 1. 1.]
          [1. 1. 1. 1.]]]
        <NDArray 2x3x4 @cpu(0)>
```

We can also specify the value of each element in the NDArray that needs to be created through a Python list.

```
In [9]: y = nd.array([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
        y
```
```
Out[9]:
        [[2. 1. 4. 3.]
         [1. 2. 3. 4.]
         [4. 3. 2. 1.]]
        <NDArray 3x4 @cpu(0)>
```

In some cases, we need to randomly generate the value of each element in the NDArray. This is especially common when we intend to use the array as a parameter in a neural network. The following creates an NDArray with a shape of (3,4). Each of its elements is randomly sampled in a normal distribution with zero mean and unit variance.

```
In [10]: nd.random.normal(0, 1, shape=(3, 4))
Out[10]:
        [[ 2.2122064   0.7740038   1.0434405   1.1839255 ]
         [ 1.8917114  -1.2347414  -1.771029   -0.45138445]
         [ 0.57938355 -1.856082   -1.9768796  -0.20801921]]
        <NDArray 3x4 @cpu(0)>
```

2.1.2 Operations

Oftentimes, we want to apply functions to arrays. Some of the simplest and most useful functions are the element-wise functions. These operate by performing a single scalar operation on the corresponding elements of two arrays. We can create an element-wise function from any function that maps from the scalars to the scalars. In math notations we would denote such a function as $f : \mathbb{R} \to \mathbb{R}$. Given any two vectors $\mathbf{u}$ and $\mathbf{v}$ *of the same shape*, and the function f, we can produce a vector $\mathbf{c} = F(\mathbf{u}, \mathbf{v})$ by setting $c_i \leftarrow f(u_i, v_i)$ for all i. Here, we produced the vector-valued $F : \mathbb{R}^d \to \mathbb{R}^d$ by *lifting* the scalar function to an element-wise vector operation. In MXNet, the common standard arithmetic operators (+,-,/,*,**) have all been *lifted* to element-wise operations for identically-shaped tensors of arbitrary shape. We can call element-wise operations on any two tensors of the same shape, including matrices.

```
In [11]: x = nd.array([1, 2, 4, 8])
         y = nd.ones_like(x) * 2
         print('x =', x)
         print('x + y', x + y)
         print('x - y', x - y)
         print('x * y', x * y)
         print('x / y', x / y)

x =
[1. 2. 4. 8.]
<NDArray 4 @cpu(0)>
x + y
[ 3.  4.  6. 10.]
<NDArray 4 @cpu(0)>
x - y
[-1.  0.  2.  6.]
<NDArray 4 @cpu(0)>
x * y
[ 2.  4.  8. 16.]
<NDArray 4 @cpu(0)>
x / y
[0.5 1.  2.  4. ]
<NDArray 4 @cpu(0)>
```

Many more operations can be applied element-wise, such as exponentiation:

```
In [12]: x.exp()
```

```
Out[12]:
        [2.7182817e+00 7.3890562e+00 5.4598148e+01 2.9809580e+03]
        <NDArray 4 @cpu(0)>
```

In addition to computations by element, we can also perform matrix operations, like matrix multiplication using the `dot` function. Next, we will perform matrix multiplication of `x` and the transpose of `y`. We define `x` as a matrix of 3 rows and 4 columns, and `y` is transposed into a matrix of 4 rows and 3 columns. The two matrices are multiplied to obtain a matrix of 3 rows and 3 columns (if you're confused about what this means, don't worry - we will explain matrix operations in much more detail in the chapter on *linear algebra*).

```
In [13]: x = nd.arange(12).reshape((3,4))
         y = nd.array([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
         nd.dot(x, y.T)

Out[13]:
        [[ 18.  20.  10.]
         [ 58.  60.  50.]
         [ 98. 100.  90.]]
        <NDArray 3x3 @cpu(0)>
```

We can also merge multiple NDArrays. For that, we need to tell the system along which dimension to merge. The example below merges two matrices along dimension 0 (along rows) and dimension 1 (along columns) respectively.

```
In [14]: nd.concat(x, y, dim=0)
         nd.concat(x, y, dim=1)

Out[14]:
        [[ 0.  1.  2.  3.  2.  1.  4.  3.]
         [ 4.  5.  6.  7.  1.  2.  3.  4.]
         [ 8.  9. 10. 11.  4.  3.  2.  1.]]
        <NDArray 3x8 @cpu(0)>
```

Just like in NumPy, we can construct binary NDArrays by a logical statement. Take `x == y` as an example. If `x` and `y` are equal for some entry, the new NDArray has a value of 1 at the same position; otherwise, it is 0.

```
In [15]: x == y

Out[15]:
        [[0. 1. 0. 1.]
         [0. 0. 0. 0.]
         [0. 0. 0. 0.]]
        <NDArray 3x4 @cpu(0)>
```

Summing all the elements in the NDArray yields an NDArray with only one element.

```
In [16]: x.sum()

Out[16]:
        [66.]
        <NDArray 1 @cpu(0)>
```

We can transform the result into a scalar in Python using the `asscalar` function. In the following example, the ℓ_2 norm of `x` yields a single element NDArray. The final result is transformed into a scalar.

2.1. Data Manipulation

```
In [17]: x.norm().asscalar()
Out[17]: 22.494442
```

For stylistic convenience, we can write `y.exp()`, `x.sum()`, `x.norm()`, etc. also as `nd.exp(y)`, `nd.sum(x)`, `nd.norm(x)`.

2.1.3 Broadcast Mechanism

In the above section, we saw how to perform operations on two NDArrays of the same shape. When their shapes differ, a broadcasting mechanism may be triggered analogous to NumPy: first, copy the elements appropriately so that the two NDArrays have the same shape, and then carry out operations by element.

```
In [18]: a = nd.arange(3).reshape((3, 1))
         b = nd.arange(2).reshape((1, 2))
         a, b
Out[18]: (
          [[0.]
           [1.]
           [2.]]
          <NDArray 3x1 @cpu(0)>,
          [[0. 1.]]
          <NDArray 1x2 @cpu(0)>)
```

Since `a` and `b` are (3x1) and (1x2) matrices respectively, their shapes do not match up if we want to add them. NDArray addresses this by broadcasting' the entries of both matrices into a larger (3x2) matrix as follows: for matrix `a` it replicates the columns, for matrix `b` it replicates the rows before adding up both element-wise.

```
In [19]: a + b
Out[19]:
          [[0. 1.]
           [1. 2.]
           [2. 3.]]
          <NDArray 3x2 @cpu(0)>
```

2.1.4 Indexing and Slicing

Just like in any other Python array, elements in an NDArray can be accessed by its index. In good Python tradition the first element has index 0 and ranges are specified to include the first but not the last. By this logic `1:3` selects the second and third element. Let's try this out by selecting the respective rows in a matrix.

```
In [20]: x[1:3]
Out[20]:
          [[ 4.  5.  6.  7.]
           [ 8.  9. 10. 11.]]
          <NDArray 2x4 @cpu(0)>
```

Beyond reading we can also write elements of a matrix.

```
In [21]: x[1, 2] = 9
         x
Out[21]:
         [[ 0.  1.  2.  3.]
          [ 4.  5.  9.  7.]
          [ 8.  9. 10. 11.]]
         <NDArray 3x4 @cpu(0)>
```

If we want to assign multiple elements the same value, we simply index all of them and then assign them the value. For instance, `[0:2, :]` accesses the first and second rows. While we discussed indexing for matrices, this obviously also works for vectors and for tensors of more than 2 dimensions.

```
In [22]: x[0:2, :] = 12
         x
Out[22]:
         [[12. 12. 12. 12.]
          [12. 12. 12. 12.]
          [ 8.  9. 10. 11.]]
         <NDArray 3x4 @cpu(0)>
```

2.1.5 Saving Memory

In the previous example, every time we ran an operation, we allocated new memory to host its results. For example, if we write `y = x + y`, we will dereference the matrix that `y` used to point to and instead point it at the newly allocated memory. In the following example we demonstrate this with Python's `id()` function, which gives us the exact address of the referenced object in memory. After running `y = y + x`, we'll find that `id(y)` points to a different location. That's because Python first evaluates `y + x`, allocating new memory for the result and then subsequently redirects `y` to point at this new location in memory.

```
In [23]: before = id(y)
         y = y + x
         id(y) == before

Out[23]: False
```

This might be undesirable for two reasons. First, we don't want to run around allocating memory unnecessarily all the time. In machine learning, we might have hundreds of megabytes of parameters and update all of them multiple times per second. Typically, we'll want to perform these updates *in place*. Second, we might point at the same parameters from multiple variables. If we don't update in place, this could cause a memory leak, and could cause us to inadvertently reference stale parameters.

Fortunately, performing in-place operations in MXNet is easy. We can assign the result of an operation to a previously allocated array with slice notation, e.g., `y[:] = <expression>`. To illustrate the behavior, we first clone the shape of a matrix using `zeros_like` to allocate a block of 0 entries.

```
In [24]: z = y.zeros_like()
         print('id(z):', id(z))
```

```
        z[:] = x + y
        print('id(z):', id(z))

id(z): 140261323923752
id(z): 140261323923752
```

While this looks pretty, `x+y` here will still allocate a temporary buffer to store the result of `x+y` before copying it to `y[:]`. To make even better use of memory, we can directly invoke the underlying `ndarray` operation, in this case `elemwise_add`, avoiding temporary buffers. We do this by specifying the `out` keyword argument, which every `ndarray` operator supports:

```
In [25]: before = id(z)
         nd.elemwise_add(x, y, out=z)
         id(z) == before

Out[25]: True
```

If the value of `x` is not reused in subsequent programs, we can also use `x[:]  = x + y` or `x += y` to reduce the memory overhead of the operation.

```
In [26]: before = id(x)
         x += y
         id(x) == before

Out[26]: True
```

2.1.6 Mutual Transformation of NDArray and NumPy

Converting MXNet NDArrays to and from NumPy is easy. The converted arrays do *not* share memory. This minor inconvenience is actually quite important: when you perform operations on the CPU or one of the GPUs, you don't want MXNet having to wait whether NumPy might want to be doing something else with the same chunk of memory. The `array` and `asnumpy` functions do the trick.

```
In [27]: import numpy as np

         a = x.asnumpy()
         print(type(a))
         b = nd.array(a)
         print(type(b))

<class 'numpy.ndarray'>
<class 'mxnet.ndarray.ndarray.NDArray'>
```

Exercises

1. Run the code in this section. Change the conditional statement `x  ==  y` in this section to `x  <  y` or `x  >  y`, and then see what kind of NDArray you can get.

2. Replace the two NDArrays that operate by element in the broadcast mechanism with other shapes, e.g. three dimensional tensors. Is the result the same as expected?

3. Assume that we have three matrices a, b and c. Rewrite `c = nd.dot(a, b.T) + c` in the most memory efficient manner.

Scan the QR Code to Discuss

2.2 Linear Algebra

Now that you can store and manipulate data, let's briefly review the subset of basic linear algebra that you'll need to understand most of the models. We'll introduce all the basic concepts, the corresponding mathematical notation, and their realization in code all in one place. If you're already confident in your basic linear algebra, feel free to skim or skip this chapter.

```
In [1]: from mxnet import nd
```

2.2.1 Scalars

If you never studied linear algebra or machine learning, you're probably used to working with one number at a time. And know how to do basic things like add them together or multiply them. For example, in Palo Alto, the temperature is 52 degrees Fahrenheit. Formally, we call these values $scalars$. If you wanted to convert this value to Celsius (using metric system's more sensible unit of temperature measurement), you'd evaluate the expression $c = (f - 32) * 5/9$ setting f to 52. In this equation, each of the terms 32, 5, and 9 is a scalar value. The placeholders c and f that we use are called variables and they stand in for unknown scalar values.

In mathematical notation, we represent scalars with ordinary lower cased letters (x, y, z). We also denote the space of all scalars as $\mathcal{R}$. For expedience, we're going to punt a bit on what precisely a space is, but for now, remember that if you want to say that x is a scalar, you can simply say $x \in \mathcal{R}$. The symbol $\in$ can be pronounced in and just denotes membership in a set.

In MXNet, we work with scalars by creating NDArrays with just one element. In this snippet, we instantiate two scalars and perform some familiar arithmetic operations with them, such as addition, multiplication, division and exponentiation.

```
In [2]: x = nd.array([3.0])
        y = nd.array([2.0])

        print('x + y = ', x + y)
        print('x * y = ', x * y)
```

```
        print('x / y = ', x / y)
        print('x ** y = ', nd.power(x,y))

x + y =
[5.]
<NDArray 1 @cpu(0)>
x * y =
[6.]
<NDArray 1 @cpu(0)>
x / y =
[1.5]
<NDArray 1 @cpu(0)>
x ** y =
[9.]
<NDArray 1 @cpu(0)>
```

We can convert any NDArray to a Python float by calling its `asscalar` method. Note that this is typically a bad idea. While you are doing this, NDArray has to stop doing anything else in order to hand the result and the process control back to Python. And unfortunately Python isn't very good at doing things in parallel. So avoid sprinkling this operation liberally throughout your code or your networks will take a long time to train.

```
In [3]: x.asscalar()

Out[3]: 3.0
```

2.2.2 Vectors

You can think of a vector as simply a list of numbers, for example `[1.0,3.0,4.0,2.0]`. Each of the numbers in the vector consists of a single scalar value. We call these values the *entries* or *components* of the vector. Often, we're interested in vectors whose values hold some real-world significance. For example, if we're studying the risk that loans default, we might associate each applicant with a vector whose components correspond to their income, length of employment, number of previous defaults, etc. If we were studying the risk of heart attack in hospital patients, we might represent each patient with a vector whose components capture their most recent vital signs, cholesterol levels, minutes of exercise per day, etc. In math notation, we'll usually denote vectors as bold-faced, lower-cased letters ($\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$). In MXNet, we work with vectors via 1D NDArrays with an arbitrary number of components.

```
In [4]: x = nd.arange(4)
        print('x = ', x)

x =
[0. 1. 2. 3.]
<NDArray 4 @cpu(0)>
```

We can refer to any element of a vector by using a subscript. For example, we can refer to the 4th element of $\mathbf{u}$ by u_4. Note that the element u_4 is a scalar, so we don't bold-face the font when referring to it. In code, we access any element i by indexing into the NDArray.

```
In [5]: x[3]
```

```
Out[5]:
      [3.]
      <NDArray 1 @cpu(0)>
```

2.2.3 Length, dimensionality and shape

Let's revisit some concepts from the previous section. A vector is just an array of numbers. And just as every array has a length, so does every vector. In math notation, if we want to say that a vector $\mathbf{x}$ consists of n real-valued scalars, we can express this as $\mathbf{x} \in \mathcal{R}^n$. The length of a vector is commonly called its *dimension*. As with an ordinary Python array, we can access the length of an NDArray by calling Python's in-built `len()` function.

We can also access a vector's length via its `.shape` attribute. The shape is a tuple that lists the dimensionality of the NDArray along each of its axes. Because a vector can only be indexed along one axis, its shape has just one element.

```
In [6]: x.shape
```

```
Out[6]: (4,)
```

Note that the word dimension is overloaded and this tends to confuse people. Some use the *dimensionality* of a vector to refer to its length (the number of components). However some use the word *dimensionality* to refer to the number of axes that an array has. In this sense, a scalar *would have* 0 dimensions and a vector *would have* 1 dimension.

To avoid confusion, when we say 2D array or 3D array, we mean an array with 2 or 3 axes respectively. But if we say :math:'n'-dimensional vector, we mean a vector of length :math:'n'.

```
In [7]: a = 2
        x = nd.array([1,2,3])
        y = nd.array([10,20,30])
        print(a * x)
        print(a * x + y)
```

```
[2. 4. 6.]
<NDArray 3 @cpu(0)>

[12. 24. 36.]
<NDArray 3 @cpu(0)>
```

2.2.4 Matrices

Just as vectors generalize scalars from order 0 to order 1, matrices generalize vectors from $1D$ to $2D$. Matrices, which we'll typically denote with capital letters (A, B, C), are represented in code as arrays with 2 axes. Visually, we can draw a matrix as a table, where each entry a_{ij} belongs to the i-th row and

j-th column.

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix}$$

We can create a matrix with n rows and m columns in MXNet by specifying a shape with two components (n,m) when calling any of our favorite functions for instantiating an `ndarray` such as `ones`, or `zeros`.

```
In [8]: A = nd.arange(20).reshape((5,4))
        print(A)


[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]
 [12. 13. 14. 15.]
 [16. 17. 18. 19.]]
<NDArray 5x4 @cpu(0)>
```

Matrices are useful data structures: they allow us to organize data that has different modalities of variation. For example, rows in our matrix might correspond to different patients, while columns might correspond to different attributes.

We can access the scalar elements a_{ij} of a matrix A by specifying the indices for the row (i) and column (j) respectively. Leaving them blank via a `:` takes all elements along the respective dimension (as seen in the previous section).

We can transpose the matrix through `T`. That is, if $B = A^T$, then $b_{ij} = a_{ji}$ for any i and j.

```
In [9]: print(A.T)


[[ 0.  4.  8. 12. 16.]
 [ 1.  5.  9. 13. 17.]
 [ 2.  6. 10. 14. 18.]
 [ 3.  7. 11. 15. 19.]]
<NDArray 4x5 @cpu(0)>
```

2.2.5 Tensors

Just as vectors generalize scalars, and matrices generalize vectors, we can actually build data structures with even more axes. Tensors give us a generic way of discussing arrays with an arbitrary number of axes. Vectors, for example, are first-order tensors, and matrices are second-order tensors.

Using tensors will become more important when we start working with images, which arrive as 3D data structures, with axes corresponding to the height, width, and the three (RGB) color channels. But in this chapter, we're going to skip past and make sure you know the basics.

```
In [10]: X = nd.arange(24).reshape((2, 3, 4))
         print('X.shape =', X.shape)
         print('X =', X)

X.shape = (2, 3, 4)
X =
[[[ 0.  1.  2.  3.]
  [ 4.  5.  6.  7.]
  [ 8.  9. 10. 11.]]

 [[12. 13. 14. 15.]
  [16. 17. 18. 19.]
  [20. 21. 22. 23.]]]
<NDArray 2x3x4 @cpu(0)>
```

2.2.6 Basic properties of tensor arithmetic

Scalars, vectors, matrices, and tensors of any order have some nice properties that we'll often rely on. For example, as you might have noticed from the definition of an element-wise operation, given operands with the same shape, the result of any element-wise operation is a tensor of that same shape. Another convenient property is that for all tensors, multiplication by a scalar produces a tensor of the same shape. In math, given two tensors X and Y with the same shape, $\alpha X + Y$ has the same shape (numerical mathematicians call this the AXPY operation).

```
In [11]: a = 2
         x = nd.ones(3)
         y = nd.zeros(3)
         print(x.shape)
         print(y.shape)
         print((a * x).shape)
         print((a * x + y).shape)

(3,)
(3,)
(3,)
(3,)
```

Shape is not the the only property preserved under addition and multiplication by a scalar. These operations also preserve membership in a vector space. But we'll postpone this discussion for the second half of this chapter because it's not critical to getting your first models up and running.

2.2.7 Sums and means

The next more sophisticated thing we can do with arbitrary tensors is to calculate the sum of their elements. In mathematical notation, we express sums using the $\sum$ symbol. To express the sum of the elements in a vector $\mathbf{u}$ of length d, we can write $\sum_{i=1}^{d} u_i$. In code, we can just call nd.sum().

```
In [12]: print(x)
         print(nd.sum(x))
```

```
[1. 1. 1.]
<NDArray 3 @cpu(0)>

[3.]
<NDArray 1 @cpu(0)>
```

We can similarly express sums over the elements of tensors of arbitrary shape. For example, the sum of the elements of an $m \times n$ matrix A could be written $\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}$.

```
In [13]: print(A)
         print(nd.sum(A))


[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]
 [12. 13. 14. 15.]
 [16. 17. 18. 19.]]
<NDArray 5x4 @cpu(0)>

[190.]
<NDArray 1 @cpu(0)>
```

A related quantity is the *mean*, which is also called the *average*. We calculate the mean by dividing the sum by the total number of elements. With mathematical notation, we could write the average over a vector $\mathbf{u}$ as $\frac{1}{d} \sum_{i=1}^{d} u_i$ and the average over a matrix A as $\frac{1}{n \cdot m} \sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}$. In code, we could just call nd.mean() on tensors of arbitrary shape:

```
In [14]: print(nd.mean(A))
         print(nd.sum(A) / A.size)


[9.5]
<NDArray 1 @cpu(0)>

[9.5]
<NDArray 1 @cpu(0)>
```

2.2.8 Dot products

So far, we've only performed element-wise operations, sums and averages. And if this was all we could do, linear algebra probably wouldn't deserve its own chapter. However, one of the most fundamental operations is the dot product. Given two vectors $\mathbf{u}$ and $\mathbf{v}$, the dot product $\mathbf{u}^T \mathbf{v}$ is a sum over the products of the corresponding elements: $\mathbf{u}^T \mathbf{v} = \sum_{i=1}^{d} u_i \cdot v_i$.

```
In [15]: x = nd.arange(4)
         y = nd.ones(4)
         print(x, y, nd.dot(x, y))


[0. 1. 2. 3.]
<NDArray 4 @cpu(0)>
[1. 1. 1. 1.]
```

```
<NDArray 4 @cpu(0)>
[6.]
<NDArray 1 @cpu(0)>
```

Note that we can express the dot product of two vectors `nd.dot(x, y)` equivalently by performing an element-wise multiplication and then a sum:

```
In [16]: nd.sum(x * y)

Out[16]:
        [6.]
        <NDArray 1 @cpu(0)>
```

Dot products are useful in a wide range of contexts. For example, given a set of weights $\mathbf{w}$, the weighted sum of some values u could be expressed as the dot product $\mathbf{u}^T\mathbf{w}$. When the weights are non-negative and sum to one $\left(\sum_{i=1}^{d} w_i = 1\right)$, the dot product expresses a *weighted average*. When two vectors each have length one (we'll discuss what *length* means below in the section on norms), dot products can also capture the cosine of the angle between them.

2.2.9 Matrix-vector products

Now that we know how to calculate dot products we can begin to understand matrix-vector products. Let's start off by visualizing a matrix A and a column vector $\mathbf{x}$.

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix}$$

We can visualize the matrix in terms of its row vectors

$$A = \begin{pmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \vdots \\ \mathbf{a}_n^T \end{pmatrix},$$

where each $\mathbf{a}_i^T \in \mathbb{R}^m$ is a row vector representing the i-th row of the matrix A.

Then the matrix vector product $\mathbf{y} = A\mathbf{x}$ is simply a column vector $\mathbf{y} \in \mathbb{R}^n$ where each entry y_i is the dot product $\mathbf{a}_i^T\mathbf{x}$.

$$A\mathbf{x} = \begin{pmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \vdots \\ \mathbf{a}_n^T \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} = \begin{pmatrix} \mathbf{a}_1^T\mathbf{x} \\ \mathbf{a}_2^T\mathbf{x} \\ \vdots \\ \mathbf{a}_n^T\mathbf{x} \end{pmatrix}$$

So you can think of multiplication by a matrix $A \in \mathbb{R}^{n \times m}$ as a transformation that projects vectors from $\mathbb{R}^m$ to $\mathbb{R}^n$.

These transformations turn out to be quite useful. For example, we can represent rotations as multiplications by a square matrix. As we'll see in subsequent chapters, we can also use matrix-vector products to describe the calculations of each layer in a neural network.

Expressing matrix-vector products in code with `ndarray`, we use the same `nd.dot()` function as for dot products. When we call `nd.dot(A, x)` with a matrix `A` and a vector `x`, MXNet knows to perform a matrix-vector product. Note that the column dimension of `A` must be the same as the dimension of `x`.

```
In [17]: nd.dot(A, x)
```

```
Out[17]:
        [ 14.   38.   62.   86.  110.]
        <NDArray 5 @cpu(0)>
```

2.2.10 Matrix-matrix multiplication

If you've gotten the hang of dot products and matrix-vector multiplication, then matrix-matrix multiplications should be pretty straightforward.

Say we have two matrices, $A \in \mathbb{R}^{n \times k}$ and $B \in \mathbb{R}^{k \times m}$:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k1} & b_{k2} & \cdots & b_{km} \end{pmatrix}$$

To produce the matrix product $C = AB$, it's easiest to think of A in terms of its row vectors and B in terms of its column vectors:

$$A = \begin{pmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \vdots \\ \mathbf{a}_n^T \end{pmatrix}, \quad B = \begin{pmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_m \end{pmatrix}.$$

Note here that each row vector $\mathbf{a}_i^T$ lies in $\mathbb{R}^k$ and that each column vector $\mathbf{b}_j$ also lies in $\mathbb{R}^k$.

Then to produce the matrix product $C \in \mathbb{R}^{n \times m}$ we simply compute each entry c_{ij} as the dot product $\mathbf{a}_i^T \mathbf{b}_j$.

$$C = AB = \begin{pmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \vdots \\ \mathbf{a}_n^T \end{pmatrix} \begin{pmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_m \end{pmatrix} = \begin{pmatrix} \mathbf{a}_1^T \mathbf{b}_1 & \mathbf{a}_1^T \mathbf{b}_2 & \cdots & \mathbf{a}_1^T \mathbf{b}_m \\ \mathbf{a}_2^T \mathbf{b}_1 & \mathbf{a}_2^T \mathbf{b}_2 & \cdots & \mathbf{a}_2^T \mathbf{b}_m \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}_n^T \mathbf{b}_1 & \mathbf{a}_n^T \mathbf{b}_2 & \cdots & \mathbf{a}_n^T \mathbf{b}_m \end{pmatrix}$$

You can think of the matrix-matrix multiplication AB as simply performing m matrix-vector products and stitching the results together to form an $n \times m$ matrix. Just as with ordinary dot products and matrix-vector products, we can compute matrix-matrix products in MXNet by using `nd.dot()`.

```
In [18]: B = nd.ones(shape=(4, 3))
         nd.dot(A, B)
Out[18]:
         [[ 6.  6.  6.]
          [22. 22. 22.]
          [38. 38. 38.]
          [54. 54. 54.]
          [70. 70. 70.]]
         <NDArray 5x3 @cpu(0)>
```

2.2.11 Norms

Before we can start implementing models, there's one last concept we're going to introduce. Some of the most useful operators in linear algebra are norms. Informally, they tell us how big a vector or matrix is. We represent norms with the notation $\|\cdot\|$. The $\cdot$ in this expression is just a placeholder. For example, we would represent the norm of a vector $\mathbf{x}$ or matrix A as $\|\mathbf{x}\|$ or $\|A\|$, respectively.

All norms must satisfy a handful of properties:

1. $\|\alpha A\| = |\alpha| \|A\|$

2. $\|A + B\| \leq \|A\| + \|B\|$

3. $\|A\| \geq 0$

4. If $\forall i, j, a_{ij} = 0$, then $\|A\| = 0$

To put it in words, the first rule says that if we scale all the components of a matrix or vector by a constant factor α, its norm also scales by the *absolute value* of the same constant factor. The second rule is the familiar triangle inequality. The third rule simply says that the norm must be non-negative. That makes sense, in most contexts the smallest *size* for anything is 0. The final rule basically says that the smallest norm is achieved by a matrix or vector consisting of all zeros. It's possible to define a norm that gives zero norm to nonzero matrices, but you can't give nonzero norm to zero matrices. That's a mouthful, but if you digest it then you probably have grepped the important concepts here.

If you remember Euclidean distances (think Pythagoras' theorem) from grade school, then non-negativity and the triangle inequality might ring a bell. You might notice that norms sound a lot like measures of distance.

In fact, the Euclidean distance $\sqrt{x_1^2 + \cdots + x_n^2}$ is a norm. Specifically it's the ℓ_2-norm. An analogous computation, performed over the entries of a matrix, e.g. $\sqrt{\sum_{i,j} a_{ij}^2}$, is called the Frobenius norm. More often, in machine learning we work with the squared ℓ_2 norm (notated ℓ_2^2). We also commonly work with the ℓ_1 norm. The ℓ_1 norm is simply the sum of the absolute values. It has the convenient property of placing less emphasis on outliers.

To calculate the ℓ_2 norm, we can just call `nd.norm()`.

```
In [19]: nd.norm(x)
```
```
Out[19]:
        [3.7416573]
        <NDArray 1 @cpu(0)>
```

To calculate the L1-norm we can simply perform the absolute value and then sum over the elements.

```
In [20]: nd.sum(nd.abs(x))
```
```
Out[20]:
        [6.]
        <NDArray 1 @cpu(0)>
```

2.2.12 Norms and objectives

While we don't want to get too far ahead of ourselves, we do want you to anticipate why these concepts are useful. In machine learning we're often trying to solve optimization problems: *Maximize* the probability assigned to observed data. *Minimize* the distance between predictions and the ground-truth observations. Assign vector representations to items (like words, products, or news articles) such that the distance between similar items is minimized, and the distance between dissimilar items is maximized. Oftentimes, these objectives, perhaps the most important component of a machine learning algorithm (besides the data itself), are expressed as norms.

2.2.13 Intermediate linear algebra

If you've made it this far, and understand everything that we've covered, then honestly, you *are* ready to begin modeling. If you're feeling antsy, this is a perfectly reasonable place to move on. You already know nearly all of the linear algebra required to implement a number of many practically useful models and you can always circle back when you want to learn more.

But there's a lot more to linear algebra, even as concerns machine learning. At some point, if you plan to make a career of machine learning, you'll need to know more than we've covered so far. In the rest of this chapter, we introduce some useful, more advanced concepts.

Basic vector properties

Vectors are useful beyond being data structures to carry numbers. In addition to reading and writing values to the components of a vector, and performing some useful mathematical operations, we can analyze vectors in some interesting ways.

One important concept is the notion of a vector space. Here are the conditions that make a vector space:

- **Additive axioms** (we assume that x,y,z are all vectors): $x + y = y + x$ and $(x + y) + z = x + (y + z)$ and $0 + x = x + 0 = x$ and $(-x) + x = x + (-x) = 0$.

- **Multiplicative axioms** (we assume that x is a vector and a, b are scalars): $0 \cdot x = 0$ and $1 \cdot x = x$ and $(ab)x = a(bx)$.

- **Distributive axioms** (we assume that x and y are vectors and a, b are scalars): $a(x + y) = ax + ay$ and $(a + b)x = ax + bx$.

Special matrices

There are a number of special matrices that we will use throughout this tutorial. Let's look at them in a bit of detail:

- **Symmetric Matrix** These are matrices where the entries below and above the diagonal are the same. In other words, we have that $M^\top = M$. An example of such matrices are those that describe pairwise distances, i.e. $M_{ij} = \|x_i - x_j\|$. Likewise, the Facebook friendship graph can be written as a symmetric matrix where $M_{ij} = 1$ if i and j are friends and $M_{ij} = 0$ if they are not. Note that the *Twitter* graph is asymmetric - $M_{ij} = 1$, i.e. i following j does not imply that $M_{ji} = 1$, i.e. j following i.

- **Antisymmetric Matrix** These matrices satisfy $M^\top = -M$. Note that any square matrix can always be decomposed into a symmetric and into an antisymmetric matrix by using $M = \frac{1}{2}(M + M^\top) + \frac{1}{2}(M - M^\top)$.

- **Diagonally Dominant Matrix** These are matrices where the off-diagonal elements are small relative to the main diagonal elements. In particular we have that $M_{ii} \geq \sum_{j \neq i} M_{ij}$ and $M_{ii} \geq \sum_{j \neq i} M_{ji}$. If a matrix has this property, we can often approximate M by its diagonal. This is often expressed as $\mathrm{diag}(M)$.

- **Positive Definite Matrix** These are matrices that have the nice property where $x^\top M x > 0$ whenever $x \neq 0$. Intuitively, they are a generalization of the squared norm of a vector $\|x\|^2 = x^\top x$. It is easy to check that whenever $M = A^\top A$, this holds since there $x^\top M x = x^\top A^\top A x = \|Ax\|^2$. There is a somewhat more profound theorem which states that all positive definite matrices can be written in this form.

Summary

In just a few pages (or one Jupyter notebook) we've taught you all the linear algebra you'll need to understand a good chunk of neural networks. Of course there's a *lot* more to linear algebra. And a lot of that math *is* useful for machine learning. For example, matrices can be decomposed into factors, and these decompositions can reveal low-dimensional structure in real-world datasets. There are entire subfields of machine learning that focus on using matrix decompositions and their generalizations to high-order tensors to discover structure in datasets and solve prediction problems. But this book focuses on deep learning. And we believe you'll be much more inclined to learn more mathematics once you've gotten your hands dirty deploying useful machine learning models on real datasets. So while we reserve the right to introduce more math much later on, we'll wrap up this chapter here.

If you're eager to learn more about linear algebra, here are some of our favorite resources on the topic

- For a solid primer on basics, check out Gilbert Strang's book Introduction to Linear Algebra
- Zico Kolter's Linear Algebra Review and Reference

2.3 Automatic Differentiation

In machine learning, we *train* models to get better and better as a function of experience. Usually, *getting better* means minimizing a *loss function*, i.e. a score that answers how *bad* is our model? With neural networks, we choose loss functions to be differentiable with respect to our parameters. Put simply, this means that for each of the model's parameters, we can determine how much *increasing* or *decreasing* it might affect the loss. While the calculations are straightforward, for complex models, working it out by hand can be a pain (and often error-prone).

The autograd package expedites this work by automatically calculating derivatives. And while most other libraries require that we compile a symbolic graph to take automatic derivatives, `autograd` allows you to take derivatives while writing ordinary imperative code. Every time you make pass through your model, `autograd` builds a graph on the fly, through which it can immediately backpropagate gradients. If you are unfamiliar with some of the math, e.g. gradients, please refer to the *Mathematical Basics* section in the appendix.

```
In [1]: from mxnet import autograd, nd
```

2.3.1 A Simple Example

As a toy example, let's say that we are interested in differentiating the mapping $y = 2\mathbf{x}^\top \mathbf{x}$ with respect to the column vector $\mathbf{x}$. Firstly, we create the variable x and assign an initial value.

```
In [2]: x = nd.arange(4).reshape((4, 1))
        print(x)


[[0.]
 [1.]
 [2.]
 [3.]]
<NDArray 4x1 @cpu(0)>
```

Once we compute the gradient of y with respect to x, we'll need a place to store it. We can tell an NDArray that we plan to store a gradient by invoking its `attach_grad()` method.

```
In [3]: x.attach_grad()
```

Now we're going to compute y and MXNet will generate a computation graph on the fly. It's as if MXNet turned on a recording device and captured the exact path by which each variable was generated.

Note that building the computation graph requires a nontrivial amount of computation. So MXNet will *only* build the graph when explicitly told to do so. This happens by placing code inside a `with autograd.record():` block.

```
In [4]: with autograd.record():
            y = 2 * nd.dot(x.T, x)
        print(y)


[[28.]]
<NDArray 1x1 @cpu(0)>
```

Since the shape of x is (4, 1), y is a scalar. Next, we can automatically find the gradient by calling the `backward` function. It should be noted that if y is not a scalar, MXNet will first sum the elements in y to get the new variable by default, and then find the gradient of the variable with respect to x.

```
In [5]: y.backward()
```

The gradient of the function $y = 2\mathbf{x}^{\top}\mathbf{x}$ with respect to $\mathbf{x}$ should be $4\mathbf{x}$. Now let's verify that the gradient produced is correct.

```
In [6]: print((x.grad - 4 * x).norm().asscalar() == 0)
        print(x.grad)

True

[[ 0.]
 [ 4.]
 [ 8.]
 [12.]]
<NDArray 4x1 @cpu(0)>
```

2.3.2 Training Mode and Prediction Mode

As you can see from the above, after calling the `record` function, MXNet will record and calculate the gradient. In addition, `autograd` will also change the running mode from the prediction mode to the training mode by default. This can be viewed by calling the `is_training` function.

```
In [7]: print(autograd.is_training())
        with autograd.record():
            print(autograd.is_training())

False
True
```

In some cases, the same model behaves differently in the training and prediction modes (such as batch normalization). In other cases, some models may store more auxiliary variables to make computing gradients easier. We will cover these differences in detail in later chapters. For now, you need not worry about these details just yet.

2.3.3 Computing the Gradient of Python Control Flow

One benefit of using automatic differentiation is that even if the computational graph of the function contains Python's control flow (such as conditional and loop control), we may still be able to find the gradient of a variable. Consider the following program: It should be emphasized that the number of iterations of the loop (while loop) and the execution of the conditional judgment (if statement) depend on the value of the input b.

```
In [8]: def f(a):
            b = a * 2
            while b.norm().asscalar() < 1000:
                b = b * 2
            if b.sum().asscalar() > 0:
                c = b
            else:
                c = 100 * b
            return c
```

Note that the number of iterations of the while loop and the execution of the conditional statement (if then else) depend on the value of a. To compute gradients, we need to `record` the calculation, and call the `backward` function to find the gradient.

```
In [9]: a = nd.random.normal(shape=1)
        a.attach_grad()
        with autograd.record():
            d = f(a)
        d.backward()
```

Let's analyze the `f` function defined above. As you can see, it is piecewise linear in its input `a`. In other words, for any `a` there exists some constant such that for a given range `f(a) = g * a`. Consequently `d / a` allows us to verify that the gradient is correct:

```
In [10]: print(a.grad == (d / a))
```

```
[1.]
<NDArray 1 @cpu(0)>
```

2.3.4 Head gradients and the chain rule

Caution: This part is tricky and not necessary to understanding subsequent sections. That said, it is needed if you want to build new layers from scratch. You can skip this on a first read.

Sometimes when we call the backward method, e.g. `y.backward()`, where `y` is a function of `x` we are just interested in the derivative of `y` with respect to `x`. Mathematicians write this as $\frac{dy(x)}{dx}$. At other times, we may be interested in the gradient of `z` with respect to `x`, where `z` is a function of `y`, which in turn, is a function of `x`. That is, we are interested in $\frac{d}{dx}z(y(x))$. Recall that by the chain rule

$$\frac{d}{dx}z(y(x)) = \frac{dz(y)}{dy}\frac{dy(x)}{dx}.$$

So, when `y` is part of a larger function `z`, and we want `x.grad` to store $\frac{dz}{dx}$, we can pass in the *head gradient* $\frac{dz}{dy}$ as an input to `backward()`. The default argument is `nd.ones_like(y)`. See Wikipedia for more details.

```
In [11]: with autograd.record():
             y = x * 2
             z = y * x

         head_gradient = nd.array([10, 1., .1, .01])
         z.backward(head_gradient)
         print(x.grad)


[[0.  ]
 [4.  ]
 [0.8 ]
 [0.12]]
<NDArray 4x1 @cpu(0)>
```

Summary

- MXNet provides an `autograd` package to automate the derivation process.

- MXNet's `autograd` package can be used to derive general imperative programs.

- The running modes of MXNet include the training mode and the prediction mode. We can determine the running mode by `autograd.is_training()`.

Exercises

1. In the example, finding the gradient of the control flow shown in this section, the variable `a` is changed to a random vector or matrix. At this point, the result of the calculation `c` is no longer a scalar. What happens to the result. How do we analyze this?

2. Redesign an example of finding the gradient of the control flow. Run and analyze the result.

3. In a second price auction (such as in eBay or in computational advertising) the winning bidder pays the second highest price. Compute the gradient of the winning bidder with regard to his bid using `autograd`. Why do you get a pathological result? What does this tell us about the mechanism? For more details read the paper by Edelman, Ostrovski and Schwartz, 2005.

4. Why is the second derivative much more expensive to compute than the first derivative?

5. Derive the head gradient relationship for the chain rule. If you get stuck, use the Wikipedia Chain Rule entry.

6. Assume $f(x) = \sin(x)$. Plot $f(x)$ and $\frac{df(x)}{dx}$ on a graph, where you computed the latter without any symbolic calculations, i.e. without exploiting that $f'(x) = \cos(x)$.

2.4 Probability and Statistics

In some form or another, machine learning is all about making predictions. We might want to predict the *probability* of a patient suffering a heart attack in the next year, given their clinical history. In anomaly detection, we might want to assess how *likely* a set of readings from an airplane's jet engine would be, were it operating normally. In reinforcement learning, we want an agent to act intelligently in an environment. This means we need to think about the probability of getting a high reward under each of the available action. And when we build recommender systems we also need to think about probability. For example, if we *hypothetically* worked for a large online bookseller, we might want to estimate the probability that a particular user would buy a particular book, if prompted. For this we need to use the language of probability and statistics. Entire courses, majors, theses, careers, and even departments, are devoted to probability. So our goal here isn't to teach the whole subject. Instead we hope to get you off the ground, to teach you just enough that you know everything necessary to start building your first machine learning models and to have enough of a flavor for the subject that you can begin to explore it on your own if you wish.

We've talked a lot about probabilities so far without articulating what precisely they are or giving a concrete example. Let's get more serious by considering the problem of distinguishing cats and dogs based on photographs. This might sound simpler but it's actually a formidable challenge. To start with, the difficulty of the problem may depend on the resolution of the image.

10px	20px	40px	80px	160px
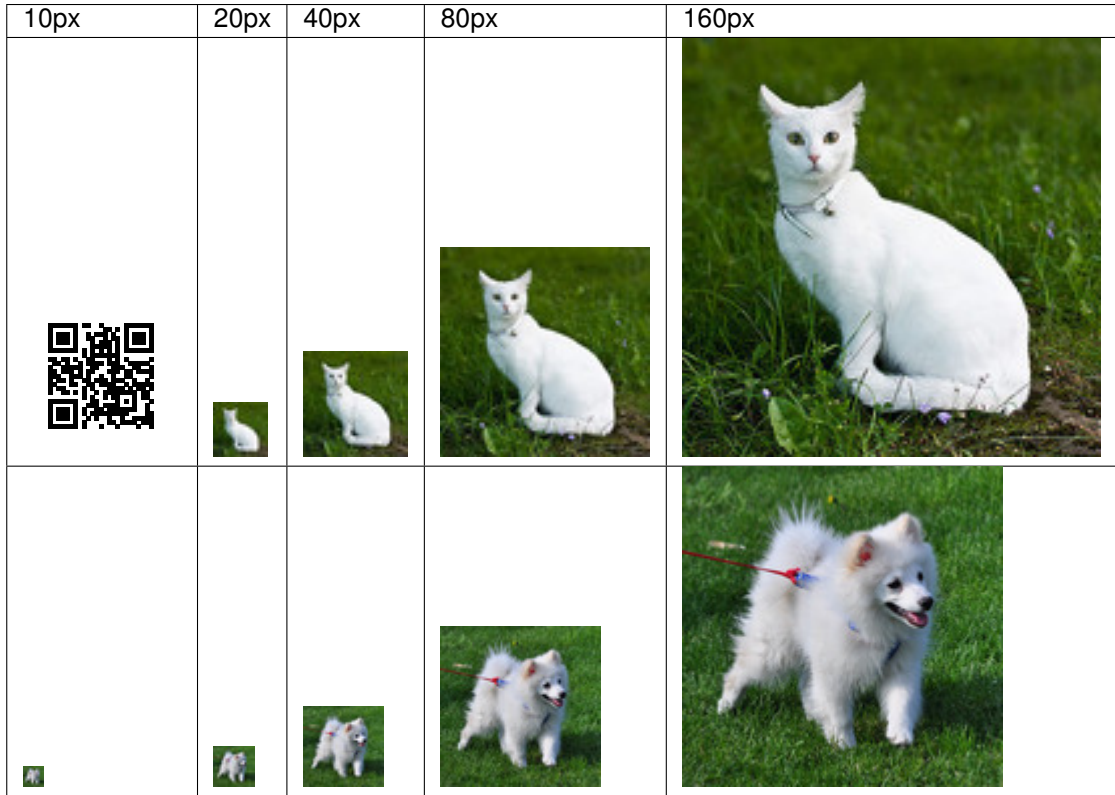
While it's easy for humans to recognize cats and dogs at 320 pixel resolution, it becomes challenging at 40 pixels and next to impossible at 10 pixels. In other words, our ability to tell cats and dogs apart at a large distance (and thus low resolution) might approach uninformed guessing. Probability gives us a formal way of reasoning about our level of certainty. If we are completely sure that the image depicts a cat, we say that the *probability* that the corresponding label l is cat, denoted $P(l = \mathrm{cat})$ equals 1.0. If we had no evidence to suggest that $l = \mathrm{cat}$ or that $l = \mathrm{dog}$, then we might say that the two possibilities were equally *likely* expressing this as $P(l = \mathrm{cat}) = 0.5$. If we were reasonably confident, but not sure that the image depicted a cat, we might assign a probability $.5 < P(l = \mathrm{cat}) < 1.0$.

Now consider a second case: given some weather monitoring data, we want to predict the probability that it will rain in Taipei tomorrow. If it's summertime, the rain might come with probability .5 In both cases, we have some value of interest. And in both cases we are uncertain about the outcome. But there's a key difference between the two cases. In this first case, the image is in fact either a dog or a cat, we just don't know which. In the second case, the outcome may actually be a random event, if you believe in such things (and most physicists do). So probability is a flexible language for reasoning about our level of certainty, and it can be applied effectively in a broad set of contexts.

2.4.1 Basic probability theory

Say that we cast a die and want to know what the chance is of seeing a 1 rather than another digit. If the die is fair, all six outcomes $\mathcal{X} = \{1, \ldots, 6\}$ are equally likely to occur, hence we would see a 1 in 1 out of 6 cases. Formally we state that 1 occurs with probability $\frac{1}{6}$.

For a real die that we receive from a factory, we might not know those proportions and we would need to check whether it is tainted. The only way to investigate the die is by casting it many times and recording the outcomes. For each cast of the die, we'll observe a value $\{1, 2, \ldots, 6\}$. Given these outcomes, we want to investigate the probability of observing each outcome.

One natural approach for each value is to take the individual count for that value and to divide it by the total number of tosses. This gives us an *estimate* of the probability of a given event. The law of large numbers tell us that as the number of tosses grows this estimate will draw closer and closer to the true underlying probability. Before going into the details of what's going here, let's try it out.

To start, let's import the necessary packages:

```
In [1]: import mxnet as mx
        from mxnet import nd
```

Next, we'll want to be able to cast the die. In statistics we call this process of drawing examples from probability distributions *sampling*. The distribution which assigns probabilities to a number of discrete choices is called the *multinomial* distribution. We'll give a more formal definition of *distribution* later, but at a high level, think of it as just an assignment of probabilities to events. In MXNet, we can sample from the multinomial distribution via the aptly named `nd.random.multinomial` function. The function can be called in many ways, but we'll focus on the simplest. To draw a single sample, we simply pass in a vector of probabilities.

```
In [2]: probabilities = nd.ones(6) / 6
        nd.random.multinomial(probabilities)
Out[2]:
        [3]
        <NDArray 1 @cpu(0)>
```

If you run the sampler a bunch of times, you'll find that you get out random values each time. As with estimating the fairness of a die, we often want to generate many samples from the same distribution. It would be really slow to do this with a Python `for` loop, so `random.multinomial` supports drawing multiple samples at once, returning an array of independent samples in any shape we might desire.

```
In [3]: print(nd.random.multinomial(probabilities, shape=(10)))
        print(nd.random.multinomial(probabilities, shape=(5,10)))


[3 4 5 3 5 3 5 2 3 3]
<NDArray 10 @cpu(0)>

[[2 2 1 5 0 5 1 2 2 4]
 [4 3 2 3 2 5 5 0 2 0]
 [3 0 2 4 5 4 0 5 5 5]
 [2 4 4 2 3 4 4 0 4 3]
 [3 0 3 5 4 3 0 2 2 1]]
<NDArray 5x10 @cpu(0)>
```

Now that we know how to sample rolls of a die, we can simulate 1000 rolls. We can then go through and count, after each of the 1000 rolls, how many times each number was rolled.

```
In [4]: rolls = nd.random.multinomial(probabilities, shape=(1000))
        counts = nd.zeros((6,1000))
        totals = nd.zeros(6)
        for i, roll in enumerate(rolls):
            totals[int(roll.asscalar())] += 1
            counts[:, i] = totals
```

To start, we can inspect the final tally at the end of 1000 rolls.

```
In [5]: totals / 1000
```
```
Out[5]:
        [0.167 0.168 0.175 0.159 0.158 0.173]
        <NDArray 6 @cpu(0)>
```

As you can see, the lowest estimated probability for any of the numbers is about .15 and the highest estimated probability is 0.188. Because we generated the data from a fair die, we know that each number actually has probability of $1/6$, roughly .167, so these estimates are pretty good. We can also visualize how these probabilities converge over time towards reasonable estimates.

To start let's take a look at the counts array which has shape (6, 1000). For each time step (out of 1000), counts says how many times each of the numbers has shown up. So we can normalize each j-th column of the counts vector by the number of tosses to give the current estimated probabilities at that time. The counts object looks like this:

```
In [6]: counts
```
```
Out[6]:
        [[ 0.   0.   0. ... 165. 166. 167.]
         [ 1.   1.   1. ... 168. 168. 168.]
         [ 0.   0.   0. ... 175. 175. 175.]
         [ 0.   0.   0. ... 159. 159. 159.]
         [ 0.   1.   2. ... 158. 158. 158.]
         [ 0.   0.   0. ... 173. 173. 173.]]
        <NDArray 6x1000 @cpu(0)>
```

Normalizing by the number of tosses, we get:

```
In [7]: x = nd.arange(1000).reshape((1,1000)) + 1
        estimates = counts / x
        print(estimates[:,0])
        print(estimates[:,1])
        print(estimates[:,100])


[0. 1. 0. 0. 0. 0.]
<NDArray 6 @cpu(0)>

[0.  0.5 0.  0.  0.5 0. ]
<NDArray 6 @cpu(0)>

[0.1980198  0.15841584 0.17821783 0.18811882 0.12871288 0.14851485]
<NDArray 6 @cpu(0)>
```

As you can see, after the first toss of the die, we get the extreme estimate that one of the numbers will be rolled with probability 1.0 and that the others have probability 0. After 100 rolls, things already look a bit more reasonable. We can visualize this convergence by using the plotting package `matplotlib`. If you don't have it installed, now would be a good time to install it.

```python
In [8]: %matplotlib inline
        from matplotlib import pyplot as plt
        from IPython import display
        display.set_matplotlib_formats('svg')

        plt.figure(figsize=(8, 6))
        for i in range(6):
            plt.plot(estimates[i, :].asnumpy(), label=("P(die=" + str(i) +")"))

        plt.axhline(y=0.16666, color='black', linestyle='dashed')
        plt.legend()
        plt.show()
```
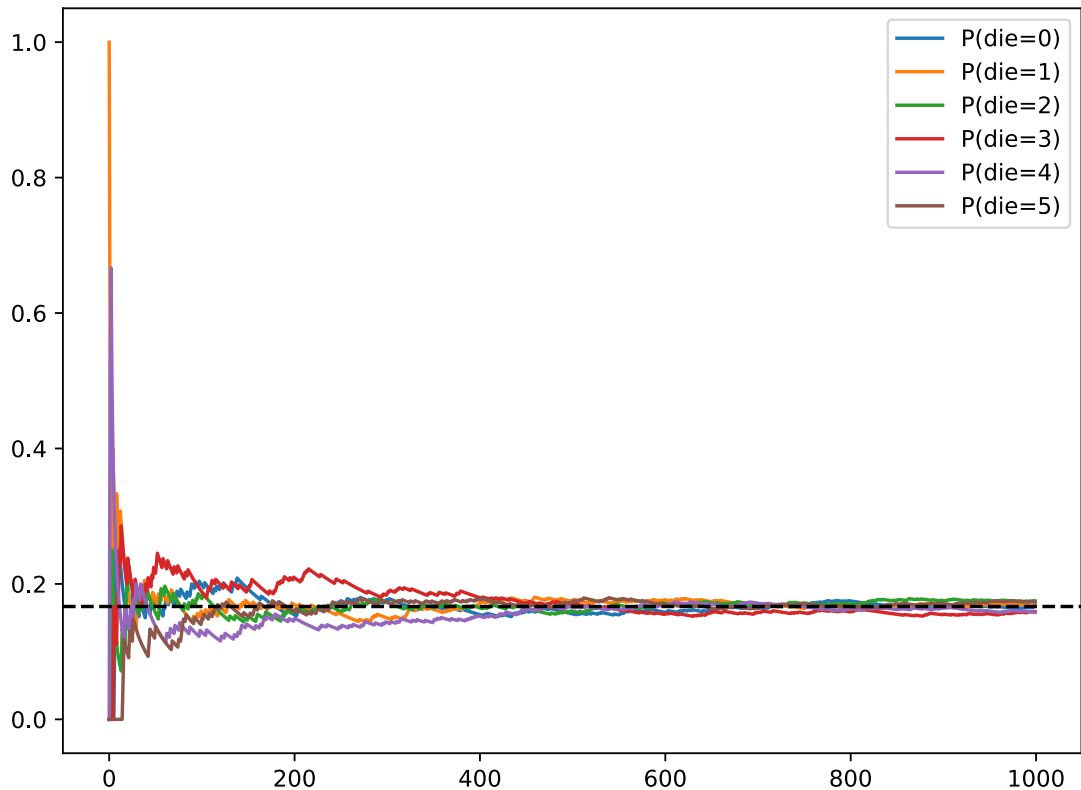

Each solid curve corresponds to one of the six values of the die and gives our estimated probability that the die turns up that value as assessed after each of the 1000 turns. The dashed black line gives the true underlying probability. As we get more data, the solid curves converge towards the true answer.

In our example of casting a die, we introduced the notion of a **random variable**. A random variable, which we denote here as X can be pretty much any quantity and is not deterministic. Random variables could take one value among a set of possibilities. We denote sets with brackets, e.g., $\{\mathrm{cat}, \mathrm{dog}, \mathrm{rabbit}\}$. The items contained in the set are called *elements*, and we can say that an element x is *in* the set S, by writing $x \in S$. The symbol $\in$ is read as in and denotes membership. For instance, we could truthfully say $\mathrm{dog} \in \{\mathrm{cat}, \mathrm{dog}, \mathrm{rabbit}\}$. When dealing with the rolls of die, we are concerned with a variable $X \in \{1, 2, 3, 4, 5, 6\}$.

Note that there is a subtle difference between discrete random variables, like the sides of a dice, and continuous ones, like the weight and the height of a person. There's little point in asking whether two people have exactly the same height. If we take precise enough measurements you'll find that no two people on the planet have the exact same height. In fact, if we take a fine enough measurement, you will not have the same height when you wake up and when you go to sleep. So there's no purpose in asking about the probability that someone is 2.00139278291028719210196740527486202 meters tall. Given the world population of humans the probability is virtually 0. It makes more sense in this case to ask whether someone's height falls into a given interval, say between 1.99 and 2.01 meters. In these cases we quantify the likelihood that we see a value as a *density*. The height of exactly 2.0 meters has no probability, but nonzero density. In the interval between any two different heights we have nonzero probability.

There are a few important axioms of probability that you'll want to remember:

- For any event z, the probability is never negative, i.e. $\Pr(Z = z) \geq 0$.

- For any two events $Z = z$ and $X = x$ the union is no more likely than the sum of the individual events, i.e. $\Pr(Z = z \cup X = x) \leq \Pr(Z = z) + \Pr(X = x)$.

- For any random variable, the probabilities of all the values it can take must sum to 1, i.e. $\sum_{i=1}^{n} \Pr(Z = z_i) = 1$.

- For any two mutually exclusive events $Z = z$ and $X = x$, the probability that either happens is equal to the sum of their individual probabilities, that is $\Pr(Z = z \cup X = x) = \Pr(Z = z) + \Pr(X = x)$.

2.4.2 Dealing with multiple random variables

Very often, we'll want to consider more than one random variable at a time. For instance, we may want to model the relationship between diseases and symptoms. Given a disease and symptom, say flu' and cough', either may or may not occur in a patient with some probability. While we hope that the probability of both would be close to zero, we may want to estimate these probabilities and their relationships to each other so that we may apply our inferences to effect better medical care.

As a more complicated example, images contain millions of pixels, thus millions of random variables. And in many cases images will come with a label, identifying objects in the image. We can also think of the label as a random variable. We can even get crazy and think of all the metadata as random variables such as location, time, aperture, focal length, ISO, focus distance, camera type, etc. All of these are random variables that occur jointly. When we deal with multiple random variables, there are several quantities of interest. The first is called the joint distribution $\Pr(A, B)$. Given any elements a and b, the

joint distribution lets us answer, what is the probability that $A = a$ and $B = b$ simultaneously? It might be clear that for any values a and b, $\Pr(A, B) \leq \Pr(A = a)$.

This has to be the case, since for A and B to happen, A has to happen *and* B also has to happen (and vice versa). Thus A, B cannot be more likely than A or B individually. This brings us to an interesting ratio: $0 \leq \frac{\Pr(A,B)}{\Pr(A)} \leq 1$. We call this a **conditional probability** and denote it by $\Pr(B|A)$, the probability that B happens, provided that A has happened.

Using the definition of conditional probabilities, we can derive one of the most useful and celebrated equations in statistics - Bayes' theorem. It goes as follows: By construction, we have that $\Pr(A, B) = \Pr(B|A)\Pr(A)$. By symmetry, this also holds for $\Pr(A, B) = \Pr(A|B)\Pr(B)$. Solving for one of the conditional variables we get:

$$\Pr(A|B) = \frac{\Pr(B|A)\Pr(A)}{\Pr(B)}$$

This is very useful if we want to infer one thing from another, say cause and effect but we only know the properties in the reverse direction. One important operation that we need, to make this work, is **marginalization**, i.e., the operation of determining $\Pr(A)$ and $\Pr(B)$ from $\Pr(A, B)$. We can see that the probability of seeing A amounts to accounting for all possible choices of B and aggregating the joint probabilities over all of them, i.e.

$$\Pr(A) = \sum_{B'} \Pr(A, B') \text{ and } \Pr(B) = \sum_{A'} \Pr(A', B)$$

A really useful property to check is for **dependence** and **independence**. Independence is when the occurrence of one event does not influence the occurrence of the other. In this case $\Pr(B|A) = \Pr(B)$. Statisticians typically use $A \perp\!\!\!\perp B$ to express this. From Bayes' Theorem it follows immediately that also $\Pr(A|B) = \Pr(A)$. In all other cases we call A and B dependent. For instance, two successive rolls of a die are independent. On the other hand, the position of a light switch and the brightness in the room are not (they are not perfectly deterministic, though, since we could always have a broken lightbulb, power failure, or a broken switch).

Let's put our skills to the test. Assume that a doctor administers an AIDS test to a patient. This test is fairly accurate and it fails only with 1% probability if the patient is healthy by reporting him as diseased. Moreover, it never fails to detect HIV if the patient actually has it. We use D to indicate the diagnosis and H to denote the HIV status. Written as a table the outcome $\Pr(D|H)$ looks as follows:

outcome	HIV positive	HIV negative
Test positive	1	0.01
Test negative	0	0.99

Note that the column sums are all one (but the row sums aren't), since the conditional probability needs to sum up to 1, just like the probability. Let us work out the probability of the patient having AIDS if the test comes back positive. Obviously this is going to depend on how common the disease is, since it affects the number of false alarms. Assume that the population is quite healthy, e.g. $\Pr(\text{HIV positive}) = 0.0015$.

To apply Bayes' Theorem we need to determine

$$\begin{aligned}
\Pr(\text{Test positive}) &= \Pr(D = 1 | H = 0)\Pr(H = 0) + \Pr(D = 1 | H = 1)\Pr(H = 1) \\
&= 0.01 \cdot 0.9985 + 1 \cdot 0.0015 \\
&= 0.011485
\end{aligned}$$

Hence we get

$$\begin{aligned}
\Pr(H = 1 | D = 1) &= \frac{\Pr(D = 1 | H = 1)\Pr(H = 1)}{\Pr(D = 1)} \\
&= \frac{1 \cdot 0.0015}{0.011485} \\
&= 0.131
\end{aligned}$$

In other words, there's only a 13.1% chance that the patient actually has AIDS, despite using a test that is 99% accurate! As we can see, statistics can be quite counterintuitive.

2.4.3 Conditional independence

What should a patient do upon receiving such terrifying news? Likely, he/she would ask the physician to administer another test to get clarity. The second test has different characteristics (it isn't as good as the first one).

outcome	HIV positive	HIV negative
Test positive	0.98	0.03
Test negative	0.02	0.97

Unfortunately, the second test comes back positive, too. Let us work out the requisite probabilities to invoke Bayes' Theorem.

- $\Pr(D_1 = 1 \text{ and } D_2 = 1 | H = 0) = 0.01 \cdot 0.03 = 0.0003$

- $\Pr(D_1 = 1 \text{ and } D_2 = 1 | H = 1) = 1 \cdot 0.98 = 0.98$

- $\Pr(D_1 = 1 \text{ and } D_2 = 1) = 0.0003 \cdot 0.9985 + 0.98 \cdot 0.0015 = 0.00176955$

- $\Pr(H = 1 | D_1 = 1 \text{ and } D_2 = 1) = \frac{0.98 \cdot 0.0015}{0.00176955} = 0.831$

That is, the second test allowed us to gain much higher confidence that not all is well. Despite the second test being considerably less accurate than the first one, it still improved our estimate quite a bit. *Why couldn't we just run the first test a second time?* After all, the first test was more accurate. The reason is that we needed a second test that confirmed *independently* of the first test that things were dire, indeed. In other words, we made the tacit assumption that $\Pr(D_1, D_2 | H) = \Pr(D_1 | H)\Pr(D_2 | H)$. Statisticians call such random variables **conditionally independent**. This is expressed as $D_1 \perp\!\!\!\perp D_2 | H$.

Summary

So far we covered probabilities, independence, conditional independence, and how to use this to draw some basic conclusions. This is already quite powerful. In the next section we will see how this can be used to perform some basic estimation using a Naive Bayes classifier.

Exercises

1. Given two events with probability $\Pr(A)$ and $\Pr(B)$, compute upper and lower bounds on $\Pr(A \cup B)$ and $\Pr(A \cap B)$. Hint - display the situation using a Venn Diagram.

2. Assume that we have a sequence of events, say A, B and C, where B only depends on A and C only on B, can you simplify the joint probability? Hint - this is a Markov Chain.

Scan the QR Code to Discuss

2.5 Naive Bayes Classification

Conditional independence is useful when dealing with data, since it simplifies a lot of equations. A popular (and very simple) algorithm is the Naive Bayes Classifier. Its key assumption is that the attributes are all independent of each other, given the labels. In other words, we have:

$$p(\mathbf{x}|y) = \prod_i p(x_i|y)$$

Using Bayes Theorem this leads to the classifier $p(y|\mathbf{x}) = \frac{\prod_i p(x_i|y)p(y)}{p(\mathbf{x})}$. Unfortunately, this is still intractable, since we don't know $p(x)$. Fortunately, we don't need it, since we know that $\sum_y p(y|\mathbf{x}) = 1$, hence we can always recover the normalization from

$$p(y|\mathbf{x}) \propto \prod_i p(x_i|y)p(y).$$

To illustrate this a bit, consider classifying emails into spam and ham. It's fair to say that the occurrence of the words `Nigeria`, `prince`, `money`, `rich` are all likely indicators that the e-mail might be spam, whereas `theorem`, `network`, `Bayes` or `statistics` are pretty good indicators that there's substance in the message. Thus, we could model the probability of occurrence for each of these words,

given the respective class and then use it to score the likelihood of a text. In fact, for a long time this *is* what many so-called Bayesian spam filters used.

2.5.1 Optical Character Recognition

Since images are much easier to deal with, we will illustrate the workings of a Naive Bayes classifier for distinguishing digits on the MNIST dataset. The problem is that we don't actually know $p(y)$ and $p(x_i|y)$. So we need to *estimate* it given some training data first. This is what is called *training* the model. Estimating $p(y)$ is not too hard. Since we are only dealing with 10 classes, this is pretty easy - simply count the number of occurrences n_y for each of the digits and divide it by the total amount of data n. For instance, if digit 8 occurs $n_8 = 5,800$ times and we have a total of $n = 60,000$ images, the probability estimate is $p(y = 8) = 0.0967$.

Now on to slightly more difficult things - $p(x_i|y)$. Since we picked black and white images, $p(x_i|y)$ denotes the probability that pixel i is switched on for class y. Just like before we can go and count the number of times n_{iy} such that an event occurs and divide it by the total number of occurrences of y, i.e. n_y. But there's something slightly troubling: certain pixels may never be black (e.g. for very well cropped images the corner pixels might always be white). A convenient way for statisticians to deal with this problem is to add pseudo counts to all occurrences. Hence, rather than n_{iy} we use $n_{iy}+1$ and instead of n_y we use $n_y + 1$. This is also called Laplace Smoothing.

```python
In [1]: %matplotlib inline
        from matplotlib import pyplot as plt
        from IPython import display
        display.set_matplotlib_formats('svg')
        import mxnet as mx
        from mxnet import nd
        import numpy as np

        # We go over one observation at a time (speed doesn't matter here)
        def transform(data, label):
            return (nd.floor(data/128)).astype(np.float32), label.astype(np.float32)
        mnist_train = mx.gluon.data.vision.MNIST(train=True, transform=transform)
        mnist_test  = mx.gluon.data.vision.MNIST(train=False, transform=transform)

        # Initialize the counters
        xcount = nd.ones((784,10))
        ycount = nd.ones((10))

        for data, label in mnist_train:
            y = int(label)
            ycount[y] += 1
            xcount[:,y] += data.reshape((784))

        # using broadcast again for division
        py = ycount / ycount.sum()
        px = (xcount / ycount.reshape(1,10))
```

Now that we computed per-pixel counts of occurrence for all pixels, it's time to see how our model behaves. Time to plot it. This is where it is so much more convenient to work with images. Visualizing 28x28x10 probabilities (for each pixel for each class) would typically be an exercise in futility. However,

by plotting them as images we get a quick overview. The astute reader probably noticed by now that these are some mean looking digits

```
In [2]: import matplotlib.pyplot as plt
        fig, figarr = plt.subplots(1, 10, figsize=(10, 10))
        for i in range(10):
            figarr[i].imshow(xcount[:, i].reshape((28, 28)).asnumpy(), cmap='hot')
            figarr[i].axes.get_xaxis().set_visible(False)
            figarr[i].axes.get_yaxis().set_visible(False)

        plt.show()
        print('Class probabilities', py)
```
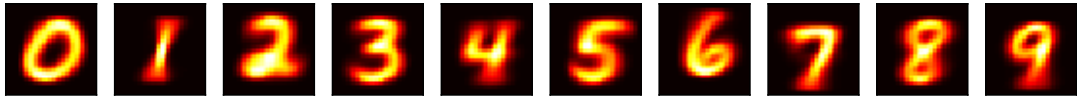


```
Class probabilities
[0.09871688 0.11236461 0.09930012 0.10218297 0.09736711 0.09035161
 0.09863356 0.10441593 0.09751708 0.09915014]
<NDArray 10 @cpu(0)>
```

Now we can compute the likelihoods of an image, given the model. This is statistician speak for $p(x|y)$, i.e. how likely it is to see a particular image under certain conditions (such as the label). Our Naive Bayes model which assumed that all pixels are independent tells us that

$$p(\mathbf{x}|y) = \prod_i p(x_i|y)$$

Using Bayes' rule, we can thus compute $p(y|\mathbf{x})$ via

$$p(y|\mathbf{x}) = \frac{p(\mathbf{x}|y)p(y)}{\sum_{y'} p(\mathbf{x}|y')}$$

Let's try this

```
In [3]: # Get the first test item
        data, label = mnist_test[0]
        data = data.reshape((784,1))

        # Compute the per pixel conditional probabilities
        xprob = (px * data + (1-px) * (1-data))
        # Take the product
        xprob = xprob.prod(0) * py
        print('Unnormalized Probabilities', xprob)
        # Normalize
        xprob = xprob / xprob.sum()
        print('Normalized Probabilities', xprob)
```

```
Unnormalized Probabilities
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
<NDArray 10 @cpu(0)>
Normalized Probabilities
```

```
[nan nan nan nan nan nan nan nan nan nan]
<NDArray 10 @cpu(0)>
```

This went horribly wrong! To find out why, let's look at the per pixel probabilities. They're typically numbers between 0.001 and 1. We are multiplying 784 of them. At this point it is worth mentioning that we are calculating these numbers on a computer, hence with a fixed range for the exponent. What happens is that we experience *numerical underflow*, i.e. multiplying all the small numbers leads to something even smaller until it is rounded down to zero. At that point we get division by zero with `nan` as a result.

To fix this we use the fact that $\log ab = \log a + \log b$, i.e. we switch to summing logarithms. This will get us unnormalized probabilities in log-space. To normalize terms we use the fact that

$$\frac{\exp(a)}{\exp(a) + \exp(b)} = \frac{\exp(a + c)}{\exp(a + c) + \exp(b + c)}$$

In particular, we can pick $c = -\max(a, b)$, which ensures that at least one of the terms in the denominator is 1.

```
In [4]: logpx = nd.log(px)
        logpxneg = nd.log(1-px)
        logpy = nd.log(py)

        def bayespost(data):
            # We need to incorporate the prior probability p(y) since p(y|x) is
            # proportional to p(x|y) p(y)
            logpost = logpy.copy()
            logpost += (logpx * data + logpxneg * (1-data)).sum(0)
            # Normalize to prevent overflow or underflow by subtracting the largest
            # value
            logpost -= nd.max(logpost)
            # Compute the softmax using logpx
            post = nd.exp(logpost).asnumpy()
            post /= np.sum(post)
            return post

        fig, figarr = plt.subplots(2, 10, figsize=(10, 3))

        # Show 10 images
        ctr = 0
        for data, label in mnist_test:
            x = data.reshape((784,1))
            y = int(label)

            post = bayespost(x)

            # Bar chart and image of digit
            figarr[1, ctr].bar(range(10), post)
            figarr[1, ctr].axes.get_yaxis().set_visible(False)
            figarr[0, ctr].imshow(x.reshape((28, 28)).asnumpy(), cmap='hot')
            figarr[0, ctr].axes.get_xaxis().set_visible(False)
            figarr[0, ctr].axes.get_yaxis().set_visible(False)
            ctr += 1

            if ctr == 10:
```

```
    plt.show()
```
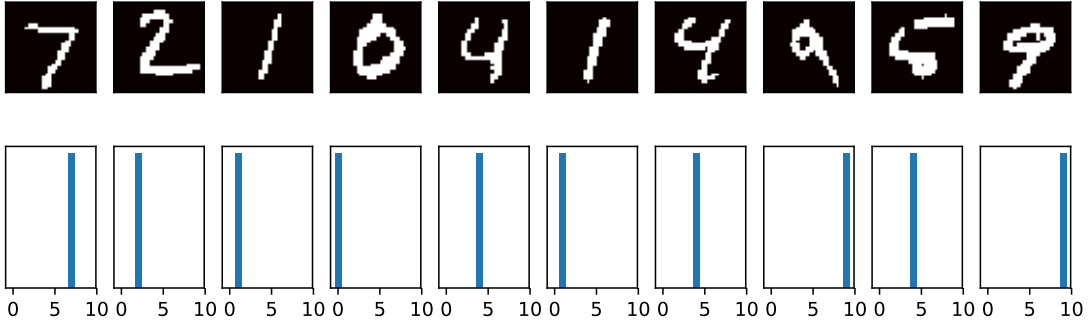

As we can see, this classifier works pretty well in many cases. However, the second last digit shows that it can be both incompetent and overly confident of its incorrect estimates. That is, even if it is horribly wrong, it generates probabilities close to 1 or 0. Not a classifier we should use very much nowadays any longer. To see how well it performs overall, let's compute the overall accuracy of the classifier.

```
In [5]: # Initialize counter
        ctr = 0
        err = 0

        for data, label in mnist_test:
            ctr += 1
            x = data.reshape((784,1))
            y = int(label)

            post = bayespost(x)
            if (post[y] < post.max()):
                err += 1

        print('Naive Bayes has an error rate of', err/ctr)
```

```
Naive Bayes has an error rate of 0.1574
```

Modern deep networks achieve error rates of less than 0.01. While Naive Bayes classifiers used to be popular in the 80s and 90s, e.g. for spam filtering, their heydays are over. The poor performance is due to the incorrect statistical assumptions that we made in our model: we assumed that each and every pixel are *independently* generated, depending only on the label. This is clearly not how humans write digits, and this wrong assumption led to the downfall of our overly naive (Bayes) classifier. Time to start building Deep Networks.

Summary

- Naive Bayes is an easy to use classifier that uses the assumption $p(\mathbf{x}|y) = \prod_i p(x_i|y)$.

- The classifier is easy to train but its estimates can be very wrong.

- To address overly confident and nonsensical estimates, the probabilities $p(x_i|y)$ are smoothed, e.g. by Laplace smoothing. That is, we add a constant to all counts.

- Naive Bayes classifiers don't exploit any correlations between observations.

Exercises

1. Design a Naive Bayes regression estimator where $p(x_i|y)$ is a normal distribution.

2. Under which situations does Naive Bayes work?

3. An eyewitness is sure that he could recognize the perpetrator with 90% accuracy, if he were to encounter him again.

   - Is this a useful statement if there are only 5 suspects?

   - Is it still useful if there are 50?

Scan the QR Code to Discuss

2.6 Sampling

Random numbers are just one form of random variables, and since computers are particularly good with numbers, pretty much everything else in code ultimately gets converted to numbers anyway. One of the basic tools needed to generate random numbers is to sample from a distribution. Let's start with what happens when we use a random number generator (after our usual import ritual).

```
In [1]: %matplotlib inline
        from matplotlib import pyplot as plt
        import mxnet as mx
        from mxnet import nd
        import numpy as np

In [2]: import random
        for i in range(10):
            print(random.random())

0.5816744675658996
0.4709375986527361
0.3746649853551849
0.5994491872942296
```

```
0.23842088668009653
0.34436537020554603
0.20852512785824662
0.0251264612567228
0.8128538352831687
0.5001615637540636
```

2.6.1 Uniform Distribution

These are some pretty random numbers. As we can see, their range is between 0 and 1, and they are evenly distributed. That means there is (actually, should be, since this is not a *real* random number generator) no interval in which numbers are more likely than in any other. In other words, the chances of any of these numbers to fall into the interval, say $[0.2, 0.3)$ are as high as in the interval $[.593264, .693264)$. The way they are generated internally is by producing a random integer first, and then dividing it by its maximum range. If we want to have integers directly, try the following instead. It generates random numbers between 0 and 100.

```
In [3]: for i in range(10):
            print(random.randint(1, 100))

91
21
16
34
29
43
90
31
59
18
```

What if we wanted to check that `randint` is actually really uniform. Intuitively the best strategy would be to run it, say 1 million times, count how many times it generates each one of the values and to ensure that the result is uniform.

```
In [4]: import math

        counts = np.zeros(100)
        fig, axes = plt.subplots(2, 3, figsize=(15, 8), sharex=True)
        axes = axes.reshape(6)
        # Mangle subplots such that we can index them in a linear fashion rather than
        # a 2d grid

        for i in range(1, 1000001):
            counts[random.randint(0, 99)] += 1
            if i in [10, 100, 1000, 10000, 100000, 1000000]:
                axes[int(math.log10(i))-1].bar(np.arange(1, 101), counts)
        plt.show()
```
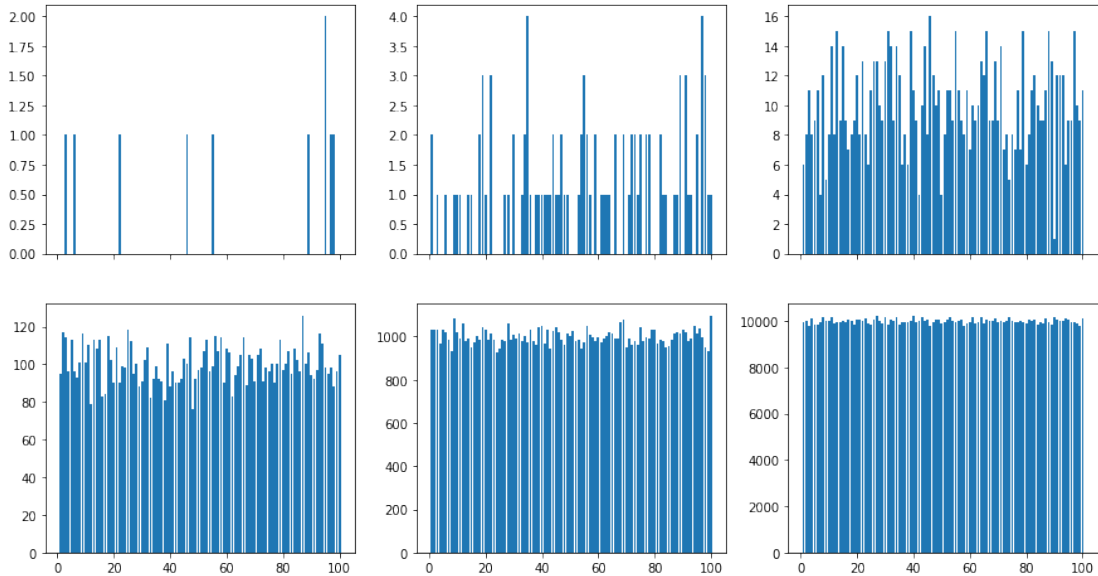
What we can see from the above figures is that the initial number of counts looks *very* uneven. If we sample fewer than 100 draws from a distribution over 100 outcomes this is pretty much expected. But even for 1000 samples there is a significant variability between the draws. What we are really aiming for is a situation where the probability of drawing a number x is given by $p(x)$.

2.6.2 The categorical distribution

Quite obviously, drawing from a uniform distribution over a set of 100 outcomes is quite simple. But what if we have nonuniform probabilities? Let's start with a simple case, a biased coin which comes up heads with probability 0.35 and tails with probability 0.65. A simple way to sample from that is to generate a uniform random variable over $[0, 1]$ and if the number is less than 0.35, we output heads and otherwise we generate tails. Let's try this out.

```
In [5]: # Number of samples
        n = 1000000
        y = np.random.uniform(0, 1, n)
        x = np.arange(1, n+1)
        # Count number of occurrences and divide by the number of total draws
        p0 = np.cumsum(y < 0.35) / x
        p1 = np.cumsum(y >= 0.35) / x

        plt.figure(figsize=(15, 8))
        plt.semilogx(x, p0)
        plt.semilogx(x, p1)
        plt.show()
```
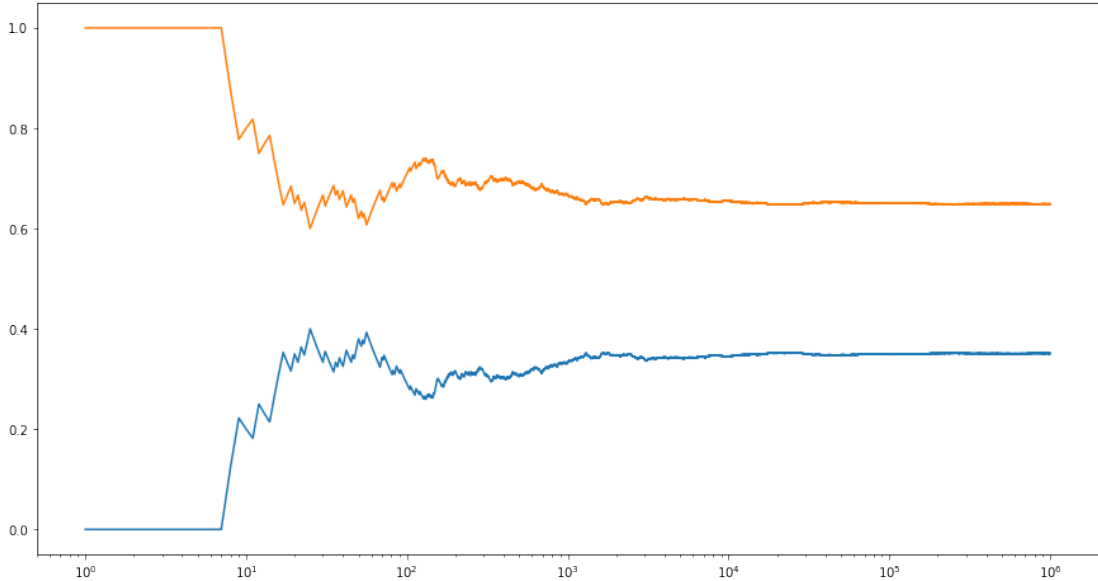
As we can see, on average this sampler will generate 35% zeros and 65% ones. Now what if we have more than two possible outcomes? We can simply generalize this idea as follows. Given any probability distribution, e.g. $p = [0.1, 0.2, 0.05, 0.3, 0.25, 0.1]$ we can compute its cumulative distribution (python's `cumsum` will do this for you) $F = [0.1, 0.3, 0.35, 0.65, 0.9, 1]$. Once we have this we draw a random variable x from the uniform distribution $U[0, 1]$ and then find the interval where $F[i-1] \leq x < F[i]$. We then return i as the sample. By construction, the chances of hitting interval $[F[i-1], F[i])$ has probability $p(i)$.

Note that there are many more efficient algorithms for sampling than the one above. For instance, binary search over F will run in $O(\log n)$ time for n random variables. There are even more clever algorithms, such as the Alias Method to sample in constant time, after $O(n)$ preprocessing.

2.6.3 The Normal distribution

The standard Normal distribution (aka the standard Gaussian distribution) is given by $p(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}x^2\right)$. Let's plot it to get a feel for it.

```
In [6]: x = np.arange(-10, 10, 0.01)
        p = (1/math.sqrt(2 * math.pi)) * np.exp(-0.5 * x**2)
        plt.figure(figsize=(10, 5))
        plt.plot(x, p)
        plt.show()
```
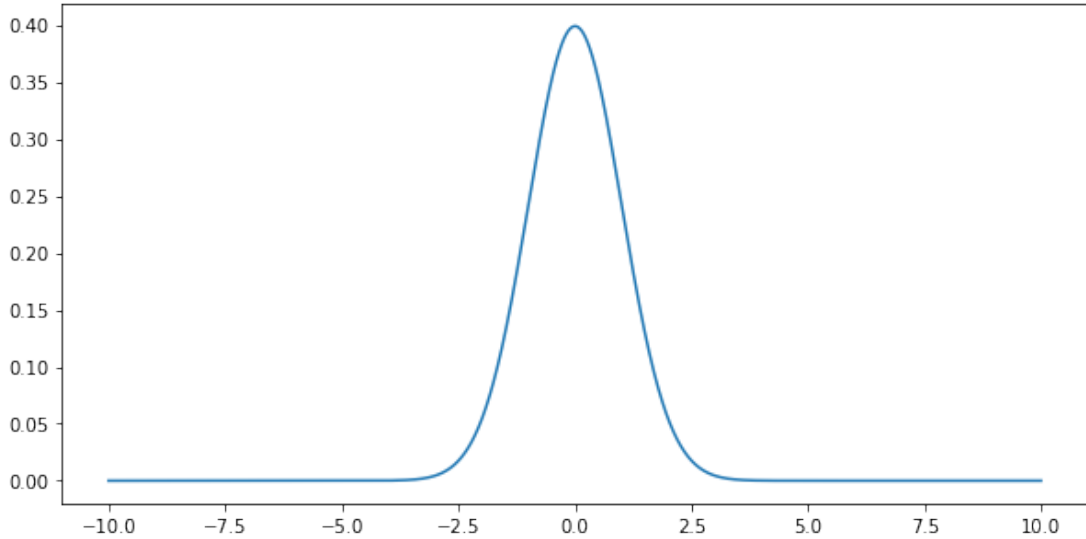
Sampling from this distribution is a lot less trivial. First off, the support is infinite, that is, for any x the density $p(x)$ is positive. Secondly, the density is nonuniform. There are many tricks for sampling from it - the key idea in all algorithms is to stratify $p(x)$ in such a way as to map it to the uniform distribution $U[0, 1]$. One way to do this is with the probability integral transform.

Denote by $F(x) = \int_{-\infty}^{x} p(z)dz$ the cumulative distribution function (CDF) of p. This is in a way the continuous version of the cumulative sum that we used previously. In the same way we can now define the inverse map $F^{-1}(\xi)$, where ξ is drawn uniformly. Unlike previously where we needed to find the correct interval for the vector F (i.e. for the piecewise constant function), we now invert the function $F(x)$.

In practice, this is slightly more tricky since inverting the CDF is hard in the case of a Gaussian. It turns out that the *twodimensional* integral is much easier to deal with, thus yielding two normal random variables than one, albeit at the price of two uniformly distributed ones. For now, suffice it to say that there are built-in algorithms to address this.

The normal distribution has yet another desirable property. In a way all distributions converge to it, if we only average over a sufficiently large number of draws from any other distribution. To understand this in a bit more detail, we need to introduce three important things: expected values, means and variances.

- The expected value $\mathbf{E}_{x \sim p(x)}[f(x)]$ of a function f under a distribution p is given by the integral $\int_x p(x)f(x)dx$. That is, we average over all possible outcomes, as given by p.

- A particularly important expected value is that for the function $f(x) = x$, i.e. $\mu := \mathbf{E}_{x \sim p(x)}[x]$. It provides us with some idea about the typical values of x.

- Another important quantity is the variance, i.e. the typical deviation from the mean $\sigma^2 := \mathbf{E}_{x \sim p(x)}[(x - \mu)^2]$. Simple math shows (check it as an exercise) that $\sigma^2 = \mathbf{E}_{x \sim p(x)}[x^2] - \mathbf{E}_{x \sim p(x)}^2[x]$.

The above allows us to change both mean and variance of random variables. Quite obviously for some random variable x with mean μ, the random variable $x+c$ has mean $\mu+c$. Moreover, γx has the variance $\gamma^2\sigma^2$. Applying this to the normal distribution we see that one with mean μ and variance σ^2 has the form $p(x) = \frac{1}{\sqrt{2\sigma^2\pi}} \exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right)$. Note the scaling factor $\frac{1}{\sigma}$ - it arises from the fact that if we stretch the distribution by σ, we need to lower it by $\frac{1}{\sigma}$ to retain the same probability mass (i.e. the weight under the distribution always needs to integrate out to 1).

Now we are ready to state one of the most fundamental theorems in statistics, the Central Limit Theorem. It states that for sufficiently well-behaved random variables, in particular random variables with well-defined mean and variance, the sum tends toward a normal distribution. To get some idea, let's repeat the experiment described in the beginning, but now using random variables with integer values of $\{0, 1, 2\}$.

```
In [7]: # Generate 10 random sequences of 10,000 uniformly distributed random variables
        tmp = np.random.uniform(size=(10000,10))
        x = 1.0 * (tmp > 0.3) + 1.0 * (tmp > 0.8)
        mean = 1 * 0.5 + 2 * 0.2
        variance = 1 * 0.5 + 4 * 0.2 - mean**2
        print('mean {}, variance {}'.format(mean, variance))

        # Cumulative sum and normalization
        y = np.arange(1,10001).reshape(10000,1)
        z = np.cumsum(x,axis=0) / y

        plt.figure(figsize=(10,5))
        for i in range(10):
            plt.semilogx(y,z[:,i])

        plt.semilogx(y,(variance**0.5) * np.power(y,-0.5) + mean,'r')
        plt.semilogx(y,-(variance**0.5) * np.power(y,-0.5) + mean,'r')
        plt.show()
mean 0.9, variance 0.49
```
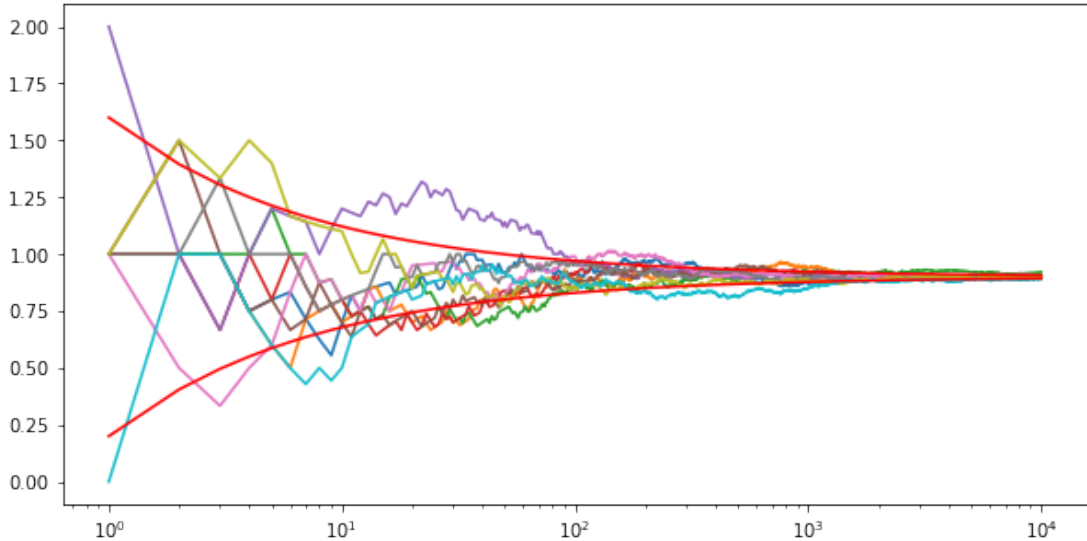
This looks very similar to the initial example, at least in the limit of averages of large numbers of variables. This is confirmed by theory. Denote by mean and variance of a random variable the quantities

$$\mu[p] := \mathbf{E}_{x \sim p(x)}[x] \text{ and } \sigma^2[p] := \mathbf{E}_{x \sim p(x)}[(x - \mu[p])^2]$$

Then we have that $\lim_{n \to \infty} \frac{1}{\sqrt{n}} \sum_{i=1}^{n} \frac{x_i - \mu}{\sigma} \to \mathcal{N}(0, 1)$. In other words, regardless of what we started out with, we will always converge to a Gaussian. This is one of the reasons why Gaussians are so popular in statistics.

2.6.4 More distributions

Many more useful distributions exist. We recommend consulting a statistics book or looking some of them up on Wikipedia for further detail.

- **Binomial Distribution** It is used to describe the distribution over multiple draws from the same distribution, e.g. the number of heads when tossing a biased coin (i.e. a coin with probability $\pi \in [0, 1]$ of returning heads) 10 times. The binomial probability is given by $p(x) = \binom{n}{x} \pi^x (1 - \pi)^{n-x}$.

- **Multinomial Distribution** Obviously we can have more than two outcomes, e.g. when rolling a dice multiple times. In this case the distribution is given by $p(x) = \frac{n!}{\prod_{i=1}^{k} x_i!} \prod_{i=1}^{k} \pi_i^{x_i}$.

- **Poisson Distribution** It is used to model the occurrence of point events that happen with a given rate, e.g. the number of raindrops arriving within a given amount of time in an area (weird fact - the number of Prussian soldiers being killed by horses kicking them followed that distribution). Given a rate λ, the number of occurrences is given by $p(x) = \frac{1}{x!} \lambda^x e^{-\lambda}$.

- **Beta, Dirichlet, Gamma, and Wishart Distributions** They are what statisticians call *conjugate*

to the Binomial, Multinomial, Poisson and Gaussian respectively. Without going into detail, these distributions are often used as priors for coefficients of the latter set of distributions, e.g. a Beta distribution as a prior for modeling the probability for binomial outcomes.

Scan the QR Code to Discuss

2.7 Documentation

Due to the length of this book, it is impossible for us to introduce all MXNet functions and classes. The API documentation and additional tutorials and examples provide plenty of documentation beyond the book.

2.7.1 Finding all the functions and classes in the module

In order to know which functions and classes can be called in a module, we use the `dir` function. For instance we can query all the members or properties in the `nd.random` module.

```
In [1]: from mxnet import nd
        print(dir(nd.random))

['NDArray', '_Null', '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
↪    '__loader__', '__name__', '__package__', '__spec__', '_internal',
↪    '_random_helper', 'current_context', 'exponential', 'exponential_like', 'gamma',
↪    'gamma_like', 'generalized_negative_binomial',
↪    'generalized_negative_binomial_like', 'multinomial', 'negative_binomial',
↪    'negative_binomial_like', 'normal', 'normal_like', 'numeric_types', 'poisson',
↪    'poisson_like', 'randint', 'randn', 'shuffle', 'uniform', 'uniform_like']
```

Generally speaking, we can ignore functions that start and end with __ (special objects in Python) or functions that start with a single _(usually internal functions). According to the remaining member names, we can then hazard a guess that this module offers a generation method for various random numbers, including uniform distribution sampling (`uniform`), normal distribution sampling (`normal`), and Poisson sampling (`poisson`).

2.7.2 Finding the usage of specific functions and classes

For specific function or class usage, we can use the `help` function. Let's take a look at the usage of the `ones_like` function of an NDArray as an example.

```
In [2]: help(nd.ones_like)

Help on function ones_like:

ones_like(data=None, out=None, name=None, **kwargs)
    Return an array of ones with the same shape and type
    as the input array.

    Examples::

      x = [[ 0.,  0.,  0.],
           [ 0.,  0.,  0.]]

      ones_like(x) = [[ 1.,  1.,  1.],
                      [ 1.,  1.,  1.]]


    Parameters
    ----------
    data : NDArray
        The input

    out : NDArray, optional
        The output NDArray to hold the result.

    Returns
    -------
    out : NDArray or list of NDArrays
        The output of this function.
```

From the documentation, we learned that the `ones_like` function creates a new one with the same shape as the NDArray and an element of 1. Let's verify it:

```
In [3]: x = nd.array([[0, 0, 0], [2, 2, 2]])
        y = x.ones_like()
        y
Out[3]:
        [[1. 1. 1.]
         [1. 1. 1.]]
        <NDArray 2x3 @cpu(0)>
```

In the Jupyter notebook, we can use `?` to display the document in another window. For example, `nd.random.uniform?` will create content that is almost identical to `help(nd.random.uniform)`, but will be displayed in an extra window. In addition, if we use two `nd.random.uniform??`, the function implementation code will also be displayed.

2.7.3 API Documentation

For further details on the API details check the MXNet website at http://mxnet.apache.org/. You can find the details under the appropriate headings (also for programming languages other than Python).

Exercise

Look up `ones_like` and `autograd` in the API documentation.

Scan the QR Code to Discuss

3

Linear Neural Networks

This chapter introduces the basics of deep learning. This includes network architectures, data, loss functions, optimization, and capacity control. In order to make things easier to grasp, we begin with very simple concepts, such as linear functions, linear regression, and stochastic gradient descent. This forms the basis for slightly more complex techniques such as softmax regression (introduced here) and multi-layer perceptrons, (introduced in the next chapter).

3.1 Linear Regression

To get our feet wet, we'll start off by looking at the problem of regression. This is the task of predicting a *real valued target* y given a data point x. Regression problems are extremely common in practice. For example, they are used for predicting continuous values, such as house prices, temperatures, sales, and so on. This is quite different from classification problems (which we study later), where the outputs are discrete (such as apple, banana, orange, etc. in image classification).

3.1.1 Basic Elements of Linear Regression

In linear regression, the simplest and still perhaps the most useful approach, we assume that prediction can be expressed as a *linear* combination of the input features (thus giving the name *linear* regression).

Linear Model

For the sake of simplicity we will use the problem of estimating the price of a house (e.g. in dollars) based on area (e.g. in square feet) and age (e.g. in years) as our running example. In this case we could model

$$\text{price} = w_{\text{area}} \cdot \text{area} + w_{\text{age}} \cdot \text{age} + b$$

While this is quite illustrative, it becomes extremely tedious when dealing with more than two variables (even just naming them becomes a pain). This is what mathematicians have invented vectors for. In the case of d variables we get

$$\hat{y} = w_1 \cdot x_1 + ... + w_d \cdot x_d + b$$

Given a collection of data points X, and corresponding target values $\mathbf{y}$, we'll try to find the *weight* vector w and bias term b (also called an *offset* or *intercept*) that approximately associate data points x_i with their corresponding labels y_i. Using slightly more advanced math notation, we can express the long sum as $\hat{y} = \mathbf{w}^\top \mathbf{x} + b$. Finally, for a collection of data points $\mathbf{X}$ the predictions $\hat{y}$ can be expressed via the matrix-vector product:

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b$$

It's quite reasonable to assume that the relationship between x and y is only approximately linear. There might be some error in measuring things. Likewise, while the price of a house typically decreases, this is probably less the case with very old historical mansions which are likely to be prized specifically for their age. To find the parameters w we need two more things: some way to measure the quality of the current model and secondly, some way to manipulate the model to improve its quality.

Training Data

The first thing that we need is data, such as the actual selling price of multiple houses as well as their corresponding area and age. We hope to find model parameters on this data to minimize the error between the predicted price and the real price of the model. In the terminology of machine learning, the data set is called a training data' or training set', a house (often a house and its price) is called a sample', and its actual selling price is called a label'. The two factors used to predict the label are called features' or covariates'. Features are used to describe the characteristics of the sample.

Typically we denote by n the number of samples that we collect. Each sample (indexed as i) is described by $x^{(i)} = [x_1^{(i)}, x_2^{(i)}]$, and the label is $y^{(i)}$.

Loss Function

In model training, we need to measure the error between the predicted value and the real value of the price. Usually, we will choose a non-negative number as the error. The smaller the value, the smaller the

error. A common choice is the square function. The expression for evaluating the error of a sample with an index of i is as follows:

$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2} \left(\hat{y}^{(i)} - y^{(i)} \right)^2,$$

The constant $1/2$ ensures that the constant coefficient, after deriving the quadratic term, is 1, which is slightly simpler in form. Obviously, the smaller the error, the closer the predicted price is to the actual price, and when the two are equal, the error will be zero. Given the training data set, this error is only related to the model parameters, so we record it as a function with the model parameters as parameters. In machine learning, we call the function that measures the error the loss function'. The squared error function used here is also referred to as square loss'.

To make things a bit more concrete, consider the example below where we plot a regression problem for a one-dimensional case, e.g. for a model where house prices depend only on area.
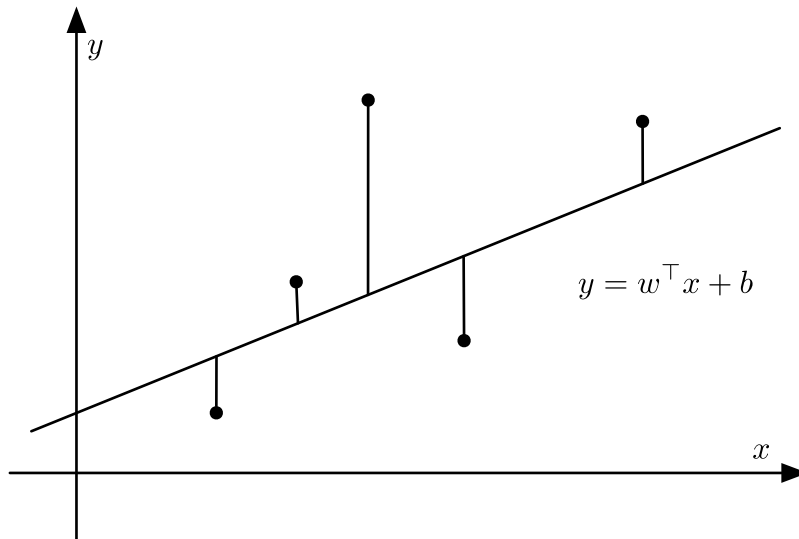
Fig. 3.1: Linear regression is a single-layer neural network.

As you can see, large differences between estimates $\hat{y}^{(i)}$ and observations $y^{(i)}$ lead to even larger contributions in terms of the loss, due to the quadratic dependence. To measure the quality of a model on the entire dataset, we can simply average the losses on the training set.

$$L(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^{n} l^{(i)}(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{2} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right)^2.$$

In model training, we want to find a set of model parameters, represented by $\mathbf{w}^*, b^*$, that can minimize

the average loss of training samples:

$$\mathbf{w}^*, b^* = \operatorname*{argmin}_{\mathbf{w}, b} \ L(\mathbf{w}, b).$$

Optimization Algorithm

When the model and loss function are in a relatively simple format, the solution to the aforementioned loss minimization problem can be expressed analytically in a closed form solution, involving matrix inversion. This is very elegant, it allows for a lot of nice mathematical analysis, *but* it is also very restrictive insofar as this approach only works for a small number of cases (e.g. multilayer perceptrons and nonlinear layers are no go). Most deep learning models do not possess such analytical solutions. The value of the loss function can only be reduced by a finite update of model parameters via an incremental optimization algorithm.

The mini-batch stochastic gradient descent is widely used for deep learning to find numerical solutions. Its algorithm is simple: first, we initialize the values of the model parameters, typically at random; then we iterate over the data multiple times, so that each iteration may reduce the value of the loss function. In each iteration, we first randomly and uniformly sample a mini-batch $\mathcal{B}$ consisting of a fixed number of training data examples; we then compute the derivative (gradient) of the average loss on the mini batch with regard to the model parameters. Finally, the product of this result and a predetermined step size $\eta > 0$ is used to change the parameters in the direction of the minimum of the loss. In math we have (∂ denotes the partial derivative):

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{(\mathbf{w}, b)} l^{(i)}(\mathbf{w}, b)$$

For quadratic losses and linear functions we can write this out explicitly as follows. Note that $\mathbf{w}$ and $\mathbf{x}$ are vectors. Here the more elegant vector notation makes the math much more readable than expressing things in terms of coefficients, say $w_1, w_2, \ldots w_d$.

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{\mathbf{w}} l^{(i)}(\mathbf{w}, b) \quad = w - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right),$$

$$b \leftarrow b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_b l^{(i)}(\mathbf{w}, b) \quad = b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right).$$

In the above equation $|\mathcal{B}|$ represents the number of samples (batch size) in each mini-batch, η is referred to as learning rate' and takes a positive number. It should be emphasized that the values of the batch size and learning rate are set somewhat manually and are typically not learned through model training. Therefore, they are referred to as *hyper-parameters*. What we usually call *tuning hyper-parameters* refers to the adjustment of these terms. In the worst case this is performed through repeated trial and error until the appropriate hyper-parameters are found. A better approach is to learn these as parts of model training. This is an advanced topic and we do not cover them here for the sake of simplicity.

3. Linear Neural Networks

Model Prediction

After model training has been completed, we then record the values of the model parameters $\mathbf{w}, b$ as $\hat{\mathbf{w}}, \hat{b}$. Note that we do not necessarily obtain the optimal solution of the loss function minimizer, $\mathbf{w}^*, b^*$ (or the true parameters), but instead we gain an approximation of the optimal solution. We can then use the learned linear regression model $\hat{\mathbf{w}}^\top x + \hat{b}$ to estimate the price of any house outside the training data set with area (square feet) as x_1 and house age (year) as x_2. Here, estimation also referred to as model prediction' or model inference'.

Note that calling this step inference' is actually quite a misnomer, albeit one that has become the default in deep learning. In statistics inference' means estimating parameters and outcomes based on other data. This misuse of terminology in deep learning can be a source of confusion when talking to statisticians. We adopt the incorrect, but by now common, terminology of using inference' when a (trained) model is applied to new data (and express our sincere apologies to centuries of statisticians).

3.1.2 From Linear Regression to Deep Networks

So far we only talked about linear functions. Neural Networks cover a lot more than that. That said, linear functions are an important building block. Let's start by rewriting things in a layer' notation.

Neural Network Diagram

While in deep learning, we can represent model structures visually using neural network diagrams. To more clearly demonstrate the linear regression as the structure of neural network, Figure 3.1 uses a neural network diagram to represent the linear regression model presented in this section. The neural network diagram hides the weight and bias of the model parameter.
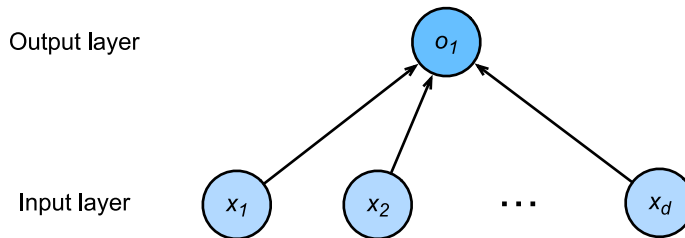
Fig. 3.2: Linear regression is a single-layer neural network.

In the neural network shown above, the inputs are $x_1, x_2, \ldots x_d$. Sometimes the number of inputs is also referred to as feature dimension. In the above cases the number of inputs is d and the number of outputs is 1. It should be noted that we use the output directly as the output of linear regression. Since the input layer does not involve any other nonlinearities or any further calculations, the number of layers is 1. Sometimes this setting is also referred to as a single neuron. Since all inputs are connected to all outputs (in this case it's just one), the layer is also referred to as a fully connected layer' or dense layer'.

A Detour to Biology

Neural networks quite clearly derive their name from Neuroscience. To understand a bit better how many network architectures were invented, it is worth while considering the basic structure of a neuron. For the purpose of the analogy it is sufficient to consider the *dendrites* (input terminals), the *nucleus* (CPU), the *axon* (output wire), and the *axon terminals* (output terminals) which connect to other neurons via *synapses*.
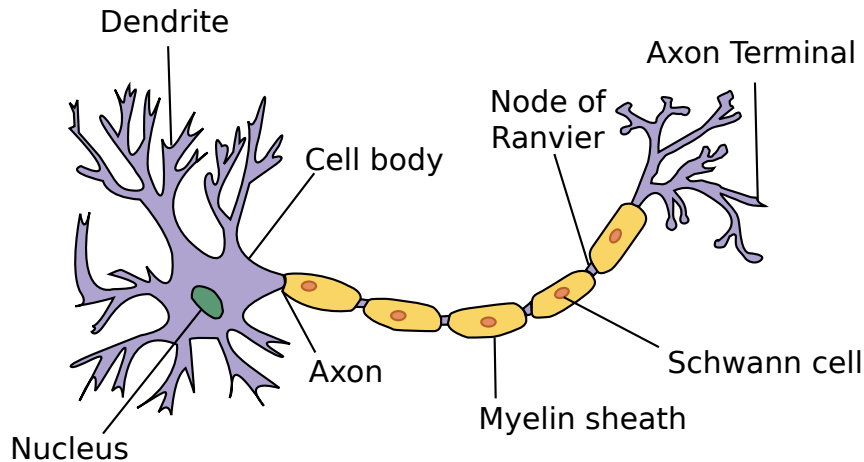
Fig. 3.3: The real neuron

Information x_i arriving from other neurons (or environmental sensors such as the retina) is received in the dendrites. In particular, that information is weighted by *synaptic weights* w_i which determine how to respond to the inputs (e.g. activation or inhibition via $x_i w_i$). All this is aggregated in the nucleus $y = \sum_i x_i w_i + b$, and this information is then sent for further processing in the axon y, typically after some nonlinear processing via $\sigma(y)$. From there it either reaches its destination (e.g. a muscle) or is fed into another neuron via its dendrites.

Brain *structures* can be quite varied. Some look rather arbitrary whereas others have a very regular structure. E.g. the visual system of many insects is quite regular. The analysis of such structures has often inspired neuroscientists to propose new architectures, and in some cases, this has been successful. Note, though, that it would be a fallacy to require a direct correspondence - just like airplanes are *inspired* by birds, they have many distinctions. Equal sources of inspiration were mathematics and computer science.

Vectorization for Speed

In model training or prediction, we often use vector calculations and process multiple observations at the same time. To illustrate why this matters, consider two methods of adding vectors. We begin by creating two 1000 dimensional ones first.

```
In [1]: from mxnet import nd
        from time import time

        a = nd.ones(shape=10000)
        b = nd.ones(shape=10000)
```

One way to add vectors is to add them one coordinate at a time using a for loop.

```
In [2]: start = time()
        c = nd.zeros(shape=10000)
        for i in range(10000):
            c[i] = a[i] + b[i]
        time() - start
```

```
Out[2]: 1.7225244045257568
```

Another way to add vectors is to add the vectors directly:

```
In [3]: start = time()
        d = a + b
        time() - start
```

```
Out[3]: 0.00032639503479003906
```

Obviously, the latter is vastly faster than the former. Vectorizing code is a good way of getting order of magnitude speedups. Likewise, as we saw above, it also greatly simplifies the mathematics and with it, it reduces the potential for errors in the notation.

3.1.3 The Normal Distribution and Squared Loss

The following is optional and can be skipped but it will greatly help with understanding some of the design choices in building deep learning models. As we saw above, using the squared loss $l(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$ has many nice properties, such as having a particularly simple derivative $\partial_{\hat{y}} l(y, \hat{y}) = (\hat{y} - y)$. That is, the gradient is given by the difference between estimate and observation. You might reasonably point out that linear regression is a classical statistical model. Legendre first developed the method of least squares regression in 1805, which was shortly thereafter rediscovered by Gauss in 1809. To understand this a bit better, recall the normal distribution with mean μ and variance σ^2.

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(x - \mu)^2\right)$$

It can be visualized as follows:

```
In [4]: %matplotlib inline
        from matplotlib import pyplot as plt
        from IPython import display
        from mxnet import nd
        import math

        x = nd.arange(-7, 7, 0.01)
        # Mean and variance pairs
        parameters = [(0,1), (0,2), (3,1)]
```

```
        # Display SVG rather than JPG
        display.set_matplotlib_formats('svg')
        plt.figure(figsize=(10, 6))
        for (mu, sigma) in parameters:
            p = (1/math.sqrt(2 * math.pi * sigma**2)) * nd.exp(-(0.5/sigma**2) *
↪   (x-mu)**2)
            plt.plot(x.asnumpy(), p.asnumpy(), label='mean ' + str(mu) + ', variance '
↪   + str(sigma))

        plt.legend()
        plt.show()
```
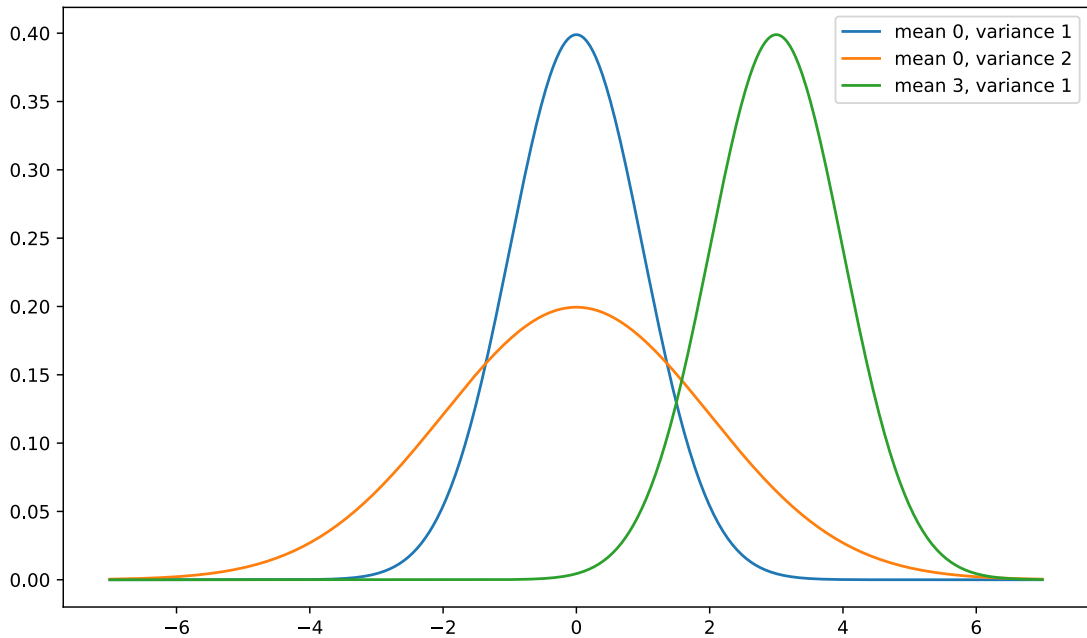

As can be seen in the figure above, changing the mean shifts the function, increasing the variance makes it more spread-out with a lower peak. The key assumption in linear regression with least mean squares loss is that the observations actually arise from noisy observations, where noise is added to the data, e.g. as part of the observations process.

$$y = \mathbf{w}^\top \mathbf{x} + b + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, \sigma^2)$$

This allows us to write out the *likelihood* of seeing a particular y for a given $\mathbf{x}$ via

$$p(y|\mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y - \mathbf{w}^\top \mathbf{x} - b)^2\right)$$

A good way of finding the most likely values of b and $\mathbf{w}$ is to maximize the *likelihood* of the entire dataset

$$p(Y|X) = \prod_{i=1}^{n} p(y^{(i)}|\mathbf{x}^{(i)})$$

The notion of maximizing the likelihood of the data subject to the parameters is well known as the *Maximum Likelihood Principle* and its estimators are usually called *Maximum Likelihood Estimators* (MLE). Unfortunately, maximizing the product of many exponential functions is pretty awkward, both in terms of implementation and in terms of writing it out on paper. Instead, a much better way is to minimize the *Negative Log-Likelihood* $-\log P(Y|X)$. In the above case this works out to be

$$-\log P(Y|X) = \sum_{i=1}^{n} \frac{1}{2} \log(2\pi\sigma^2) + \frac{1}{2\sigma^2} \left(y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b\right)^2$$

A closer inspection reveals that for the purpose of minimizing $-\log P(Y|X)$ we can skip the first term since it doesn't depend on $\mathbf{w}, b$ or even the data. The second term is identical to the objective we initially introduced, but for the multiplicative constant $\frac{1}{\sigma^2}$. Again, this can be skipped if we just want to get the most likely solution. It follows that maximum likelihood in a linear model with additive Gaussian noise is equivalent to linear regression with squared loss.

Summary

- Key ingredients in a machine learning model are training data, a loss function, an optimization algorithm, and quite obviously, the model itself.

- Vectorizing makes everything better (mostly math) and faster (mostly code).

- Minimizing an objective function and performing maximum likelihood can mean the same thing.

- Linear models are neural networks, too.

Exercises

1. Assume that we have some data $x_1, \ldots x_n \in \mathbb{R}$. Our goal is to find a constant b such that $\sum_i (x_i - b)^2$ is minimized.

   - Find the optimal closed form solution.
   - What does this mean in terms of the Normal distribution?

2. Assume that we want to solve the optimization problem for linear regression with quadratic loss explicitly in closed form. To keep things simple, you can omit the bias b from the problem.

   - Rewrite the problem in matrix and vector notation (hint - treat all the data as a single matrix).
   - Compute the gradient of the optimization problem with respect to w.

- Find the closed form solution by solving a matrix equation.

- When might this be better than using stochastic gradient descent (i.e. the incremental optimization approach that we discussed above)? When will this break (hint - what happens for high-dimensional x, what if many observations are very similar)?.

3. Assume that the noise model governing the additive noise ϵ is the exponential distribution. That is, $p(\epsilon) = \frac{1}{2}\exp(-|\epsilon|)$.

   - Write out the negative log-likelihood of the data under the model $-\log p(Y|X)$.

   - Can you find a closed form solution?

   - Suggest a stochastic gradient descent algorithm to solve this problem. What could possibly go wrong (hint - what happens near the stationary point as we keep on updating the parameters). Can you fix this?

4. Compare the runtime of the two methods of adding two vectors using other packages (such as NumPy) or other programming languages (such as MATLAB).

Scan the QR Code to Discuss

3.2 Linear Regression Implementation from Scratch

After getting some background on linear regression, we are now ready for a hands-on implementation. While a powerful deep learning framework minimizes repetitive work, relying on it too much to make things easy can make it hard to properly understand how deep learning works. This matters in particular if we want to change things later, e.g. define our own layers, loss functions, etc. Because of this, we start by describing how to implement linear regression training using only NDArray and `autograd`.

Before we begin, let's import the package or module required for this section's experiment; `matplotlib` will be used for plotting and will be set to embed in the GUI.

```
In [1]: %matplotlib inline
        from IPython import display
        from matplotlib import pyplot as plt
        from mxnet import autograd, nd
        import random
```

3.2.1 Generating Data Sets

By constructing a simple artificial training data set, we can visually compare the differences between the parameters we have learned and the actual model parameters. Set the number of examples in the training data set as 1000 and the number of inputs (feature number) as 2. Using the randomly generated batch example feature $\mathbf{X} \in \mathbb{R}^{1000 \times 2}$, we use the actual weight $\mathbf{w} = [2, -3.4]^\top$ and bias $b = 4.2$ of the linear regression model, as well as a random noise item ϵ to generate the tag

$$\mathbf{y} = \mathbf{X}\mathbf{w} + b + \epsilon$$

The noise term ϵ (or rather each coordinate of it) obeys a normal distribution with a mean of 0 and a standard deviation of 0.01. To get a better idea, let us generate the dataset.

```
In [2]: num_inputs = 2
        num_examples = 1000
        true_w = nd.array([2, -3.4])
        true_b = 4.2
        features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
        labels = nd.dot(features, true_w) + true_b
        labels += nd.random.normal(scale=0.01, shape=labels.shape)
```

Note that each row in `features` consists of a 2-dimensional data point and that each row in `labels` consists of a 1-dimensional target value (a scalar).

```
In [3]: features[0], labels[0]
```

```
Out[3]: (
        [2.2122064 0.7740038]
        <NDArray 2 @cpu(0)>,
        [6.000587]
        <NDArray 1 @cpu(0)>)
```

By generating a scatter plot using the second `features[:,  1]` and `labels`, we can clearly observe the linear correlation between the two.

```
In [4]: def use_svg_display():
            # Display in vector graphics
            display.set_matplotlib_formats('svg')

        def set_figsize(figsize=(3.5, 2.5)):
            use_svg_display()
            # Set the size of the graph to be plotted
            plt.rcParams['figure.figsize'] = figsize

        set_figsize()
        plt.figure(figsize=(10, 6))
        plt.scatter(features[:, 1].asnumpy(), labels.asnumpy(), 1);
```
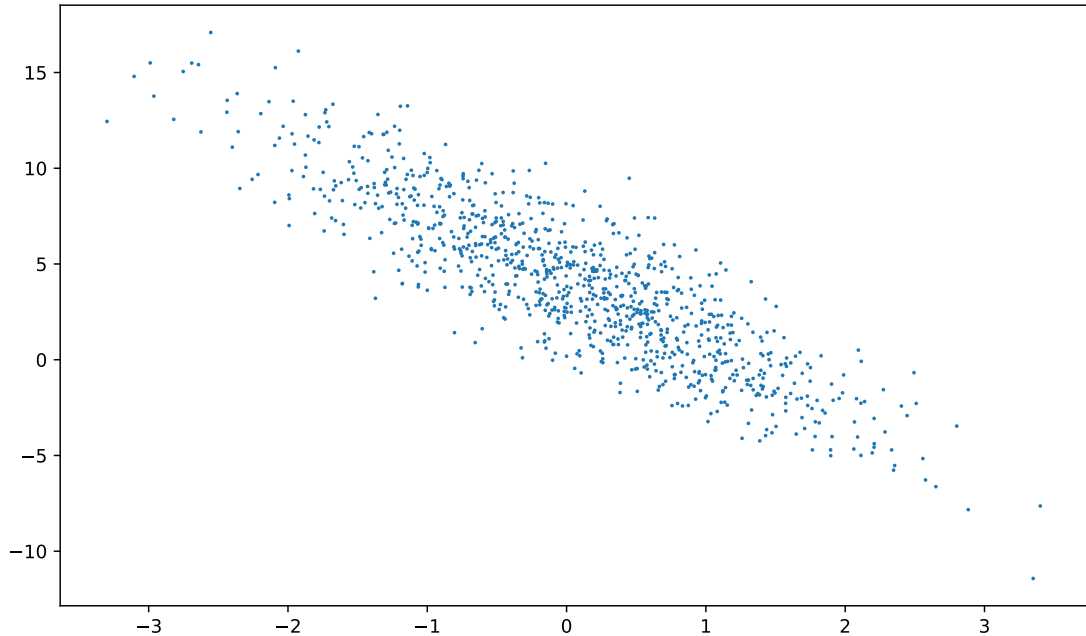
The plotting function `plt` as well as the `use_svg_display` and `set_figsize` functions are defined in the `d2l` package. We will call `d2l.plt` directly for future plotting. To print the vector diagram and set its size, we only need to call `d2l.set_figsize()` before plotting, because `plt` is a global variable in the `d2l` package.

3.2.2 Reading Data

We need to iterate over the entire data set and continuously examine mini-batches of data examples when training the model. Here we define a function. Its purpose is to return the features and tags of random `batch_size` (batch size) examples every time it's called. One might wonder why we are not reading one observation at a time but rather construct an iterator which returns a few observations at a time. This has mostly to do with efficiency when optimizing. Recall that when we processed one dimension at a time the algorithm was quite slow. The same thing happens when processing single observations vs. an entire batch' of them, which can be represented as a matrix rather than just a vector. In particular, GPUs are much faster when it comes to dealing with matrices, up to an order of magnitude. This is one of the reasons why deep learning usually operates on mini-batches rather than singletons.

```
In [5]: # This function has been saved in the d2l package for future use
        def data_iter(batch_size, features, labels):
            num_examples = len(features)
            indices = list(range(num_examples))
            # The examples are read at random, in no particular order
            random.shuffle(indices)
            for i in range(0, num_examples, batch_size):
```

```
        j = nd.array(indices[i: min(i + batch_size, num_examples)])
        yield features.take(j), labels.take(j)
        # The take function will then return the corresponding element based
        # on the indices
```

Let's read and print the first small batch of data examples. The shape of the features in each batch corresponds to the batch size and the number of input dimensions. Likewise, we obtain as many labels as requested by the batch size.

```
In [6]: batch_size = 10

        for X, y in data_iter(batch_size, features, labels):
            print(X, y)
            break


[[-0.08512726 -0.90995044]
 [-0.51129895  0.46543437]
 [-1.2217911  -0.06566298]
 [-0.3990028  -0.57255214]
 [ 0.08145867 -0.68939894]
 [-0.17634277  0.35655284]
 [ 0.50540894 -0.07698673]
 [-0.27530846  1.1180499 ]
 [ 0.15603082  0.5625533 ]
 [ 0.3679345   1.4534262 ]]
<NDArray 10x2 @cpu(0)>
[ 7.1249194e+00  1.6139998e+00  1.9653395e+00  5.3430595e+00
  6.7073822e+00  2.6176562e+00  5.4833393e+00 -1.7545006e-01
  2.6085372e+00  4.2967927e-03]
<NDArray 10 @cpu(0)>
```

Clearly, if we run the iterator again, we obtain a different minibatch until all the data has been exhausted (try this). Note that the iterator described above is a bit inefficient (it requires that we load all data in memory and that we perform a lot of random memory access). The built-in iterators are more efficient and they can deal with data stored on file (or being fed via a data stream).

3.2.3 Initialize Model Parameters

Weights are initialized to normal random numbers using a mean of 0 and a standard deviation of 0.01, with the bias b set to zero.

```
In [7]: w = nd.random.normal(scale=0.01, shape=(num_inputs, 1))
        b = nd.zeros(shape=(1,))
```

In the succeeding cells, we're going to update these parameters to better fit our data. This will involve taking the gradient (a multi-dimensional derivative) of some loss function with respect to the parameters. We'll update each parameter in the direction that reduces the loss. In order for autograd to know that it needs to set up the appropriate data structures, track changes, etc., we need to attach gradients explicitly.

```
In [8]: w.attach_grad()
        b.attach_grad()
```

3.2.4 Define the Model

Next we'll want to define our model. In this case, we'll be working with linear models, the simplest possible useful neural network. To calculate the output of the linear model, we simply multiply a given input with the model's weights w, and add the offset b.

```
In [9]: # This function has been saved in the d2l package for future use
        def linreg(X, w, b):
            return nd.dot(X, w) + b
```

3.2.5 Define the Loss Function

We will use the squared loss function described in the previous section to define the linear regression loss. In the implementation, we need to transform the true value `y` into the predicted value's shape `y_hat`. The result returned by the following function will also be the same as the `y_hat` shape.

```
In [10]: # This function has been saved in the d2l package for future use
         def squared_loss(y_hat, y):
             return (y_hat - y.reshape(y_hat.shape)) ** 2 / 2
```

3.2.6 Define the Optimization Algorithm

Linear regression actually has a closed-form solution. However, most interesting models that we'll care about cannot be solved analytically. So we'll solve this problem by stochastic gradient descent `sgd`. At each step, we'll estimate the gradient of the loss with respect to our weights, using one batch randomly drawn from our dataset. Then, we'll update our parameters a small amount in the direction that reduces the loss. Here, the gradient calculated by the automatic differentiation module is the gradient sum of a batch of examples. We divide it by the batch size to obtain the average. The size of the step is determined by the learning rate `lr`.

```
In [11]: # This function has been saved in the d2l package for future use
         def sgd(params, lr, batch_size):
             for param in params:
                 param[:] = param - lr * param.grad / batch_size
```

3.2.7 Training

In training, we will iterate over the data to improve the model parameters. In each iteration, the mini-batch stochastic gradient is calculated by first calling the inverse function `backward` depending on the currently read mini-batch data examples (feature `X` and label `y`), and then calling the optimization algorithm `sgd` to iterate the model parameters. Since we previously set the batch size `batch_size` to 10, the loss shape `l` for each small batch is (10, 1).

- Initialize parameters $(\mathbf{w}, b)$

- Repeat until done

- Compute gradient $\mathbf{g} \leftarrow \partial_{(\mathbf{w},b)} \frac{1}{\mathcal{B}} \sum_{i \in \mathcal{B}} l(\mathbf{x}^i, y^i, \mathbf{w}, b)$

- Update parameters $(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta \mathbf{g}$

Since nobody wants to compute gradients explicitly (this is tedious and error prone) we use automatic differentiation to compute g. See section Automatic Gradient for more details. Since the loss `l` is not a scalar variable, running `l.backward()` will add together the elements in `l` to obtain the new variable, and then calculate the variable model parameters' gradient.

In an epoch (a pass through the data), we will iterate through the `data_iter` function once and use it for all the examples in the training data set (assuming the number of examples is divisible by the batch size). The number of epochs `num_epochs` and the learning rate `lr` are both hyper-parameters and are set to 3 and 0.03, respectively. Unfortunately in practice, the majority of the hyper-parameters will require some adjustment by trial and error. For instance, the model might actually become more accurate by training longer (but this increases computational cost). Likewise, we might want to change the learning rate on the fly. We will discuss this later in the chapter on *Optimization Algorithms*.

```
In [12]: lr = 0.03  # Learning rate
         num_epochs = 3  # Number of iterations
         net = linreg  # Our fancy linear model
         loss = squared_loss  # 0.5 (y-y')^2

         for epoch in range(num_epochs):
             # Assuming the number of examples can be divided by the batch size, all
             # the examples in the training data set are used once in one epoch
             # iteration. The features and tags of mini-batch examples are given by X
             # and y respectively
             for X, y in data_iter(batch_size, features, labels):
                 with autograd.record():
                     l = loss(net(X, w, b), y)  # Minibatch loss in X and y
                 l.backward()  # Compute gradient on l with respect to [w,b]
                 sgd([w, b], lr, batch_size)  # Update parameters using their gradient
             train_l = loss(net(features, w, b), labels)
             print('epoch %d, loss %f' % (epoch + 1, train_l.mean().asnumpy()))

epoch 1, loss 0.040473
epoch 2, loss 0.000153
epoch 3, loss 0.000051
```

To evaluate the trained model, we can compare the actual parameters used with the parameters we have learned after the training has been completed. They are very close to each other.

```
In [13]: print('Error in estimating w', true_w - w.reshape(true_w.shape))
         print('Error in estimating b', true_b - b)

Error in estimating w
[ 3.2007694e-04 -4.5299530e-06]
<NDArray 2 @cpu(0)>
Error in estimating b
[0.00112724]
<NDArray 1 @cpu(0)>
```

Note that we should not take it for granted that we are able to reover the parameters accurately. This only happens for a special category problems: strongly convex optimization problems with enough' data to ensure that the noisy samples allow us to recover the underlying dependency correctly. In most cases

---

this is *not* the case. In fact, the parameters of a deep network are rarely the same (or even close) between two different runs, unless everything is kept identically, including the order in which the data is traversed. Nonetheless this can lead to very good solutions, mostly due to the fact that quite often there are many sets of parameters that work well.

Summary

We saw how a deep network can be implemented and optimized from scratch, using just NDArray and `autograd` without any need for defining layers, fancy optimizers, etc. This only scratches the surface of what is possible. In the following sections, we will describe additional deep learning models based on what we have just learned and you will learn how to implement them using more concisely.

Exercises

1. What would happen if we were to initialize the weights $\mathbf{w} = 0$. Would the algorithm still work?

2. Assume that you're Georg Simon Ohm trying to come up with a model between voltage and current. Can you use `autograd` to learn the parameters of your model.

3. Can you use Planck's Law to determine the temperature of an object using spectral energy density.

4. What are the problems you might encounter if you wanted to extend `autograd` to second derivatives? How would you fix them?

5. Why is the `reshape` function needed in the `squared_loss` function?

6. Experiment using different learning rates to find out how fast the loss function value drops.

7. If the number of examples cannot be divided by the batch size, what happens to the `data_iter` function's behavior?

Scan the QR Code to Discuss

3.3 Concise Implementation of Linear Regression

With the development of deep learning frameworks, it has become increasingly easy to develop deep learning applications. In practice, we can usually implement the same model, but much more concisely

than how we introduce it in the previous section. In this section, we will introduce how to use the Gluon interface provided by MXNet.

3.3.1 Generating Data Sets

We will generate the same data set as that used in the previous section.

```
In [1]: from mxnet import autograd, nd

        num_inputs = 2
        num_examples = 1000
        true_w = nd.array([2, -3.4])
        true_b = 4.2
        features = nd.random.normal(scale=1, shape=(num_examples, num_inputs))
        labels = nd.dot(features, true_w) + true_b
        labels += nd.random.normal(scale=0.01, shape=labels.shape)
```

3.3.2 Reading Data

Gluon provides the `data` module to read data. Since `data` is often used as a variable name, we will replace it with the pseudonym `gdata` (adding the first letter of Gluon) when referring to the imported `data` module. In each iteration, we will randomly read a mini-batch containing 10 data instances.

```
In [2]: from mxnet.gluon import data as gdata

        batch_size = 10
        # Combine the features and labels of the training data
        dataset = gdata.ArrayDataset(features, labels)
        # Randomly reading mini-batches
        data_iter = gdata.DataLoader(dataset, batch_size, shuffle=True)
```

The use of `data_iter` here is the same as in the previous section. Now, we can read and print the first mini-batch of instances.

```
In [3]: for X, y in data_iter:
            print(X, y)
            break


[[-0.05439435  0.58971226]
 [ 0.34202248  0.46252233]
 [-0.61270916 -2.8199658 ]
 [-0.48212412  0.13752407]
 [-1.0677007   0.372026  ]
 [ 0.8258905   1.0249989 ]
 [-0.11902292  1.1829549 ]
 [-0.3139806   0.27631834]
 [-1.7677424  -0.12628348]
 [ 0.24951884  0.52448374]]
<NDArray 10x2 @cpu(0)>
[ 2.0815115   3.3229208  12.554433    2.7621648   0.7931337   2.3662217
 -0.04490374  2.6284976   1.0976561   2.9193122 ]
<NDArray 10 @cpu(0)>
```

3.3.3 Define the Model

When we implemented the linear regression model from scratch in the previous section, we needed to define the model parameters and use them to describe step by step how the model is evaluated. This can become complicated as we build complex models. Gluon provides a large number of predefined layers, which allow us to focus especially on the layers used to construct the model rather than having to focus on the implementation.

To define a linear model, first import the module `nn`. `nn` is an abbreviation for neural networks. As the name implies, this module defines a large number of neural network layers. We will first define a model variable `net`, which is a `Sequential` instance. In Gluon, a `Sequential` instance can be regarded as a container that concatenates the various layers in sequence. When constructing the model, we will add the layers in their order of occurrence in the container. When input data is given, each layer in the container will be calculated in order, and the output of one layer will be the input of the next layer.

```
In [4]: from mxnet.gluon import nn
        net = nn.Sequential()
```

Recall the architecture of a single layer network. The layer is fully connected since it connects all inputs with all outputs by means of a matrix-vector multiplication. In Gluon, the fully connected layer is referred to as a `Dense` instance. Since we only want to generate a single scalar output, we set that number to 1.
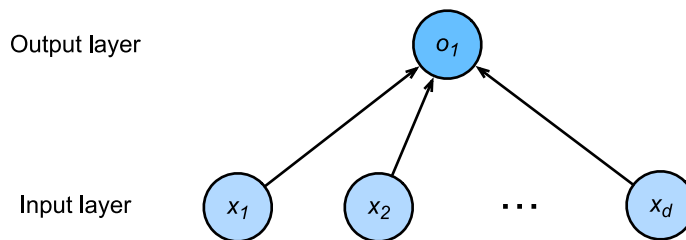
Fig. 3.4: Linear regression is a single-layer neural network.

```
In [5]: net.add(nn.Dense(1))
```

It is worth noting that, in Gluon, we do not need to specify the input shape for each layer, such as the number of linear regression inputs. When the model sees the data, for example, when the `net(X)` is executed later, the model will automatically infer the number of inputs in each layer. We will describe this mechanism in detail in the chapter Deep Learning Computation. Gluon introduces this design to make model development more convenient.

3.3.4 Initialize Model Parameters

Before using `net`, we need to initialize the model parameters, such as the weights and biases in the linear regression model. We will import the `initializer` module from MXNet. This module provides various methods for model parameter initialization. The `init` here is the abbreviation of `initializer`. By `init.Normal(sigma=0.01)` we specify that each weight parameter element is to be randomly

sampled at initialization with a normal distribution with a mean of 0 and standard deviation of 0.01. The bias parameter will be initialized to zero by default.

```
In [6]: from mxnet import init
        net.initialize(init.Normal(sigma=0.01))
```

The code above looks pretty straightforward but in reality something quite strange is happening here. We are initializing parameters for a networks where we haven't told Gluon yet how many dimensions the input will have. It might be 2 as in our example or 2,000, so we couldn't just preallocate enough space to make it work. What happens behind the scenes is that the updates are deferred until the first time that data is sent through the networks. In doing so, we prime all settings (and the user doesn't even need to worry about it). The only cautionary notice is that since the parameters have not been initialized yet, we would not be able to manipulate them yet.

3.3.5 Define the Loss Function

In Gluon, the module `loss` defines various loss functions. We will replace the imported module `loss` with the pseudonym `gloss`, and directly use the squared loss it provides as a loss function for the model.

```
In [7]: from mxnet.gluon import loss as gloss
        loss = gloss.L2Loss()  # The squared loss is also known as the L2 norm loss
```

3.3.6 Define the Optimization Algorithm

Again, we do not need to implement mini-batch stochastic gradient descent. After importing Gluon, we now create a `Trainer` instance and specify a mini-batch stochastic gradient descent with a learning rate of 0.03 (`sgd`) as the optimization algorithm. This optimization algorithm will be used to iterate through all the parameters contained in the `net` instance's nested layers through the `add` function. These parameters can be obtained by the `collect_params` function.

```
In [8]: from mxnet import gluon
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.03})
```

3.3.7 Training

You might have noticed that it was a bit more concise to express our model in Gluon. For example, we didn't have to individually allocate parameters, define our loss function, or implement stochastic gradient descent. The benefits of relying on Gluon's abstractions will grow substantially once we start working with much more complex models. But once we have all the basic pieces in place, the training loop itself is quite similar to what we would do if implementing everything from scratch.

To refresh your memory. For some number of epochs, we'll make a complete pass over the dataset (train_data), grabbing one mini-batch of inputs and the corresponding ground-truth labels at a time. Then, for each batch, we'll go through the following ritual.

- Generate predictions `net(X)` and the loss `l` by executing a forward pass through the network.

- Calculate gradients by making a backwards pass through the network via `l.backward()`.

- Update the model parameters by invoking our SGD optimizer (note that we need not tell trainer.step about which parameters but rather just the amount of data, since we already performed that in the initialization of trainer).

For good measure we compute the loss on the features after each epoch and print it to monitor progress.

```
In [9]: num_epochs = 3
        for epoch in range(1, num_epochs + 1):
            for X, y in data_iter:
                with autograd.record():
                    l = loss(net(X), y)
                l.backward()
                trainer.step(batch_size)
            l = loss(net(features), labels)
            print('epoch %d, loss: %f' % (epoch, l.mean().asnumpy()))

epoch 1, loss: 0.040891
epoch 2, loss: 0.000153
epoch 3, loss: 0.000050
```

The model parameters we have learned and the actual model parameters are compared as below. We get the layer we need from the `net` and access its weight (`weight`) and bias (`bias`). The parameters we have learned and the actual parameters are very close.

```
In [10]: w = net[0].weight.data()
         print('Error in estimating w', true_w.reshape(w.shape) - w)
         b = net[0].bias.data()
         print('Error in estimating b', true_b - b)

Error in estimating w
[[2.9993057e-04 5.2928925e-05]]
<NDArray 1x2 @cpu(0)>
Error in estimating b
[0.00046349]
<NDArray 1 @cpu(0)>
```

Summary

- Using Gluon, we can implement the model more succinctly.

- In Gluon, the module `data` provides tools for data processing, the module `nn` defines a large number of neural network layers, and the module `loss` defines various loss functions.

- MXNet's module `initializer` provides various methods for model parameter initialization.

- Dimensionality and storage are automagically inferred (but caution if you want to access parameters before they've been initialized).

Exercises

1. If we replace `l = loss(output, y)` with `l = loss(output, y).mean()`, we need to change `trainer.step(batch_size)` to `trainer.step(1)` accordingly. Why?

2. Review the MXNet documentation to see what loss functions and initialization methods are provided in the modules `gluon.loss` and `init`. Replace the loss by Huber's loss.

3. How do you access the gradient of `dense.weight`?

Scan the QR Code to Discuss

3.4 Softmax Regression

Over the last two sections we worked through how to implement a linear regression model, both *from scratch* and *using Gluon* to automate most of the repetitive work like allocating and initializing parameters, defining loss functions, and implementing optimizers.

Regression is the hammer we reach for when we want to answer *how much?* or *how many?* questions. If you want to predict the number of dollars (the *price*) at which a house will be sold, or the number of wins a baseball team might have, or the number of days that a patient will remain hospitalized before being discharged, then you're probably looking for a regression model.

In reality, we're more often interested in making categorical assignments.

- Does this email belong in the spam folder or the inbox*?
- How likely is this customer to sign up for subscription service?*
- What is the object in the image (donkey, dog, cat, rooster, etc.)?
- Which object is a customer most likely to purchase?

When we're interested in either assigning datapoints to categories or assessing the *probability* that a category applies, we call this task *classification*. The issue with the models that we studied so far is that they cannot be applied to problems of probability estimation.

3.4.1 Classification Problems

Let's start with an admittedly somewhat contrived image problem where the input image has a height and width of 2 pixels and the color is grayscale. Thus, each pixel value can be represented by a scalar. We record the four pixels in the image as x_1, x_2, x_3, x_4. We assume that the actual labels of the images in the training data set are cat, chicken or dog (assuming that the three animals can be represented by 4 pixels).

To represent these labels we have two choices. Either we set $y \in \{1, 2, 3\}$, where the integers represent {dog, cat, chicken} respectively. This is a great way of *storing* such information on a computer. It would also lend itself rather neatly to regression, but the ordering of outcomes imposes some quite unnatural ordering. In our toy case, this would presume that cats are more similar to chickens than to dogs, at least mathematically. It doesn't work so well in practice either, which is why statisticians invented an alternative approach: one hot encoding via

$$y \in \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\}$$

That is, y is viewed as a three-dimensional vector, where $(1, 0, 0)$ corresponds to cat, $(0, 1, 0)$ to chicken and $(0, 0, 1)$ to dog.

Network Architecture

If we want to estimate multiple classes, we need multiple outputs, matching the number of categories. This is one of the main differences to regression. Because there are 4 features and 3 output animal categories, the weight contains 12 scalars (w with subscripts) and the bias contains 3 scalars (b with subscripts). We compute these three outputs, o_1, o_2, and o_3, for each input:

$$o_1 = x_1 w_{11} + x_2 w_{21} + x_3 w_{31} + x_4 w_{41} + b_1,$$
$$o_2 = x_1 w_{12} + x_2 w_{22} + x_3 w_{32} + x_4 w_{42} + b_2,$$
$$o_3 = x_1 w_{13} + x_2 w_{23} + x_3 w_{33} + x_4 w_{43} + b_3.$$

The neural network diagram below depicts the calculation above. Like linear regression, softmax regression is also a single-layer neural network. Since the calculation of each output, o_1, o_2, and o_3, depends on all inputs, x_1, x_2, x_3, and x_4, the output layer of the softmax regression is also a fully connected layer.
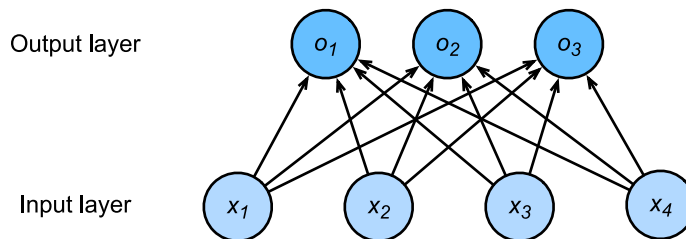
Fig. 3.5: Softmax regression is a single-layer neural network.

Softmax Operation

The chosen notation is somewhat verbose. In vector form we arrive at $\mathbf{o} = \mathbf{W}\mathbf{x} + \mathbf{b}$, which is much more compact to write and code. Since the classification problem requires discrete prediction output, we can use a simple approach to treat the output value o_i as the confidence level of the prediction category i. We can choose the class with the largest output value as the predicted output, which is output $\operatorname{argmax}_i o_i$. For example, if o_1, o_2, and o_3 are 0.1, 10, and 0.1, respectively, then the prediction category is 2, which represents chicken.

However, there are two problems with using the output from the output layer directly. On the one hand, because the range of output values from the output layer is uncertain, it is difficult for us to visually judge the meaning of these values. For instance, the output value 10 from the previous example indicates a level of very confident that the image category is chicken. That is because its output value is 100 times that of the other two categories. However, if $o_1 = o_3 = 10^3$, then an output value of 10 means that the chance for the image category to be chicken is very low. On the other hand, since the actual label has discrete values, the error between these discrete values and the output values from an uncertain range is difficult to measure.

We could try forcing the outputs to correspond to probabilities, but there's no guarantee that on new (unseen) data the probabilities would be nonnegative, let alone sum up to 1. For this kind of discrete value prediction problem, statisticians have invented classification models such as (softmax) logistic regression. Unlike linear regression, the output of softmax regression is subjected to a nonlinearity which ensures that the sum over all outcomes always adds up to 1 and that none of the terms is ever negative. The nonlinear transformation works as follows:

$$\hat{\mathbf{y}} = \operatorname{softmax}(\mathbf{o}) \text{ where } \hat{y}_i = \frac{\exp(o_i)}{\sum_j \exp(o_j)}$$

It is easy to see $\hat{y}_1 + \hat{y}_2 + \hat{y}_3 = 1$ with $0 \le \hat{y}_i \le 1$ for all i. Thus, $\hat{y}$ is a proper probability distribution and the values of o now assume an easily quantifiable meaning. Note that we can still find the most likely class by

$$\hat{\imath}(\mathbf{o}) = \operatorname*{argmax}_i o_i = \operatorname*{argmax}_i \hat{y}_i$$

So, the softmax operation does not change the prediction category output but rather it gives the outputs $\mathbf{o}$ proper meaning. Summarizing it all in vector notation we get $\mathbf{o}^{(i)} = \mathbf{W}\mathbf{x}^{(i)} + \mathbf{b}$ where $\hat{\mathbf{y}}^{(i)} = \operatorname{softmax}(\mathbf{o}^{(i)})$.

Vectorization for Minibatches

To improve computational efficiency further, we usually carry out vector calculations for mini-batches of data. Assume that we are given a mini-batch $\mathbf{X}$ of examples with dimensionality d and batch size n. Moreover, assume that we have q categories (outputs). Then the minibatch features $\mathbf{X}$ are in $\mathbb{R}^{n \times d}$,

weights $\mathbf{W} \in \mathbb{R}^{d \times q}$ and the bias satisfies $\mathbf{b} \in \mathbb{R}^q$.

$$\mathbf{O} = \mathbf{X}\mathbf{W} + \mathbf{b}$$
$$\hat{\mathbf{Y}} = \text{softmax}(\mathbf{O})$$

This accelerates the dominant operation: $\mathbf{W}\mathbf{X}$ from a matrix-vector to a matrix-matrix product. The softmax itself can be computed by exponentiating all entries in $\mathbf{O}$ and then normalizing them by the sum appropriately.

3.4.2 Loss Function

Now that we have some mechanism for outputting probabilities, we need to transform this into a measure of how accurate things are, i.e. we need a *loss function*. For this we use the same concept that we already encountered in linear regression, namely likelihood maximization.

Log-Likelihood

The softmax function maps $\mathbf{o}$ into a vector of probabilities corresponding to various outcomes, such as $p(y = \text{cat}|\mathbf{x})$. This allows us to compare the estimates with reality, simply by checking how well it predicted what we observe.

$$p(Y|X) = \prod_{i=1}^{n} p(y^{(i)}|x^{(i)}) \text{ and thus } -\log p(Y|X) = \sum_{i=1}^{n} -\log p(y^{(i)}|x^{(i)})$$

Minimizing $-\log p(Y|X)$ corresponds to predicting things well. This yields the loss function (we dropped the superscript (i) to avoid notation clutter):

$$l = -\log p(y|x) = -\sum_{j} y_j \log \hat{y}_j$$

Here we used that by construction $\hat{y} = \text{softmax}(\mathbf{o})$ and moreover, that the vector $\mathbf{y}$ consists of all zeroes but for the correct label, such as $(1, 0, 0)$. Hence the the sum over all coordinates j vanishes for all but one term. Since all $\hat{y}_j$ are probabilities, their logarithm is never larger than 0. Consequently, the loss function is minimized if we correctly predict y with *certainty*, i.e. if $p(y|x) = 1$ for the correct label.

Softmax and Derivatives

Since the Softmax and the corresponding loss are so common, it is worth while understanding a bit better how it is computed. Plugging o into the definition of the loss l and using the definition of the softmax we obtain:

$$l = -\sum_{j} y_j \log \hat{y}_j = \sum_{j} y_j \log \sum_{k} \exp(o_k) - \sum_{j} y_j o_j = \log \sum_{k} \exp(o_k) - \sum_{j} y_j o_j$$

3. **Linear Neural Networks**

To understand a bit better what is going on, consider the derivative with respect to o. We get

$$\partial_{o_j} l = \frac{\exp(o_j)}{\sum_k \exp(o_k)} - y_j = \text{softmax}(\mathbf{o})_j - y_j = \Pr(y = j|x) - y_j$$

In other words, the gradient is the difference between what the model thinks should happen, as expressed by the probability $p(y|x)$, and what actually happened, as expressed by y. In this sense, it is very similar to what we saw in regression, where the gradient was the difference between the observation y and estimate $\hat{y}$. This seems too much of a coincidence, and indeed, it isn't. In any exponential family model the gradients of the log-likelihood are given by precisely this term. This fact makes computing gradients a lot easier in practice.

Cross-Entropy Loss

Now consider the case where we don't just observe a single outcome but maybe, an entire distribution over outcomes. We can use the same representation as before for y. The only difference is that rather than a vector containing only binary entries, say $(0, 0, 1)$, we now have a generic probability vector, say $(0.1, 0.2, 0.7)$. The math that we used previously to define the loss l still works out fine, just that the interpretation is slightly more general. It is the expected value of the loss for a distribution over labels.

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_j y_j \log \hat{y}_j$$

This loss is called the cross-entropy loss. It is one of the most commonly used ones for multiclass classification. To demystify its name we need some information theory. The following section can be skipped if needed.

3.4.3 Information Theory Basics

Information theory deals with the problem of encoding, decoding, transmitting and manipulating information (aka data), preferentially in as concise form as possible.

Entropy

A key concept is how many bits of information (or randomness) are contained in data. It can be measured as the entropy of a distribution p via

$$H[p] = \sum_j -p(j) \log p(j)$$

One of the fundamental theorems of information theory states that in order to encode data drawn randomly from the distribution p we need at least $H[p]$ nats' to encode it. If you wonder what a nat' is, it is the

equivalent of bit but when using a code with base e rather than one with base 2. One nat is $\frac{1}{\log(2)} \approx 1.44$ bit. $H[p]/2$ is often also called the binary entropy.

To make this all a bit more theoretical consider the following: $p(1) = \frac{1}{2}$ whereas $p(2) = p(3) = \frac{1}{4}$. In this case we can easily design an optimal code for data drawn from this distribution, by using 0 to encode 1, 10 for 2 and 11 for 3. The expected number of bit is $1.5 = 0.5 * 1 + 0.25 * 2 + 0.25 * 2$. It is easy to check that this is the same as the binary entropy $H[p]/\log 2$.

Kullback Leibler Divergence

One way of measuring the difference between two distributions arises directly from the entropy. Since $H[p]$ is the minimum number of bits that we need to encode data drawn from p, we could ask how well it is encoded if we pick the wrong' distribution q. The amount of extra bits that we need to encode q gives us some idea of how different these two distributions are. Let us compute this directly - recall that to encode j using an optimal code for q would cost $-\log q(j)$ nats, and we need to use this in $p(j)$ of all cases. Hence we have

$$D(p\|q) = -\sum_j p(j)\log q(j) - H[p] = \sum_j p(j)\log\frac{p(j)}{q(j)}$$

Note that minimizing $D(p\|q)$ with respect to q is equivalent to minimizing the cross-entropy loss. This can be seen directly by dropping $H[p]$ which doesn't depend on q. We thus showed that softmax regression tries the minimize the surprise (and thus the number of bits) we experience when seeing the true label y rather than our prediction $\hat{y}$.

3.4.4 Model Prediction and Evaluation

After training the softmax regression model, given any example features, we can predict the probability of each output category. Normally, we use the category with the highest predicted probability as the output category. The prediction is correct if it is consistent with the actual category (label). In the next part of the experiment, we will use accuracy to evaluate the model's performance. This is equal to the ratio between the number of correct predictions and the total number of predictions.

Summary

- We introduced the softmax operation which takes a vector maps it into probabilities.

- Softmax regression applies to classification problems. It uses the probability distribution of the output category in the softmax operation.

- Cross entropy is a good measure of the difference between two probability distributions. It measures the number of bits needed to encode the data given our model.

Exercises

1. Show that the Kullback-Leibler divergence $D(p\|q)$ is nonnegative for all distributions p and q. Hint - use Jensen's inequality, i.e. use the fact that $-\log x$ is a convex function.

2. Show that $\log \sum_j \exp(o_j)$ is a convex function in o.

3. We can explore the connection between exponential families and the softmax in some more depth

   • Compute the second derivative of the cross entropy loss $l(y, \hat{y})$ for the softmax.

   • Compute the variance of the distribution given by $\text{softmax}(o)$ and show that it matches the second derivative computed above.

4. Assume that we three classes which occur with equal probability, i.e. the probability vector is $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.

   • What is the problem if we try to design a binary code for it? Can we match the entropy lower bound on the number of bits?

   • Can you design a better code. Hint - what happens if we try to encode two independent observations? What if we encode n observations jointly?

5. Softmax is a misnomer for the mapping introduced above (but everyone in deep learning uses it). The real softmax is defined as $\text{RealSoftMax}(a, b) = \log(\exp(a) + \exp(b))$.

   • Prove that $\text{RealSoftMax}(a, b) > \max(a, b)$.

   • Prove that this holds for $\lambda^{-1}\text{RealSoftMax}(\lambda a, \lambda b)$, provided that $\lambda > 0$.

   • Show that for $\lambda \to \infty$ we have $\lambda^{-1}\text{RealSoftMax}(\lambda a, \lambda b) \to \max(a, b)$.

   • What does the soft-min look like?

   • Extend this to more than two numbers.

Scan the QR Code to Discuss

3.5 Image Classification Data (Fashion-MNIST)

Before introducing the implementation for softmax regression, we need a suitable dataset. To make things more visually compelling we pick one on classification. It will be used multiple times in later chapters

to allow us to observe the difference between model accuracy and computational efficiency between comparison algorithms. The most commonly used image classification data set is the MNIST handwritten digit recognition data set. It was proposed by LeCun, Cortes and Burges in the 1990s. However, most models have a classification accuracy of over 95% on MNIST, hence it is hard to spot the difference between different models. In order to get a better intuition about the difference between algorithms we use a more complex data set. Fashion-MNIST was proposed by Xiao, Rasul and Vollgraf in 2017.

3.5.1 Getting the Data

First, import the packages or modules required in this section.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet.gluon import data as gdata
        import sys
        import time
```

Next, we will download this data set through Gluon's `data` package. The data is automatically retrieved from the Internet the first time it is called. We specify the acquisition of a training data set, or a testing data set by the parameter `train`. The test data set, also called the testing set, is only used to evaluate the performance of the model and is not used to train the model.

```
In [2]: mnist_train = gdata.vision.FashionMNIST(train=True)
        mnist_test = gdata.vision.FashionMNIST(train=False)
```

The number of images for each category in the training set and the testing set is 6,000 and 1,000, respectively. Since there are 10 categories, the number of examples for the training set and the testing set is 60,000 and 10,000, respectively.

```
In [3]: len(mnist_train), len(mnist_test)

Out[3]: (60000, 10000)
```

We can access any example by square brackets `[]`, and next, we will get the image and label of the first example.

```
In [4]: feature, label = mnist_train[0]
```

The variable `feature` corresponds to an image with a height and width of 28 pixels. Each pixel is an 8-bit unsigned integer (uint8) with values between 0 and 255. It is stored in a 3D NDArray. Its last dimension is the number of channels. Since the data set is a grayscale image, the number of channels is 1. For the sake of simplicity, we will record the shape of the image with the height and width of h and w pixels, respectively, as $h \times w$ or `(h, w)`.

```
In [5]: feature.shape, feature.dtype

Out[5]: ((28, 28, 1), numpy.uint8)
```

The label of each image is represented as a scalar in NumPy. Its type is a 32-bit integer.

```
In [6]: label, type(label), label.dtype
```

```
Out[6]: (2, numpy.int32, dtype('int32'))
```

There are 10 categories in Fashion-MNIST: t-shirt, trousers, pullover, dress, coat, sandal, shirt, sneaker, bag and ankle boot. The following function can convert a numeric label into a corresponding text label.

```
In [7]: # This function has been saved in the d2l package for future use
        def get_fashion_mnist_labels(labels):
            text_labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat',
                           'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
            return [text_labels[int(i)] for i in labels]
```

The following defines a function that can draw multiple images and corresponding labels in a single line.

```
In [8]: # This function has been saved in the d2l package for future use
        def show_fashion_mnist(images, labels):
            d2l.use_svg_display()
            # Here _ means that we ignore (not use) variables
            _, figs = d2l.plt.subplots(1, len(images), figsize=(12, 12))
            for f, img, lbl in zip(figs, images, labels):
                f.imshow(img.reshape((28, 28)).asnumpy())
                f.set_title(lbl)
                f.axes.get_xaxis().set_visible(False)
                f.axes.get_yaxis().set_visible(False)
```

Next, let's take a look at the image contents and text labels for the first nine examples in the training data set.

```
In [9]: X, y = mnist_train[0:9]
        show_fashion_mnist(X, get_fashion_mnist_labels(y))
```


3.5.2 Reading a Minibatch

To make our life easier when reading from the training and test sets we use a `DataLoader` rather than creating one from scratch, as we did in the section on *Linear Regression Implementation Starting from Scratch*. The data loader reads a mini-batch of data with an example number of `batch_size` each time.

In practice, data reading is often a performance bottleneck for training, especially when the model is simple or when the computer is fast. A handy feature of Gluon's `DataLoader` is the ability to use multiple processes to speed up data reading (not currently supported on Windows). For instance, we can set aside 4 processes to read the data (via `num_workers`).

In addition, we convert the image data from uint8 to 32-bit floating point numbers using the `ToTensor` class. Beyond that we divide all numbers by 255 so that all pixels have values between 0 and 1. The `ToTensor` class also moves the image channel from the last dimension to the first dimension to facilitate the convolutional neural network calculations introduced later. Through the `transform_first`

function of the data set, we apply the transformation of `ToTensor` to the first element of each data example (image and label), i.e., the image.

```
In [10]: batch_size = 256
         transformer = gdata.vision.transforms.ToTensor()
         if sys.platform.startswith('win'):
             # 0 means no additional processes are needed to speed up the reading of
             # data
             num_workers = 0
         else:
             num_workers = 4

         train_iter = gdata.DataLoader(mnist_train.transform_first(transformer),
                                       batch_size, shuffle=True,
                                       num_workers=num_workers)
         test_iter = gdata.DataLoader(mnist_test.transform_first(transformer),
                                      batch_size, shuffle=False,
                                      num_workers=num_workers)
```

The logic that we will use to obtain and read the Fashion-MNIST data set is encapsulated in the `d2l.load_data_fashion_mnist` function, which we will use in later chapters. This function will return two variables, `train_iter` and `test_iter`. As the content of this book continues to deepen, we will further improve this function. Its full implementation will be described in the section *Deep Convolutional Neural Networks (AlexNet)*.

Let's look at the time it takes to read the training data.

```
In [11]: start = time.time()
         for X, y in train_iter:
             continue
         '%.2f sec' % (time.time() - start)
Out[11]: '1.19 sec'
```

Summary

- Fashion-MNIST is an apparel classification data set containing 10 categories, which we will use to test the performance of different algorithms in later chapters.

- We store the shape of image using height and width of h and w pixels, respectively, as $h \times w$ or `(h, w)`.

- Data iterators are a key component for efficient performance. Use existing ones if available.

Exercises

1. Does reducing `batch_size` (for instance, to 1) affect read performance?

2. For non-Windows users, try modifying `num_workers` to see how it affects read performance.

3. Use the MXNet documentation to see which other datasets are available in `mxnet.gluon.data.vision`.

4. Use the MXNet documentation to see which other transformations are available in `mxnet.gluon.data.vision.transforms`.

Scan the QR Code to Discuss

3.6 Implementation of Softmax Regression from Scratch

Just like we learned how to implement linear regression from scratch, it is very instructive to do the same for softmax regression. After that we'll repeat the same procedure using Gluon for comparison. We begin with our regular import ritual.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet import autograd, nd
```

We use the Fashion-MNIST data set with batch size 256.

```
In [2]: batch_size = 256
        train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

3.6.1 Initialize Model Parameters

Just as in linear regression, we use vectors to represent examples. Since each example is an image with 28×28 pixels we can store it as a 784 dimensional vector. Moreover, since we have 10 categories, the single layer network has an output dimension of 10. Consequently, the weight and bias parameters of the softmax regression are matrices of size 784×10 and 1×10 respectively. We initialize W with Gaussian noise.

```
In [3]: num_inputs = 784
        num_outputs = 10

        W = nd.random.normal(scale=0.01, shape=(num_inputs, num_outputs))
        b = nd.zeros(num_outputs)
```

As before, we have to attach a gradient to the model parameters.

```
In [4]: W.attach_grad()
        b.attach_grad()
```

3.6.2 The Softmax

Before defining softmax regression let us briefly review how operators such as `sum` work along specific dimensions in an NDArray. Given a matrix `X` we can sum over all elements (default) or only over elements in the same column (`axis=0`) or the same row (`axis=1`). Moreover, we can retain the same dimensionality rather than collapsing out the dimension that we summed over, if required (`keepdims=True`).

```
In [5]: X = nd.array([[1, 2, 3], [4, 5, 6]])
        X.sum(axis=0, keepdims=True), X.sum(axis=1, keepdims=True)

Out[5]: (
         [[5. 7. 9.]]
         <NDArray 1x3 @cpu(0)>,
         [[ 6.]
          [15.]]
         <NDArray 2x1 @cpu(0)>)
```

We can now define the softmax function. For that we first exponentiate each term using `exp` and then sum each row to get the normalization constant. Last we divide each row by its normalization constant and return the result. Before looking at the code, let's look at this in equation form:

$$\text{softmax}(\mathbf{X})_{ij} = \frac{\exp(X_{ij})}{\sum_k \exp(X_{ik})}$$

The denominator is sometimes called the partition function (and its logarithm the log-partition function). The origins of that name are in statistical physics where a related equation models the distribution over an ensemble of particles). Also note that in the definition below we are somewhat sloppy as we do not take any precautions against numerical overflow or underflow due to large (or very small) elements of the matrix, as we did in *Naive Bayes*.

```
In [6]: def softmax(X):
            X_exp = X.exp()
            partition = X_exp.sum(axis=1, keepdims=True)
            return X_exp / partition  # The broadcast mechanism is applied here
```

As you can see, for any random input, we turn each element into a non-negative number. Moreover, each row sums up to 1, as is required for a probability.

```
In [7]: X = nd.random.normal(shape=(2, 5))
        X_prob = softmax(X)
        X_prob, X_prob.sum(axis=1)

Out[7]: (
         [[0.21324193 0.33961776 0.1239742  0.27106097 0.05210521]
          [0.11462264 0.3461234  0.19401033 0.29583326 0.04941036]]
         <NDArray 2x5 @cpu(0)>,
         [1.0000001 1.        ]
         <NDArray 2 @cpu(0)>)
```

3.6.3 The Model

With the softmax operation, we can define the softmax regression model discussed in the last section. We change each original image into a vector with length `num inputs` through the `reshape` function.

```
In [8]: def net(X):
            return softmax(nd.dot(X.reshape((-1, num_inputs)), W) + b)
```

3.6.4 The Loss Function

In the *last section*, we introduced the cross-entropy loss function used by softmax regression. It may be the most common loss function you'll find in all of deep learning. That's because at the moment, classification problems tend to be far more abundant than regression problems.

Recall that it picks the label's predicted probability and takes its logarithm $-\log p(y|x)$. Rather than having to do this using a Python `for` loop (which tends to be inefficient) we have a `pick` function which allows us to select the appropriate terms from the matrix of softmax entries easily. We illustrate this in the case of 3 categories and 2 examples.

```
In [9]: y_hat = nd.array([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
        y = nd.array([0, 2], dtype='int32')
        nd.pick(y_hat, y)
Out[9]:
        [0.1 0.5]
        <NDArray 2 @cpu(0)>
```

This yields the cross-entropy loss function.

```
In [10]: def cross_entropy(y_hat, y):
             return - nd.pick(y_hat, y).log()
```

3.6.5 Classification Accuracy

Given a class of predicted probability distributions `y_hat`, we use the one with the highest predicted probability as the output category. If it is consistent with the actual category `y`, then this prediction is correct. The classification accuracy is the ratio between the number of correct predictions and the total number of predictions made.

To demostrate how to compute accuracy, the function `accuracy` is defined as follows: `y_hat. argmax(axis=1)` returns the largest element index to matrix `y_hat`, the result has the same shape as variable `y`. Now all we need to do is check whether both match. Since the equality operator `==` is datatype-sensitive (e.g. an `int` and a `float32` are never equal), we also need to convert both to the same type (we pick `float32`). The result is an NDArray containing entries of 0 (false) and 1 (true). Taking the mean yields the desired result.

```
In [11]: def accuracy(y_hat, y):
             return (y_hat.argmax(axis=1) == y.astype('float32')).mean().asscalar()
```

We will continue to use the variables `y_hat` and `y` defined in the `pick` function, as the predicted probability distribution and label, respectively. We can see that the first example's prediction category is 2 (the largest element of the row is 0.6 with an index of 2), which is inconsistent with the actual label, 0. The second example's prediction category is 2 (the largest element of the row is 0.5 with an index of 2), which is consistent with the actual label, 2. Therefore, the classification accuracy rate for these two examples is 0.5.

```
In [12]: accuracy(y_hat, y)

Out[12]: 0.5
```

Similarly, we can evaluate the accuracy for model `net` on the data set `data_iter`.

```
In [13]: # The function will be gradually improved: the complete implementation will be
         # discussed in the "Image Augmentation" section
         def evaluate_accuracy(data_iter, net):
             acc_sum, n = 0.0, 0
             for X, y in data_iter:
                 y = y.astype('float32')
                 acc_sum += (net(X).argmax(axis=1) == y).sum().asscalar()
                 n += y.size
             return acc_sum / n
```

Because we initialized the `net` model with random weights, the accuracy of this model should be close to random guessing, i.e. 0.1 for 10 classes.

```
In [14]: evaluate_accuracy(test_iter, net)

Out[14]: 0.0925
```

3.6.6 Model Training

The implementation for training softmax regression is very similar to the implementation of linear regression discussed earlier. We still use the mini-batch stochastic gradient descent to optimize the loss function of the model. When training the model, the number of epochs, `num_epochs`, and learning rate `lr` are both adjustable hyper-parameters. By changing their values, we may be able to increase the classification accuracy of the model.

```
In [15]: num_epochs, lr = 5, 0.1

         # This function has been saved in the d2l package for future use
         def train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size,
                       params=None, lr=None, trainer=None):
             for epoch in range(num_epochs):
                 train_l_sum, train_acc_sum, n = 0.0, 0.0, 0
                 for X, y in train_iter:
                     with autograd.record():
                         y_hat = net(X)
                         l = loss(y_hat, y).sum()
                     l.backward()
                     if trainer is None:
                         d2l.sgd(params, lr, batch_size)
                     else:
                         # This will be illustrated in the next section
```

```
                    trainer.step(batch_size)
                y = y.astype('float32')
                train_l_sum += l.asscalar()
                train_acc_sum += (y_hat.argmax(axis=1) == y).sum().asscalar()
                n += y.size
            test_acc = evaluate_accuracy(test_iter, net)
            print('epoch %d, loss %.4f, train acc %.3f, test acc %.3f'
                  % (epoch + 1, train_l_sum / n, train_acc_sum / n, test_acc))

        train_ch3(net, train_iter, test_iter, cross_entropy, num_epochs,
                  batch_size, [W, b], lr)

epoch 1, loss 0.7886, train acc 0.745, test acc 0.797
epoch 2, loss 0.5743, train acc 0.810, test acc 0.822
epoch 3, loss 0.5291, train acc 0.824, test acc 0.824
epoch 4, loss 0.5054, train acc 0.830, test acc 0.827
epoch 5, loss 0.4896, train acc 0.834, test acc 0.840
```

3.6.7 Prediction

Now that training is complete, we can show how to classify the image. Given a series of images, we will compare their actual labels (first line of text output) and the model predictions (second line of text output).

```
In [16]: for X, y in test_iter:
             break

         true_labels = d2l.get_fashion_mnist_labels(y.asnumpy())
         pred_labels = d2l.get_fashion_mnist_labels(net(X).argmax(axis=1).asnumpy())
         titles = [truelabel + '\n' + predlabel
                   for truelabel, predlabel in zip(true_labels, pred_labels)]

         d2l.show_fashion_mnist(X[0:9], titles[0:9])
```
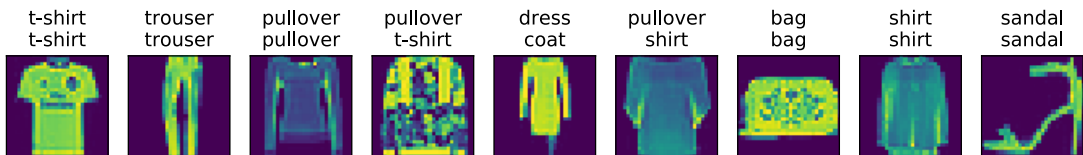

Summary

We can use softmax regression to carry out multi-category classification. Training is very similar to that of linear regression: retrieve and read data, define models and loss functions, then train models using optimization algorithms. In fact, most common deep learning models have a similar training procedure.

Exercises

1. In this section, we directly implemented the softmax function based on the mathematical definition of the softmax operation. What problems might this cause (hint - try to calculate the size of $\exp(50)$)?

2. The function `cross_entropy` in this section is implemented according to the definition of the cross-entropy loss function. What could be the problem with this implementation (hint - consider the domain of the logarithm)?

3. What solutions you can think of to fix the two problems above?

4. Is it always a good idea to return the most likely label. E.g. would you do this for medical diagnosis?

5. Assume that we want to use softmax regression to predict the next word based on some features. What are some problems that might arise from a large vocabulary?

Scan the QR Code to Discuss

3.7 Concise Implementation of Softmax Regression

We already saw that it is much more convenient to use Gluon in the context of *linear regression*. Now we will see how this applies to classification, too. We being with our import ritual.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet import gluon, init
        from mxnet.gluon import loss as gloss, nn
```

We still use the Fashion-MNIST data set and the batch size set from the last section.

```
In [2]: batch_size = 256
        train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

3.7.1 Initialize Model Parameters

As *mentioned previously*, the output layer of softmax regression is a fully connected layer. Therefore, we are adding a fully connected layer with 10 outputs. We initialize the weights at random with zero mean and standard deviation 0.01.

```
In [3]: net = nn.Sequential()
        net.add(nn.Dense(10))
        net.initialize(init.Normal(sigma=0.01))
```

3.7.2 The Softmax

In the previous example, we calculated our model's output and then ran this output through the cross-entropy loss. At its heart it uses `-nd.pick(y_hat, y).log()`. Mathematically, that's a perfectly reasonable thing to do. However, computationally, things can get hairy, as we've already alluded to a few times (e.g. in the context of *Naive Bayes* and in the problem set of the previous chapter). Recall that the softmax function calculates $\hat{y}_j = \frac{e^{z_j}}{\sum_{i=1}^{n} e^{z_i}}$, where $\hat{y}_j$ is the j-th element of `yhat` and z_j is the j-th element of the input `y_linear` variable, as computed by the softmax.

If some of the z_i are very large (i.e. very positive), e^{z_i} might be larger than the largest number we can have for certain types of `float` (i.e. overflow). This would make the denominator (and/or numerator) `inf` and we get zero, or `inf`, or `nan` for $\hat{y}_j$. In any case, we won't get a well-defined return value for `cross_entropy`. This is the reason we subtract $\max(z_i)$ from all z_i first in `softmax` function. You can verify that this shifting in z_i will not change the return value of `softmax`.

After the above subtraction/ normalization step, it is possible that z_j is very negative. Thus, e^{z_j} will be very close to zero and might be rounded to zero due to finite precision (i.e underflow), which makes $\hat{y}_j$ zero and we get $-\inf$ for $\log(\hat{y}_j)$. A few steps down the road in backpropagation, we start to get horrific not-a-number (`nan`) results printed to screen.

Our salvation is that even though we're computing these exponential functions, we ultimately plan to take their log in the cross-entropy functions. It turns out that by combining these two operators `softmax` and `cross_entropy` together, we can elude the numerical stability issues that might otherwise plague us during backpropagation. As shown in the equation below, we avoided calculating e^{z_j} but directly used z_j due to $\log(\exp(\cdot))$.

$$
\begin{aligned}
\log\left(\hat{y}_j\right) &= \log\left(\frac{e^{z_j}}{\sum_{i=1}^{n} e^{z_i}}\right) \\
&= \log\left(e^{z_j}\right) - \log\left(\sum_{i=1}^{n} e^{z_i}\right) \\
&= z_j - \log\left(\sum_{i=1}^{n} e^{z_i}\right)
\end{aligned}
$$

We'll want to keep the conventional softmax function handy in case we ever want to evaluate the probabilities output by our model. But instead of passing softmax probabilities into our new loss function,

we'll just pass $\hat{y}$ and compute the softmax and its log all at once inside the softmax_cross_entropy loss function, which does smart things like the log-sum-exp trick (see on Wikipedia).

```
In [4]: loss = gloss.SoftmaxCrossEntropyLoss()
```

3.7.3 Optimization Algorithm

We use the mini-batch random gradient descent with a learning rate of 0.1 as the optimization algorithm. Note that this is the same choice as for linear regression and it illustrates the portability of the optimizers.

```
In [5]: trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.1})
```

3.7.4 Training

Next, we use the training functions defined in the last section to train a model.

```
In [6]: num_epochs = 5
        d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size, None,
                      None, trainer)

epoch 1, loss 0.7898, train acc 0.747, test acc 0.802
epoch 2, loss 0.5744, train acc 0.811, test acc 0.822
epoch 3, loss 0.5302, train acc 0.822, test acc 0.833
epoch 4, loss 0.5041, train acc 0.831, test acc 0.837
epoch 5, loss 0.4901, train acc 0.834, test acc 0.838
```

Just as before, this algorithm converges to a fairly decent accuracy of 83.7%, albeit this time with a lot fewer lines of code than before. Note that in many cases Gluon takes specific precautions beyond what one would naively do to ensure numerical stability. This takes care of many common pitfalls when coding a model from scratch.

Exercises

1. Try adjusting the hyper-parameters, such as batch size, epoch, and learning rate, to see what the results are.

2. Why might the test accuracy decrease again after a while? How could we fix this?

Scan the QR Code to Discuss

4

Multilayer Perceptrons

In this chapter we will introduce your first truly *deep* networks. The simplest deep networks are called multilayer perceptrons, and they consist of many layers of neurons each fully connected to those in the layer below (from which they receive input) and those above (which they, in turn, influence). When we train high-capacity models we run the risk of overfitting. Thus, we will need to introduce the notions of overfitting, underfitting, and capacity control,. Regularization techniques such as dropout, numerical stability and initialization round out the presentation. Throughout, we focus on applying the models to real data, to give the reader a firm grasp not just of the concepts but also of the practice of using deep networks. Issues of performance, scalability and efficiency are relegated to the next chapters.

4.1 Multilayer Perceptron

In the previous chapters we showed how you could implement multiclass logistic regression (also called softmax regression) for classifying images of clothing into the 10 possible categories. This is where things start to get fun. We understand how to wrangle data, coerce our outputs into a valid probability distribution (via `softmax`), how to apply an appropriate loss function, and how to optimize over our parameters. Now that we've covered these preliminaries, we can extend our toolbox to include deep neural networks.

4.1.1 Hidden Layers

Recall that before, we mapped our inputs directly onto our outputs through a single linear transformation via

$$\hat{\mathbf{o}} = \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b})$$
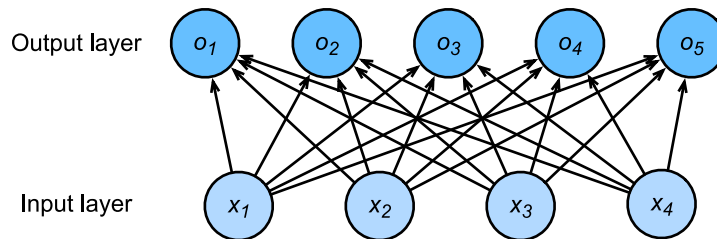
Fig. 4.1: Single layer perceptron with 5 output units.

If our labels really were related to our input data by an approximately linear function, then this approach might be adequate. But linearity is a *strong assumption*. Linearity means that given an output of interest, for each input, increasing the value of the input should either drive the value of the output up or drive it down, irrespective of the value of the other inputs.

From one to many

Imagine the case of classifying cats and dogs based on black and white images. That's like saying that for each pixel, increasing its value either increases the probability that it depicts a dog or decreases it. That's not reasonable. After all, the world contains both black dogs and black cats, and both white dogs and white cats.

Teasing out what is depicted in an image generally requires allowing more complex relationships between our inputs and outputs, considering the possibility that our pattern might be characterized by interactions among the many features. In these cases, linear models will have low accuracy. We can model a more general class of functions by incorporating one or more hidden layers. The easiest way to do this is to stack a bunch of layers of neurons on top of each other. Each layer feeds into the layer above it, until we generate an output. This architecture is commonly called a multilayer perceptron. With an MLP, we stack a bunch of layers on top of each other. Here's an example:
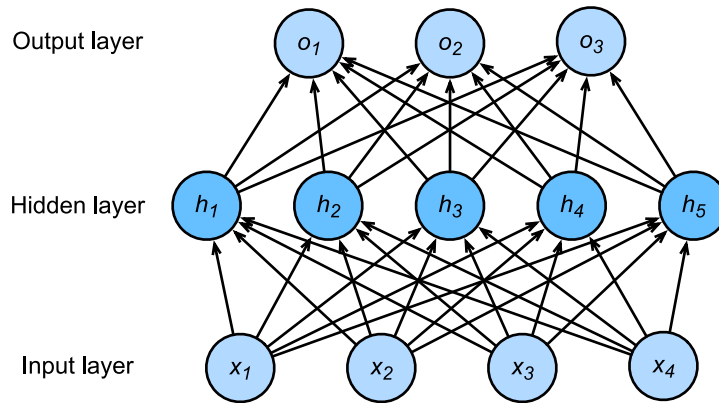
Fig. 4.2: Multilayer perceptron with hidden layers. This example contains a hidden layer with 5 hidden units in it.

In the multilayer perceptron above, the number of inputs and outputs is 4 and 3 respectively, and the hidden layer in the middle contains 5 hidden units. Since the input layer does not involve any calculations, there are a total of 2 layers in the multilayer perceptron. The neurons in the hidden layer are fully connected to the inputs within the input layer. The neurons in the output layer and the neurons in the hidden layer are also fully connected. Therefore, both the hidden layer and the output layer in the multilayer perceptron are fully connected layers.

From linear to nonlinear

Let us write out what is happening mathematically in the picture above, e.g. for multiclass classification.

$$\mathbf{h} = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$$
$$\mathbf{o} = \mathbf{W}_2\mathbf{h} + \mathbf{b}_2$$
$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{o})$$

The problem with the approach above is that we have gained nothing over a simple single layer perceptron since we can collapse out the hidden layer by an equivalently parametrized single layer perceptron using $\mathbf{W} = \mathbf{W}_2\mathbf{W}_1$ and $\mathbf{b} = \mathbf{W}_2\mathbf{b}_1 + \mathbf{b}_2$.

$$\mathbf{o} = \mathbf{W}_2\mathbf{h} + \mathbf{b}_2 = \mathbf{W}_2(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2 = (\mathbf{W}_2\mathbf{W}_1)\mathbf{x} + (\mathbf{W}_2\mathbf{b}_1 + \mathbf{b}_2) = \mathbf{W}\mathbf{x} + \mathbf{b}$$

To fix this we need another key ingredient - a nonlinearity σ such as $\max(x, 0)$ after each layer. Once we do this, it becomes impossible to merge layers. This yields

$$\mathbf{h} = \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$$
$$\mathbf{o} = \mathbf{W}_2\mathbf{h} + \mathbf{b}_2$$
$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{o})$$

Clearly we could continue stacking such hidden layers, e.g. $\mathbf{h}_1 = \sigma(\mathbf{W}_1\mathbf{x}+\mathbf{b}_1)$ and $\mathbf{h}_2 = \sigma(\mathbf{W}_2\mathbf{h}_1+\mathbf{b}_2)$ on top of each other to obtain a true multilayer perceptron.

Multilayer perceptrons can account for complex interactions in the inputs because the hidden neurons depend on the values of each of the inputs. It's easy to design a hidden node that that does arbitrary computation, such as, for instance, logical operations on its inputs. And it's even widely known that multilayer perceptrons are universal approximators. That means that even for a single-hidden-layer neural network, with enough nodes, and the right set of weights, it could model any function at all! Actually learning that function is the hard part. And it turns out that we can approximate functions much more compactly if we use deeper (vs wider) neural networks. We'll get more into the math in a subsequent chapter, but for now let's actually build an MLP. In this example, we'll implement a multilayer perceptron with two hidden layers and one output layer.

Vectorization and mini-batch

When given a mini-batch of samples we can use vectorization to gain better efficiency in implementation. In a nutshell, we replace vectors by matrices. As before, denote by $\mathbf{X}$ the matrix of inputs from a minibatch. Then an MLP with two hidden layers can be expressed as

$$\mathbf{H}_1 = \sigma(\mathbf{W}_1\mathbf{X} + \mathbf{b}_1)$$
$$\mathbf{H}_2 = \sigma(\mathbf{W}_2\mathbf{H}_1 + \mathbf{b}_2)$$
$$\mathbf{O} = \mathrm{softmax}(\mathbf{W}_3\mathbf{H}_2 + \mathbf{b}_3)$$

This is easy to implement and easy to optimize. With some abuse of notation we define the nonlinearity σ to apply to its inputs on a row-wise fashion, i.e. one observation at a time, often one coordinate at a time. This is true for most activation functions (the *batch normalization* is a notable exception from that rule).

4.1.2 Activation Functions

Let us look a bit more at examples of activation functions. After all, it is this alternation between linear and nonlinear terms that makes deep networks work. A rather popular choice, due to its simplicity of implementation and its efficacy is the ReLu function.

ReLU Function

The ReLU (rectified linear unit) function provides a very simple nonlinear transformation. Given the element x, the function is defined as

$$\mathrm{ReLU}(x) = \max(x, 0).$$

It can be understood that the ReLU function retains only positive elements and discards negative elements. To get a better idea of what it looks like it helps to plot it. For convenience we define a plotting function

`xyplot` to take care of the gruntwork.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet import autograd, nd

        def xyplot(x_vals, y_vals, name):
            d2l.set_figsize(figsize=(5, 2.5))
            d2l.plt.plot(x_vals.asnumpy(), y_vals.asnumpy())
            d2l.plt.xlabel('x')
            d2l.plt.ylabel(name + '(x)')
```

Then, we can plot the ReLU function using the `relu` function provided by NDArray. As you can see, the activation function is a two-stage linear function.

```
In [2]: x = nd.arange(-8.0, 8.0, 0.1)
        x.attach_grad()
        with autograd.record():
            y = x.relu()
        xyplot(x, y, 'relu')
```
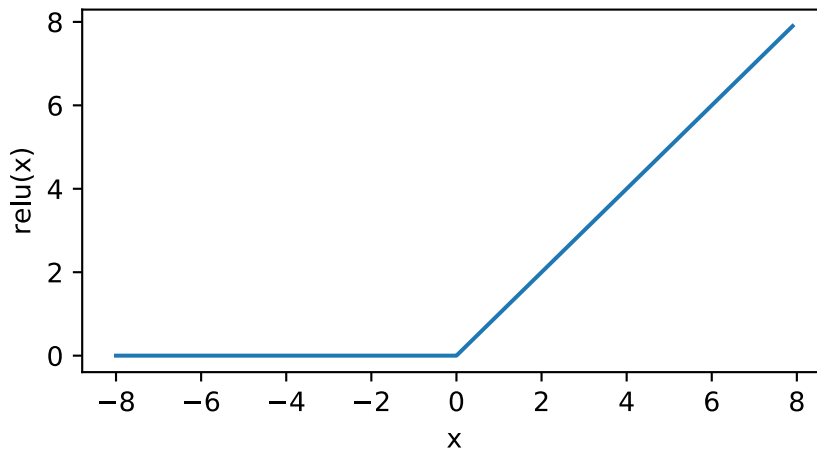

Obviously, when the input is negative, the derivative of ReLU function is 0; when the input is positive, the derivative of ReLU function is 1. Note that the ReLU function is not differentiable when the input is 0. Instead, we pick its left-hand-side (LHS) derivative 0 at location 0. The derivative of the ReLU function is plotted below.

```
In [3]: y.backward()
        xyplot(x, x.grad, 'grad of relu')
```
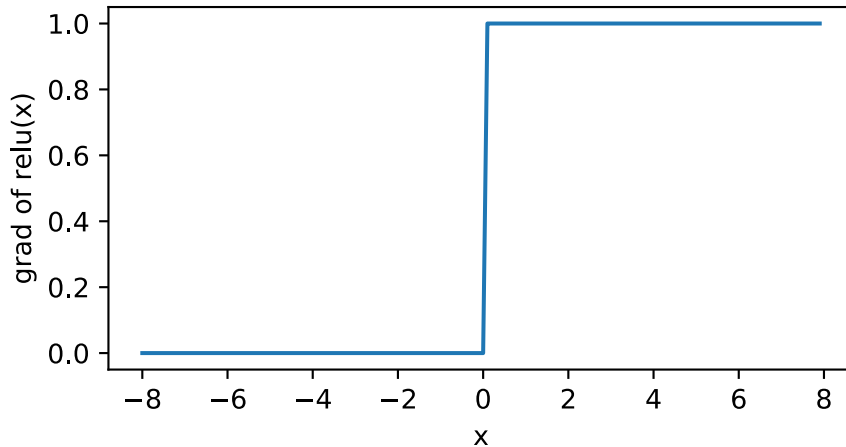
Note that there are many variants to the ReLU function, such as the parameterized ReLU (pReLU) of He et al., 2015. Effectively it adds a linear term to the ReLU, so some information still gets through, even when the argument is negative.

$$\mathrm{pReLU}(x) = \max(0, x) - \alpha x$$

The reason for using the ReLU is that its derivatives are particularly well behaved - either they vanish or they just let the argument through. This makes optimization better behaved and it reduces the issue of the vanishing gradient problem (more on this later).

Sigmoid Function

The Sigmoid function can transform the value of an element in $\mathbb{R}$ to the interval $(0, 1)$.

$$\mathrm{sigmoid}(x) = \frac{1}{1 + \exp(-x)}.$$

The Sigmoid function was commonly used in early neural networks, but is currently being replaced by the simpler ReLU function. In the Recurrent Neural Network chapter, we will describe how to utilize the function's ability to control the flow of information in a neural network thanks to its capacity to transform the value range between 0 and 1. The derivative of the Sigmoid function is plotted below. When the input is close to 0, the Sigmoid function approaches a linear transformation.

```
In [4]: with autograd.record():
            y = x.sigmoid()
        xyplot(x, y, 'sigmoid')
```
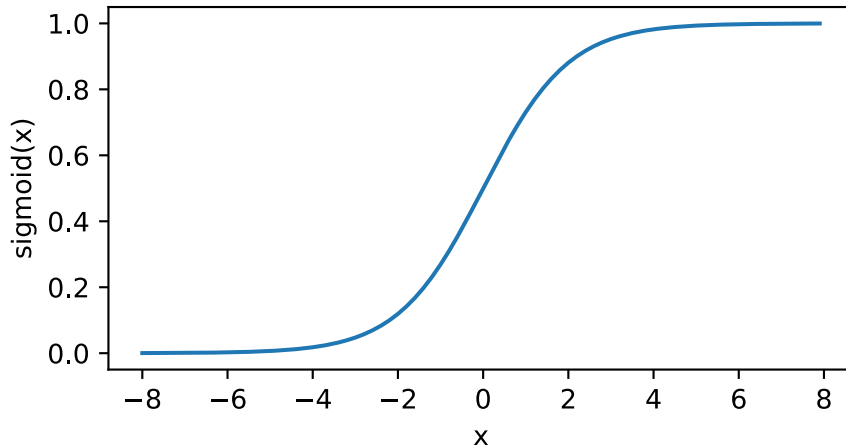
The derivative of Sigmoid function is as follows:

$$\frac{d}{dx}\text{sigmoid}(x) = \frac{\exp(-x)}{(1 + \exp(-x))^2} = \text{sigmoid}(x)\left(1 - \text{sigmoid}(x)\right).$$

The derivative of Sigmoid function is plotted below. When the input is 0, the derivative of the Sigmoid function reaches a maximum of 0.25; as the input deviates further from 0, the derivative of Sigmoid function approaches 0.

```
In [5]: y.backward()
        xyplot(x, x.grad, 'grad of sigmoid')
```
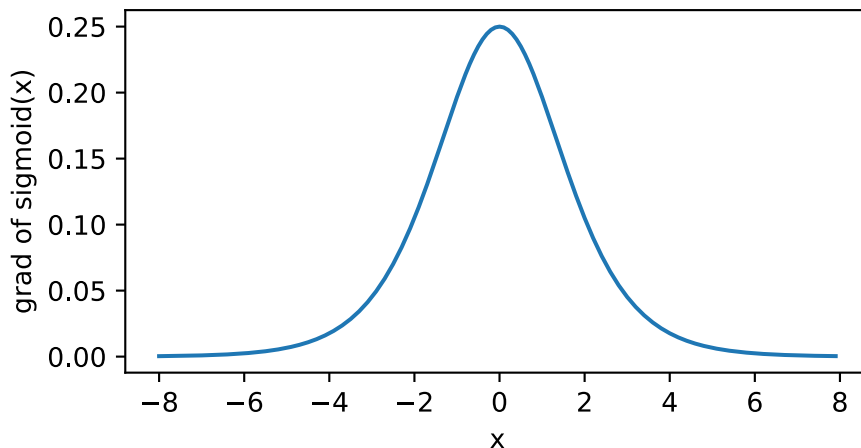
Tanh Function

The Tanh (Hyperbolic Tangent) function transforms the value of an element to the interval between -1 and 1:

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}.$$

We can then plot the Tanh function. As the input nears 0, the Tanh function approaches linear transformation. Although the shape of the function is similar to that of the Sigmoid function, the Tanh function is symmetric at the origin of the coordinate system.

```
In [6]: with autograd.record():
            y = x.tanh()
        xyplot(x, y, 'tanh')
```
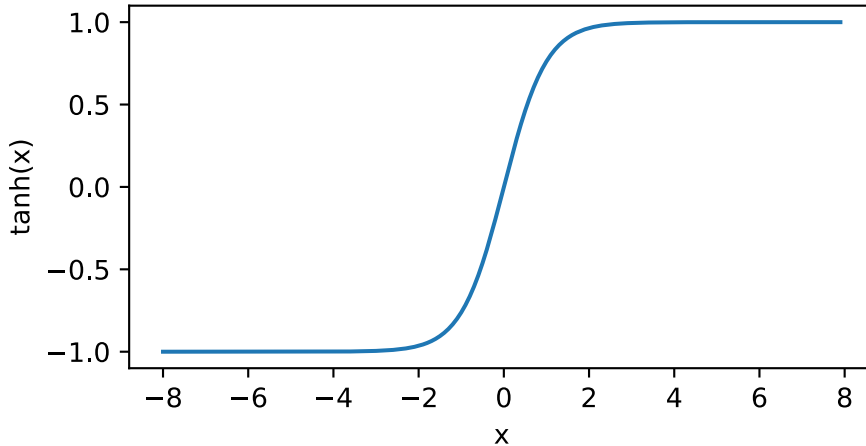

The derivative of the Tanh function is:

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x).$$

The derivative of Tanh function is plotted below. As the input nears 0, the derivative of the Tanh function approaches a maximum of 1; as the input deviates away from 0, the derivative of the Tanh function approaches 0.

```
In [7]: y.backward()
        xyplot(x, x.grad, 'grad of tanh')
```
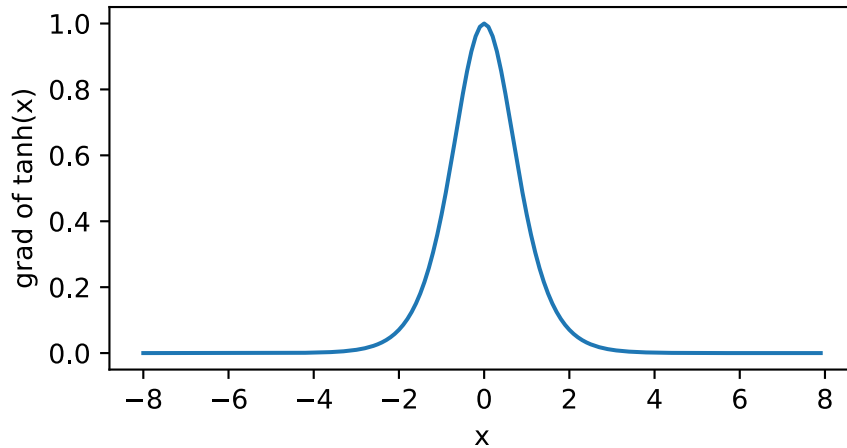
In summary, we have a range of nonlinearities and now know how to layer them to build quite powerful network architectures. As a side note, we have now pretty much reached the state of the art in deep learning, anno 1990. The main difference is that we have a powerful deep learning framework which lets us build models in a few lines of code where previously thousands of lines of C and Fortran would have been needed.

Summary

- The multilayer perceptron adds one or multiple fully connected hidden layers between the output and input layers and transforms the output of the hidden layer via an activation function.

- Commonly used activation functions include the ReLU function, the Sigmoid function, and the Tanh function.

Exercises

1. Compute the derivative of the Tanh and the pReLU activation function.

2. Show that a multilayer perceptron using only ReLU (or pReLU) constructs a continuous piecewise linear function.

3. Show that $\tanh(x) + 1 = 2\mathrm{sigmoid}(2x)$.

4. Assume we have a multilayer perceptron *without* nonlinearities between the layers. In particular, assume that we have d input dimensions, d output dimensions and that one of the layers had only $d/2$ dimensions. Show that this network is less expressive (powerful) than a single layer perceptron.

5. Assume that we have a nonlinearity that applies to one minibatch at a time. What kinds of problems to you expect this to cause?

4.2 Implementation of Multilayer Perceptron from Scratch

Now that we learned how multilayer perceptrons (MLPs) work in theory, let's implement them. First, import the required packages or modules.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet import nd
        from mxnet.gluon import loss as gloss
```

We continue to use the Fashion-MNIST data set. We will use the Multilayer Perceptron for image classification

```
In [2]: batch_size = 256
        train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
```

4.2.1 Initialize Model Parameters

We know that the dataset contains 10 classes and that the images are of $28 \times 28 = 784$ pixel resolution. Thus the number of inputs is 784 and the number of outputs is 10. Moreover, we use an MLP with one hidden layer and we set the number of hidden units to 256, but we could have picked some other value for this *hyperparameter*, too. Typically one uses powers of 2 since things align more nicely in memory.

```
In [3]: num_inputs, num_outputs, num_hiddens = 784, 10, 256

        W1 = nd.random.normal(scale=0.01, shape=(num_inputs, num_hiddens))
        b1 = nd.zeros(num_hiddens)
        W2 = nd.random.normal(scale=0.01, shape=(num_hiddens, num_outputs))
        b2 = nd.zeros(num_outputs)
        params = [W1, b1, W2, b2]

        for param in params:
            param.attach_grad()
```

4.2.2 Activation Function

Here, we use the underlying `maximum` function to implement the ReLU, instead of invoking `ReLU` directly.

```
In [4]: def relu(X):
            return nd.maximum(X, 0)
```

4.2.3 The model

As in softmax regression, using `reshape` we change each original image to a length vector of `num_inputs`. We then implement implement the MLP just as discussed previously.

```
In [5]: def net(X):
            X = X.reshape((-1, num_inputs))
            H = relu(nd.dot(X, W1) + b1)
            return nd.dot(H, W2) + b2
```

4.2.4 The Loss Function

For better numerical stability, we use Gluon's functions, including softmax calculation and cross-entropy loss calculation. We discussed the intricacies of that in the *previous section*. This is simply to avoid lots of fairly detailed and specific code (the interested reader is welcome to look at the source code for more details, something that is useful for implementing other related functions).

```
In [6]: loss = gloss.SoftmaxCrossEntropyLoss()
```

4.2.5 Training

Steps for training the Multilayer Perceptron are no different from Softmax Regression training steps. In the `d2l` package, we directly call the `train_ch3` function, whose implementation was introduced here. We set the number of epochs to 10 and the learning rate to 0.5.

```
In [7]: num_epochs, lr = 10, 0.5
        d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size,
                      params, lr)

epoch 1, loss 0.8017, train acc 0.698, test acc 0.793
epoch 2, loss 0.4882, train acc 0.820, test acc 0.851
epoch 3, loss 0.4241, train acc 0.842, test acc 0.846
epoch 4, loss 0.3935, train acc 0.853, test acc 0.868
epoch 5, loss 0.3784, train acc 0.859, test acc 0.870
epoch 6, loss 0.3551, train acc 0.870, test acc 0.871
epoch 7, loss 0.3348, train acc 0.876, test acc 0.869
epoch 8, loss 0.3277, train acc 0.880, test acc 0.880
epoch 9, loss 0.3171, train acc 0.883, test acc 0.882
epoch 10, loss 0.3046, train acc 0.887, test acc 0.885
```

To see how well we did, let's apply the model to some test data. If you're interested, compare the result to corresponding linear model.

```
In [8]: for X, y in test_iter:
            break

        true_labels = d2l.get_fashion_mnist_labels(y.asnumpy())
        pred_labels = d2l.get_fashion_mnist_labels(net(X).argmax(axis=1).asnumpy())
        titles = [truelabel + '\n' + predlabel
                    for truelabel, predlabel in zip(true_labels, pred_labels)]

        d2l.show_fashion_mnist(X[0:9], titles[0:9])
```
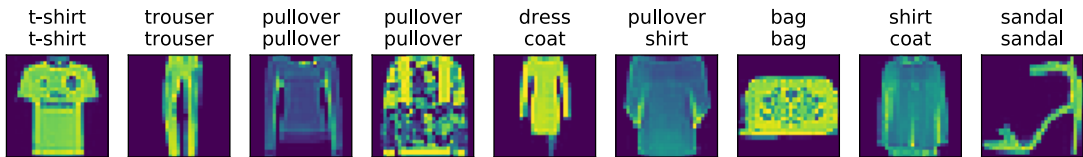

This looks slightly better than before, a clear sign that we're on to something good here.

Summary

We saw that implementing a simple MLP is quite easy, when done manually. That said, for a large number of layers this can get quite complicated (e.g. naming the model parameters, etc).

Exercises

1. Change the value of the hyper-parameter `num_hiddens` in order to see the result effects.

2. Try adding a new hidden layer to see how it affects the results.

3. How does changing the learning rate change the result.

4. What is the best result you can get by optimizing over all the parameters (learning rate, iterations, number of hidden layers, number of hidden units per layer)?

Scan the QR Code to Discuss

4.3 Concise Implementation of Multilayer Perceptron

Now that we learned how multilayer perceptrons (MLPs) work in theory, let's implement them. We begin, as always, by importing modules.

```
In [1]: import sys
        sys.path.insert(0, '..')

        import d2l
        from mxnet import gluon, init
        from mxnet.gluon import loss as gloss, nn
```

4.3.1 The Model

The only difference from the softmax regression is the addition of a fully connected layer as a hidden layer. It has 256 hidden units and uses ReLU as the activation function.

```
In [2]: net = nn.Sequential()
        net.add(nn.Dense(256, activation='relu'))
        net.add(nn.Dense(10))
        net.initialize(init.Normal(sigma=0.01))
```

One minor detail is of note when invoking `net.add()`. It adds one or more layers to the network. That is, an equivalent to the above lines would be `net.add(nn.Dense(256, activation='relu'), nn.Dense(10))`. Also note that Gluon automagically infers the missing parameteters, such as the fact that the second layer needs a matrix of size 256×10. This happens the first time the network is invoked.

We use almost the same steps for softmax regression training as we do for reading and training the model.

```
In [3]: batch_size = 256
        train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)

        loss = gloss.SoftmaxCrossEntropyLoss()
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.5})
        num_epochs = 10
        d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size, None,
                      None, trainer)

epoch 1, loss 0.8218, train acc 0.694, test acc 0.820
epoch 2, loss 0.5017, train acc 0.814, test acc 0.843
epoch 3, loss 0.4377, train acc 0.839, test acc 0.862
epoch 4, loss 0.4001, train acc 0.852, test acc 0.865
epoch 5, loss 0.3744, train acc 0.863, test acc 0.858
epoch 6, loss 0.3626, train acc 0.867, test acc 0.878
epoch 7, loss 0.3401, train acc 0.875, test acc 0.873
epoch 8, loss 0.3285, train acc 0.879, test acc 0.870
epoch 9, loss 0.3206, train acc 0.881, test acc 0.880
epoch 10, loss 0.3123, train acc 0.885, test acc 0.882
```

Exercises

1. Try adding a few more hidden layers to see how the result changes.

2. Try out different activation functions. Which ones work best?

3. Try out different initializations of the weights.

Scan the QR Code to Discuss

4.4 Model Selection, Underfitting and Overfitting

In machine learning, our goal is to discover general patterns. For example, we might want to learn an association between genetic markers and the development of dementia in adulthood. Our hope would be to uncover a pattern that could be applied successfully to assess risk for the entire population.

However, when we train models, we don't have access to the entire population (or current or potential humans). Instead, we can access only a small, finite sample. Even in a large hospital system, we might get hundreds of thousands of medical records. Given such a finite sample size, it's possible to uncover spurious associations that don't hold up for unseen data.

Let's consider an extreme pathological case. Imagine that you want to learn to predict which people will repay their loans. A lender hires you as a data scientist to investigate the case and gives you complete files on 100 applicants, of which 5 defaulted on their loans within 3 years. The files might include hundreds of features including income, occupation, credit score, length of employment etcetera. Imagine that they additionally give you video footage of their interview with a lending agent. That might seem like a lot of data!

Now suppose that after generating an enormous set of features, you discover that of the 5 applicants who defaults, all 5 were wearing blue shirts during their interviews, while only 40% of general population wore blue shirts. There's a good chance that any model you train would pick up on this signal and use it as an important part of its learned pattern.

Even if defaulters are no more likely to wear blue shirts, there's a 1% chance that we'll observe all five defaulters wearing blue shirts. And keeping the sample size low while we have hundreds or thousands of features, we may observe a large number of spurious correlations. Given trillions of training examples, these false associations might disappear. But we seldom have that luxury.

The phenomena of fitting our training distribution more closely than the real distribution is called overfitting, and the techniques used to combat overfitting are called regularization. More to the point, in the previous sections we observed this effect on the Fashion-MNIST dataset. If you altered the model structure or the hyper-parameters during the experiment, you may have found that for some choices the model might not have been as accurate when using the testing data set compared to when the training data set was used.

4.4.1 Training Error and Generalization Error

Before we can explain this phenomenon, we need to differentiate between training and a generalization error. In layman's terms, training error refers to the error exhibited by the model during its use of the training data set and generalization error refers to any expected error when applying the model to an imaginary stream of additional data drawn from the underlying data distribution. The latter is often estimated by applying the model to the test set. In the previously discussed loss functions, for example, the squared loss function used for linear regression or the cross-entropy loss function used for softmax regression, can be used to calculate training and generalization error rates.

The following three thought experiments will help illustrate this situation better. Consider a college student trying to prepare for his final exam. A diligent student will strive to practice well and test his abilities using exams from previous years. Nonetheless, doing well on past exams is no guarantee that he will excel when it matters. For instance, the student might try to prepare by rote learning the answers to the exam questions. This requires the student to memorize many things. He might even remember the answers for past exams perfectly. Another student might prepare by trying to understand the reasons for giving certain answers. In most cases, the latter student will do much better.

Likewise, we would expect that a model that simply performs table lookup to answer questions. If the inputs are discrete, this might very well work after seeing *many* examples. Nonetheless, such a model is unlikely to work well in practice, as data is often real-valued and more scarce than we would like. Furthermore, we only have a finite amount of RAM to store our model in.

Lastly, consider a trivial classification problem. Our training data consists of labels only, namely 0 for heads and 1 for tails, obtained by tossing a fair coin. No matter what we do, the generalization error will always be $\frac{1}{2}$. Yet the training error may be quite a bit less than that, depending on the luck of the draw. E.g. for the dataset $\{0, 1, 1, 1, 0, 1\}$ we will predict' class 1 and incur an error of $\frac{1}{3}$, which is considerably better than what it should be. We can also see that as we increase the amount of data, the probability for large deviations from $\frac{1}{2}$ will diminish and the training error will be close to it, too. This is because our model overfit' to the data and since things will average out' as we increase the amount of data.

Statistical Learning Theory

There is a formal theory of this phenomenon. Glivenko and Cantelli derived in their eponymous theorem the rate at which the training error converges to the generalization error. In a series of seminal papers Vapnik and Chervonenkis extended this to much more general function classes. This laid the foundations of Statistical Learning Theory.

Unless stated otherwise, we assume that both the training set and the test set are drawn independently and identically drawn from the same distribution. This means that in drawing from the distribution there is no memory between draws. Moreover, it means that we use the same distribution in both cases. Obvious cases where this might be violated is if we want to build a face recognition by training it on elementary students and then want to deploy it in the general population. This is unlikely to work since, well, the students tend to look quite from the general population. By training we try to find a function that does particularly well on the training data. If the function is very flexible such as to be able to adapt well to any details in the training data, it might do a bit too well. This is precisely what we want to avoid (or at least control). Instead we want to find a model that reduces the generalization error. A lot of tuning in deep learning is devoted to making sure that this does not happen.

Model Complexity

When we have simple models and abundant data, we expect the generalization error to resemble the training error. When we work with more complex models and fewer examples, we expect the training error to go down but the generalization gap to grow. What precisely constitutes model complexity is a complex matter. Many factors govern whether a model will generalize well. For example a model with more parameters might be considered more complex. A model whose parameters can take a wider range of values might be more complex. Often with neural networks, we think of a model that takes more training steps as more complex, and one subject to early stopping as less complex.

It can be difficult to compare the complexity among members of very different model classes (say decision trees versus neural networks). For now a simple rule of thumb is quite useful: A model that can readily explain arbitrary facts is what statisticians view as complex, whereas one that has only a limited expressive power but still manages to explain the data well is probably closer to the truth. In philosophy this is closely related to Popper's criterion of falsifiability of a scientific theory: a theory is good if it fits data and if there are specific tests which can be used to disprove it. This is important since all statistical estimation is post hoc, i.e. we estimate after we observe the facts, hence vulnerable to the associated fallacy. Ok, enough of philosophy, let's get to more tangible issues. To give you some intuition in this chapter, we'll focus on a few factors that tend to influence the generalizability of a model class:

1. The number of tunable parameters. When the number of tunable parameters, sometimes denoted as the number of degrees of freedom, is large, models tend to be more susceptible to overfitting.

2. The values taken by the parameters. When weights can take a wider range of values, models can be more susceptible to over fitting.

3. The number of training examples. It's trivially easy to overfit a dataset containing only one or two examples even if your model is simple. But overfitting a dataset with millions of examples requires an extremely flexible model.

4.4.2 Model Selection

In machine learning we usually select our model based on an evaluation of the performance of several candidate models. This process is called model selection. The candidate models can be similar models

using different hyper-parameters. Using the multilayer perceptron as an example, we can select the number of hidden layers as well as the number of hidden units, and activation functions in each hidden layer. A significant effort in model selection is usually required in order to end up with an effective model. In the following section we will be describing the validation data set often used in model selection.

Validation Data Set

Strictly speaking, the test set can only be used after all the hyper-parameters and model parameters have been selected. In particular, the test data must not be used in model selection process, such as in the tuning of hyper-parameters. We should not rely solely on the training data during model selection, since the generalization error rate cannot be estimated from the training error rate. Bearing this in mind, we can reserve a portion of data outside of the training and testing data sets to be used in model selection. This reserved data is known as the validation data set, or validation set. For example, a small, randomly selected portion from a given training set can be used as a validation set, with the remainder used as the true training set.

However, in practical applications the test data is rarely discarded after one use since it's not easily obtainable. Therefore, in practice, there may be unclear boundaries between validation and testing data sets. Unless explicitly stated, the test data sets used in the experiments provided in this book should be considered as validation sets, and the test accuracy in the experiment report are for validation accuracy. The good news is that we don't need too much data in the validation set. The uncertainty in our estimates can be shown to be of the order of $O(n^{-\frac{1}{2}})$.

K-Fold Cross-Validation

When there is not enough training data, it is considered excessive to reserve a large amount of validation data, since the validation data set does not play a part in model training. A solution to this is the K-fold cross-validation method. In K-fold cross-validation, the original training data set is split into K non-coincident sub-data sets. Next, the model training and validation process is repeated K times. Every time the validation process is repeated, we validate the model using a sub-data set and use the $K-1$ sub-data set to train the model. The sub-data set used to validate the model is continuously changed throughout this K training and validation process. Finally, the average over K training and validation error rates are calculated respectively.

4.4.3 Underfitting and Overfitting

Next, we will look into two common problems that occur during model training. One type of problem occurs when the model is unable to reduce training errors since the model is too simplistic. This phenomenon is known as underfitting. As discussed, another type of problem is when the number of model training errors is significantly less than that of the testing data set, also known as overfitting. In practice, both underfitting and overfitting should be dealt with simultaneously whenever possible. Although many factors could cause the above two fitting problems, for the time being we'll be focusing primarily on two factors: model complexity and training data set size.

Model Complexity

We use polynomials as a way to illustrate the issue. Given training data consisting of the scalar data feature x and the corresponding scalar label y, we try to find a polynomial of degree d

$$\hat{y} = \sum_{i=0}^{d} x^i w_i$$

to estimate y. Here w_i refers to the model's weight parameter. The bias is implicit in w_0 since $x^0 = 1$. Similar to linear regression we also use a squared loss for simplicity (note that $d = 1$ we recover linear regression).

A higher order polynomial function is more complex than a lower order polynomial function, since the higher-order polynomial has more parameters and the model function's selection range is wider. Therefore, using the same training data set, higher order polynomial functions should be able to achieve a lower training error rate (relative to lower degree polynomials). Bearing in mind the given training data set, the typical relationship between model complexity and error is shown in the diagram below. If the model is too simple for the dataset, we are likely to see underfitting, whereas if we pick an overly complex model we see overfitting. Choosing an appropriately complex model for the data set is one way to avoid underfitting and overfitting.
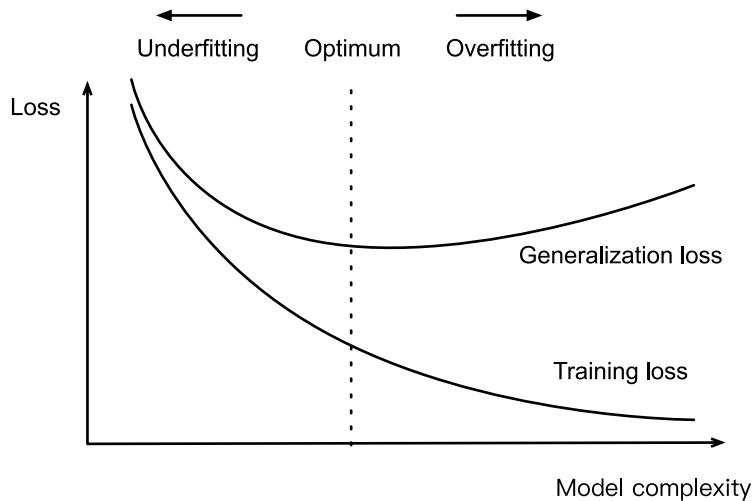
Fig. 4.3: Influence of Model Complexity on Underfitting and Overfitting

Data Set Size

Another influence is the amount of training data. Typically, if there are not enough samples in the training data set, especially if the number of samples is less than the number of model parameters (count by

element), overfitting is more likely to occur. Additionally, as we increase the amount of training data, the generalization error typically decreases. This means that more data never hurts. Moreover, it also means that we should typically only use complex models (e.g. many layers) if we have sufficient data.

4.4.4 Polynomial Regression

Let us try how this works in practice by fitting polynomials to data. As before we start by importing some modules.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet import autograd, gluon, nd
        from mxnet.gluon import data as gdata, loss as gloss, nn
```

Generating Data Sets

First we need data. Given x we will use the following cubic polynomial to generate the labels on training and test data:

$$y = 5 + 1.2x - 3.4\frac{x^2}{2!} + 5.6\frac{x^3}{3!} + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.1)$$

The noise term ϵ obeys a normal distribution with a mean of 0 and a standard deviation of 0.1. The number of samples for both the training and the testing data sets is set to 100.

```
In [2]: maxdegree = 20  # Maximum degree of the polynomial
        n_train, n_test = 100, 1000  # Training and test data set sizes
        true_w = nd.zeros(maxdegree)  # Allocate lots of empty space
        true_w[0:4] = nd.array([5, 1.2, -3.4, 5.6])

        features = nd.random.normal(shape=(n_train + n_test, 1))
        features = nd.random.shuffle(features)
        poly_features = nd.power(features, nd.arange(maxdegree).reshape((1, -1)))
        poly_features = poly_features / (
            nd.gamma(nd.arange(maxdegree) + 1).reshape((1, -1)))
        labels = nd.dot(poly_features, true_w)
        labels += nd.random.normal(scale=0.1, shape=labels.shape)
```

For optimization we typically want to avoid very large values of gradients, losses, etc.; This is why the monomials stored in `poly_features` are rescaled from x^i to $\frac{1}{i!}x^i$. It allows us to avoid very large values for large exponents i. Factorials are implemented in Gluon using the Gamma function, where $n! = \Gamma(n+1)$.

Take a look at the first 2 samples from the generated data set. The value 1 is technically a feature, namely the constant feature corresponding to the bias.

```
In [3]: features[:2], poly_features[:2], labels[:2]
```

```
Out[3]: (
        [[-0.5095612 ]
         [ 0.34202248]]
        <NDArray 2x1 @cpu(0)>,
        [[ 1.00000000e+00 -5.09561181e-01  1.29826277e-01 -2.20514797e-02
           2.80914456e-03 -2.86286173e-04  2.43133891e-05 -1.76987987e-06
           1.12732764e-07 -6.38269260e-09  3.25237282e-10 -1.50662070e-11
           6.39762874e-13 -2.50767950e-14  9.12725858e-16 -3.10059752e-17
           9.87465261e-19 -2.95984643e-20  8.37901598e-22 -2.24716890e-23]
         [ 1.00000000e+00  3.42022479e-01  5.84896803e-02  6.66826218e-03
           5.70173899e-04  3.90024616e-05  2.22328640e-06  1.08630559e-07
           4.64426186e-09  1.76493528e-10  6.03647601e-12  1.87691835e-13
           5.34956872e-15  1.40744067e-16  3.43840286e-18  7.84007342e-20
           1.67592618e-21  3.37178999e-23  6.40682210e-25  1.15330363e-26]]
        <NDArray 2x20 @cpu(0)>,
        [3.8980482 5.3267784]
        <NDArray 2 @cpu(0)>)
```

Defining, Training and Testing Model

We first define the plotting function `semilogy`, where the y axis makes use of the logarithmic scale.

```
In [4]: # This function has been saved in the d2l package for future use
        def semilogy(x_vals, y_vals, x_label, y_label, x2_vals=None, y2_vals=None,
                     legend=None, figsize=(3.5, 2.5)):
            d2l.set_figsize(figsize)
            d2l.plt.xlabel(x_label)
            d2l.plt.ylabel(y_label)
            d2l.plt.semilogy(x_vals, y_vals)
            if x2_vals and y2_vals:
                d2l.plt.semilogy(x2_vals, y2_vals, linestyle=':')
                d2l.plt.legend(legend)
```

Similar to linear regression, polynomial function fitting also makes use of a squared loss function. Since we will be attempting to fit the generated data set using models of varying complexity, we insert the model definition into the `fit_and_plot` function. The training and testing steps involved in polynomial function fitting are similar to those previously described in softmax regression.

```
In [5]: num_epochs, loss = 200, gloss.L2Loss()

        def fit_and_plot(train_features, test_features, train_labels, test_labels):
            net = nn.Sequential()
            # Switch off the bias since we already catered for it in the polynomial
            # features
            net.add(nn.Dense(1, use_bias=False))
            net.initialize()
            batch_size = min(10, train_labels.shape[0])
            train_iter = gdata.DataLoader(gdata.ArrayDataset(
                train_features, train_labels), batch_size, shuffle=True)
            trainer = gluon.Trainer(net.collect_params(), 'sgd',
                                    {'learning_rate': 0.01})
            train_ls, test_ls = [], []
            for _ in range(num_epochs):
                for X, y in train_iter:
```

```
        with autograd.record():
            l = loss(net(X), y)
        l.backward()
        trainer.step(batch_size)
    train_ls.append(loss(net(train_features),
                         train_labels).mean().asscalar())
    test_ls.append(loss(net(test_features),
                        test_labels).mean().asscalar())
print('final epoch: train loss', train_ls[-1], 'test loss', test_ls[-1])
semilogy(range(1, num_epochs + 1), train_ls, 'epochs', 'loss',
         range(1, num_epochs + 1), test_ls, ['train', 'test'])
print('weight:', net[0].weight.data().asnumpy())
```

Third-order Polynomial Function Fitting (Normal)

We will begin by first using a third-order polynomial function with the same order as the data generation function. The results show that this model's training error rate when using the testing data set is low. The trained model parameters are also close to the true values $w = [5, 1.2, -3.4, 5.6]$.

```
In [6]: num_epochs = 1000
        # Pick the first four dimensions, i.e. 1, x, x^2, x^3 from the polynomial
        # features
        fit_and_plot(poly_features[:n_train, 0:4], poly_features[n_train:, 0:4],
                     labels[:n_train], labels[n_train:])
```

```
final epoch: train loss 0.0045331274 test loss 0.005139231
weight: [[ 4.995314   1.2209368 -3.3927372  5.5615845]]
```
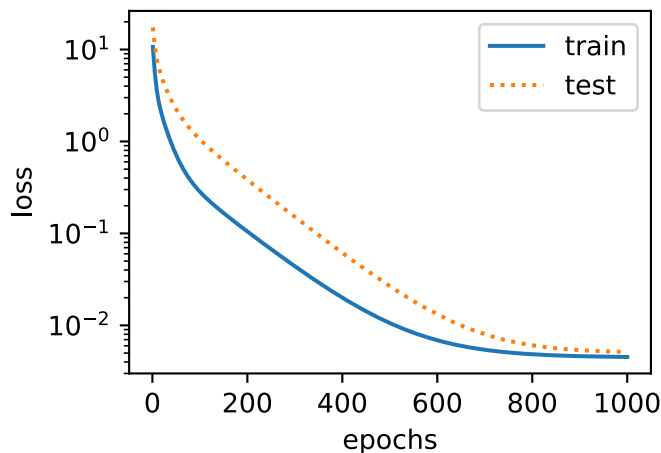

Linear Function Fitting (Underfitting)

Let's take another look at linear function fitting. Naturally, after the decline in the early epoch, it's difficult to further decrease this model's training error rate. After the last epoch iteration has been completed, the

---

training error rate is still high. When used in data sets generated by non-linear models (like the third-order polynomial function) linear models are susceptible to underfitting.

```
In [7]: num_epochs = 1000
        # Pick the first four dimensions, i.e. 1, x from the polynomial features
        fit_and_plot(poly_features[:n_train, 0:3], poly_features[n_train:, 0:3],
                     labels[:n_train], labels[n_train:])
```

```
final epoch: train loss 0.8632311 test loss 3.3488147
weight: [[ 4.927687  3.380442 -2.673047]]
```
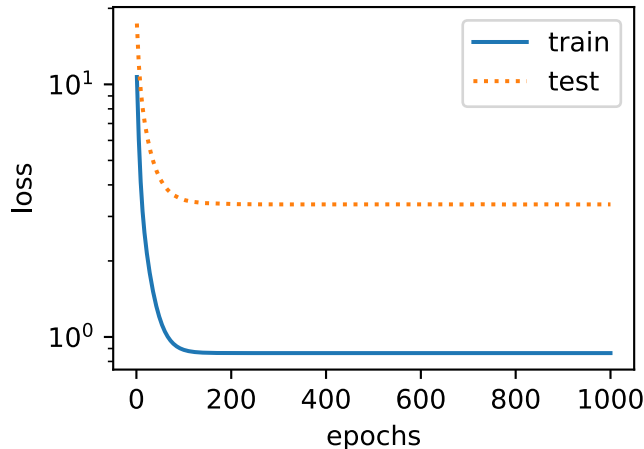

Insufficient Training (Overfitting)

In practice, if the model hasn't been trained sufficiently, it is still easy to overfit even if a third-order polynomial function with the same order as the data generation model is used. Let's train the model using a polynomial of too high degree. There is insufficient data to pin down the fact that all higher degree coefficients are close to zero. This will result in a model that's too complex to be easily influenced by noise in the training data. Even if the training error rate is low, the testing error data rate will still be high.

Try out different model complexities (n_degree) and training set sizes (n_subset) to gain some intuition of what is happening.

```
In [8]: num_epochs = 1000
        n_subset = 100  # Subset of data to train on
        n_degree = 20   # Degree of polynomials
        fit_and_plot(poly_features[1:n_subset, 0:n_degree],
                     poly_features[n_train:, 0:n_degree], labels[1:n_subset],
                     labels[n_train:])
```

```
final epoch: train loss 0.005196244 test loss 0.03174217
weight: [[ 4.9594054e+00  1.2579250e+00 -3.2018733e+00  5.2782574e+00
  -6.3626951e-01  1.3321927e+00 -1.6413799e-02  2.0419168e-01
  -6.1865937e-02  5.8488302e-02 -3.7163243e-02 -6.7995355e-02
```

```
3.7113734e-02 -1.9592566e-02  6.2177803e-02  3.2198679e-02
3.4999892e-02 -4.5971844e-02 -2.2483468e-02  2.9451251e-03]]
```
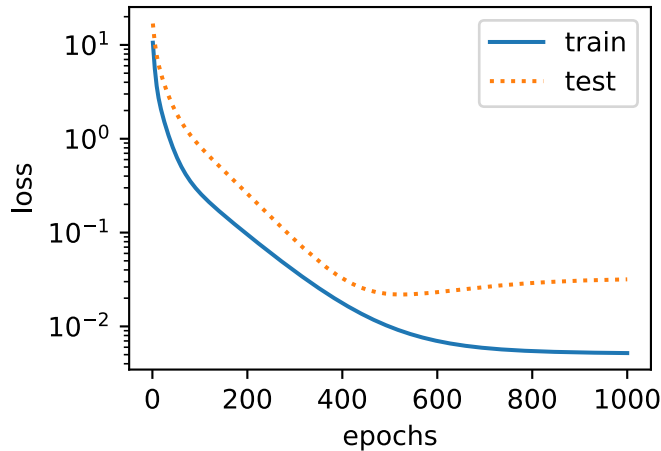

Further along in later chapters, we will continue discussing overfitting problems and methods for dealing with them, such as weight decay and dropout.

Summary

- Since the generalization error rate cannot be estimated based on the training error rate, simply minimizing the training error rate will not necessarily mean a reduction in the generalization error rate. Machine learning models need to be careful to safeguard against overfitting such as to minimize the generalization error.

- A validation set can be used for model selection (provided that it isn't used too liberally).

- Underfitting means that the model is not able to reduce the training error rate while overfitting is a result of the model training error rate being much lower than the testing data set rate.

- We should choose an appropriately complex model and avoid using insufficient training samples.

Exercises

1. Can you solve the polynomial regression problem exactly? Hint - use linear algebra.

2. Model selection for polynomials

   - Plot the training error vs. model complexity (degree of the polynomial). What do you observe?

   - Plot the test error in this case.

- Generate the same graph as a function of the amount of data?

3. What happens if you drop the normalization of the polynomial features x^i by $1/i!$. Can you fix this in some other way?

4. What degree of polynomial do you need to reduce the training error to 0?

5. Can you ever expect to see 0 generalization error?

Scan the QR Code to Discuss

4.5 Weight Decay

In the previous section, we encountered overfitting and the need for capacity control. While increasing the training data set may mitigate overfitting, obtaining additional training data is often costly, hence it is preferable to control the complexity of the functions we use. In particular, we saw that we could control the complexity of a polynomial by adjusting its degree. While this might be a fine strategy for problems on one-dimensional data, this quickly becomes difficult to manage and too coarse. For instance, for vectors of dimensionality D the number of monomials of a given degree d is $\binom{D-1+d}{D-1}$. Hence, instead of controlling for the number of functions we need a more fine-grained tool for adjusting function complexity.

4.5.1 Squared Norm Regularization

One of the most commonly used techniques is weight decay. It relies on the notion that among all functions f the function $f = 0$ is the simplest of all. Hence we can measure functions by their proximity to zero. There are many ways of doing this. In fact there exist entire branches of mathematics, e.g. in functional analysis and the theory of Banach spaces which are devoted to answering this issue.

For our purpose something much simpler will suffice: A linear function $f(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}$ can be considered simple if its weight vector is small. We can measure this via $\|\mathbf{w}\|^2$. One way of keeping the weight vector small is to add its value as a penalty to the problem of minimizing the loss. This way if the weight vector becomes too large, the learning algorithm will prioritize minimizing $\mathbf{w}$ over minimizing the training error. That's exactly what we want. To illustrate things in code, consider the previous section

on Linear Regression. There the loss is given by

$$l(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{2} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right)^2.$$

Recall that $\mathbf{x}^{(i)}$ are the observations, $y^{(i)}$ are labels, and $(\mathbf{w}, b)$ are the weight and bias parameters respectively. To arrive at the new loss function which penalizes the size of the weight vector we need to add $\|\mathbf{w}\|^2$, but how much should we add? This is where the regularization constant (hyperparameter) λ comes in:

$$l(\mathbf{w}, b) + \frac{\lambda}{2} \|\boldsymbol{w}\|^2$$

$\lambda \geq 0$ governs the amount of regularization. For $\lambda = 0$ we recover the previous loss function, whereas for $\lambda > 0$ we ensure that $\mathbf{w}$ cannot grow too large. The astute reader might wonder why we are squaring the weight vector. This is done both for computational convenience since it leads to easy to compute derivatives, and for statistical performance, as it penalizes large weight vectors a lot more than small ones. The stochastic gradient descent updates look as follows:

$$w \leftarrow \left(1 - \frac{\eta \lambda}{|\mathcal{B}|} \right) \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right),$$

As before, we update $\mathbf{w}$ in accordance to the amount to which our estimate differs from the observation. However, we also shrink the size of $\mathbf{w}$ towards 0, i.e. the weight decays'. This is much more convenient than having to pick the number of parameters as we did for polynomials. In particular, we now have a continuous mechanism for adjusting the complexity of f. Small values of λ correspond to fairly unconstrained $\mathbf{w}$ whereas large values of λ constrain $\mathbf{w}$ considerably. Since we don't want to have large bias terms either, we often add b^2 as penalty, too.

4.5.2 High-dimensional Linear Regression

For high-dimensional regression it is difficult to pick the right' dimensions to omit. Weight-decay regularization is a much more convenient alternative. We will illustrate this below. But first we need to generate some data via

$$y = 0.05 + \sum_{i=1}^{d} 0.01 x_i + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.01)$$

That is, we have additive Gaussian noise with zero mean and variance 0.01. In order to observe overfitting more easily we pick a high-dimensional problem with $d = 200$ and a deliberately low number of training examples, e.g. 20. As before we begin with our import ritual (and data generation).

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
```

```
import d2l
from mxnet import autograd, gluon, init, nd
from mxnet.gluon import data as gdata, loss as gloss, nn

n_train, n_test, num_inputs = 20, 100, 200
true_w, true_b = nd.ones((num_inputs, 1)) * 0.01, 0.05

features = nd.random.normal(shape=(n_train + n_test, num_inputs))
labels = nd.dot(features, true_w) + true_b
labels += nd.random.normal(scale=0.01, shape=labels.shape)
train_features, test_features = features[:n_train, :], features[n_train:, :]
train_labels, test_labels = labels[:n_train], labels[n_train:]
```

4.5.3 Implementation from Scratch

Next, we will show how to implement weight decay from scratch. For this we simply add the ℓ_2 penalty as an additional loss term after the target function. The squared norm penalty derives its name from the fact that we are adding the second power $\sum_i x_i^2$. There are many other related penalties. In particular, the ℓ_p norm is defined as

$$\|\mathbf{x}\|_p^p := \sum_{i=1}^d |x_i|^p$$

Initialize Model Parameters

First, define a function that randomly initializes model parameters. This function attaches a gradient to each parameter.

```
In [2]: def init_params():
            w = nd.random.normal(scale=1, shape=(num_inputs, 1))
            b = nd.zeros(shape=(1,))
            w.attach_grad()
            b.attach_grad()
            return [w, b]
```

Define ℓ_2 Norm Penalty

A convenient way of defining this penalty is by squaring all terms in place and summing them up. We divide by 2 to keep the math looking nice and simple.

```
In [3]: def l2_penalty(w):
            return (w**2).sum() / 2
```

Define Training and Testing

The following defines how to train and test the model separately on the training data set and the test data set. Unlike the previous sections, here, the ℓ_2 norm penalty term is added when calculating the final loss function. The linear network and the squared loss are as before and thus imported via `d2l.linreg` and `d2l.squared_loss` respectively.

```
In [4]: batch_size, num_epochs, lr = 1, 100, 0.003
        net, loss = d2l.linreg, d2l.squared_loss
        train_iter = gdata.DataLoader(gdata.ArrayDataset(
            train_features, train_labels), batch_size, shuffle=True)

        def fit_and_plot(lambd):
            w, b = init_params()
            train_ls, test_ls = [], []
            for _ in range(num_epochs):
                for X, y in train_iter:
                    with autograd.record():
                        # The L2 norm penalty term has been added
                        l = loss(net(X, w, b), y) + lambd * l2_penalty(w)
                    l.backward()
                    d2l.sgd([w, b], lr, batch_size)
                train_ls.append(loss(net(train_features, w, b),
                                     train_labels).mean().asscalar())
                test_ls.append(loss(net(test_features, w, b),
                                    test_labels).mean().asscalar())
            d2l.semilogy(range(1, num_epochs + 1), train_ls, 'epochs', 'loss',
                         range(1, num_epochs + 1), test_ls, ['train', 'test'])
            print('l2 norm of w:', w.norm().asscalar())
```

Training without Regularization

Next, let's train and test the high-dimensional linear regression model. When `lambd = 0` we do not use weight decay. As a result, while the training error decreases, the test error does not. This is a perfect example of overfitting.

```
In [5]: fit_and_plot(lambd=0)
```
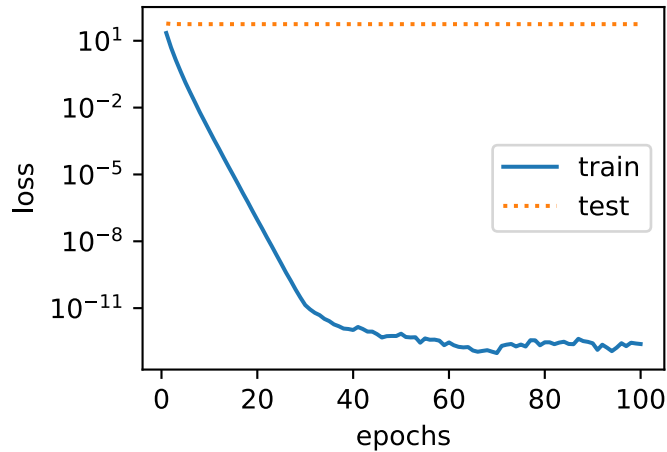
l2 norm of w: 11.61194

Using Weight Decay

The example below shows that even though the training error increased, the error on the test set decreased. This is precisely the improvement that we expect from using weight decay. While not perfect, overfitting has been mitigated to some extent. In addition, the ℓ_2 norm of the weight $\mathbf{w}$ is smaller than without using weight decay.

```
In [6]: fit_and_plot(lambd=3)
```
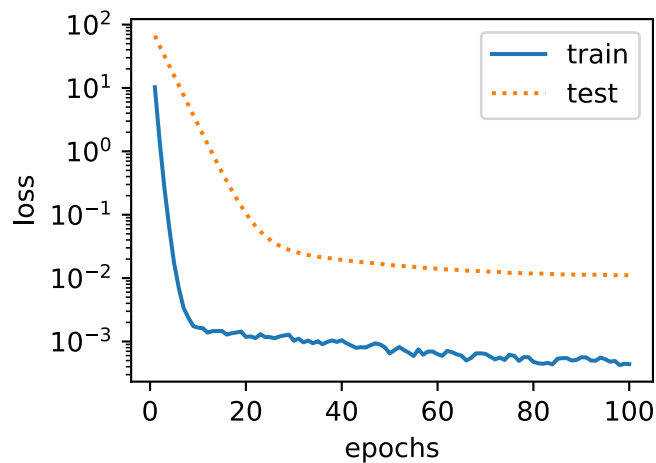

l2 norm of w: 0.041871272

4.5.4 Concise Implementation

Weight decay in Gluon is quite convenient (and also a bit special) insofar as it is typically integrated with the optimization algorithm itself. The reason for this is that it is much faster (in terms of runtime) for the optimizer to take care of weight decay and related things right inside the optimization algorithm itself, since the optimizer itself needs to touch all parameters anyway.

Here, we directly specify the weight decay hyper-parameter through the `wd` parameter when constructing the `Trainer` instance. By default, Gluon decays weight and bias simultaneously. Note that we can have *different* optimizers for different sets of parameters. For instance, we can have a `Trainer` with weight decay and one without to take care of **w** and *b* respectively.

```
In [7]: def fit_and_plot_gluon(wd):
            net = nn.Sequential()
            net.add(nn.Dense(1))
            net.initialize(init.Normal(sigma=1))
            # The weight parameter has been decayed. Weight names generally end with
            # "weight".
            trainer_w = gluon.Trainer(net.collect_params('.*weight'), 'sgd',
                                      {'learning_rate': lr, 'wd': wd})
            # The bias parameter has not decayed. Bias names generally end with "bias"
            trainer_b = gluon.Trainer(net.collect_params('.*bias'), 'sgd',
                                      {'learning_rate': lr})
            train_ls, test_ls = [], []
            for _ in range(num_epochs):
                for X, y in train_iter:
                    with autograd.record():
                        l = loss(net(X), y)
                    l.backward()
                    # Call the step function on each of the two Trainer instances to
                    # update the weight and bias separately
                    trainer_w.step(batch_size)
                    trainer_b.step(batch_size)
                train_ls.append(loss(net(train_features),
                                     train_labels).mean().asscalar())
                test_ls.append(loss(net(test_features),
                                    test_labels).mean().asscalar())
            d2l.semilogy(range(1, num_epochs + 1), train_ls, 'epochs', 'loss',
                         range(1, num_epochs + 1), test_ls, ['train', 'test'])
            print('L2 norm of w:', net[0].weight.data().norm().asscalar())
```

The plots look just the same as when we implemented weight decay from scratch (but they run a bit faster and are a bit easier to implement, in particular for large problems).

```
In [8]: fit_and_plot_gluon(0)
```
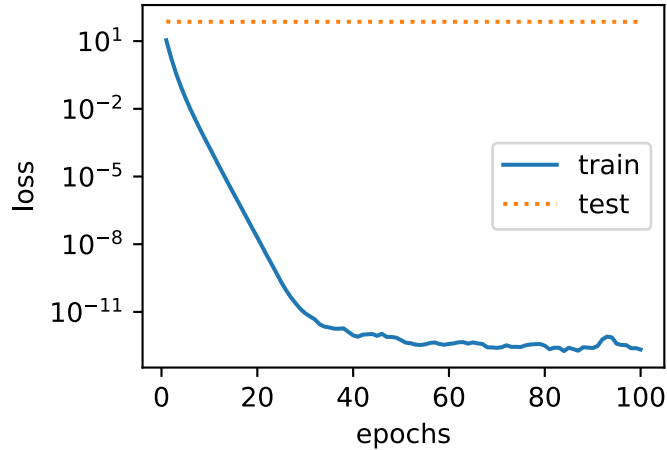
```
L2 norm of w: 13.311797
In [9]: fit_and_plot_gluon(3)
```
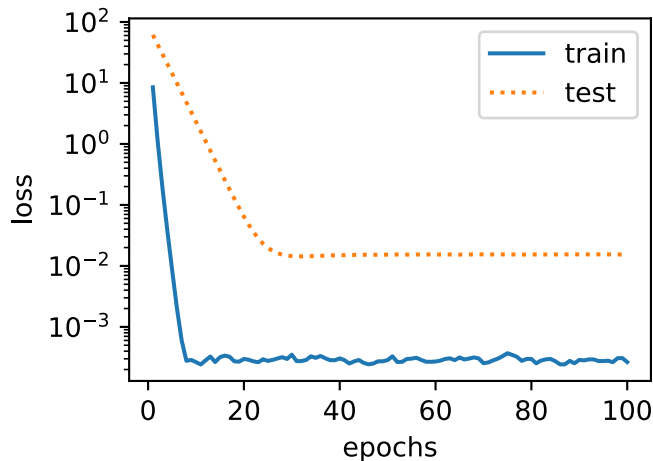


```
L2 norm of w: 0.033165913
```

So far we only touched upon what constitutes a simple *linear* function. For nonlinear functions answering this question is way more complex. For instance, there exist Reproducing Kernel Hilbert Spaces which allow one to use many of the tools introduced for linear functions in a nonlinear context. Unfortunately, algorithms using them do not always scale well to very large amounts of data. For all intents and purposes of this book we limit ourselves to simply summing over the weights for different layers, e.g. via $\sum_l \|\mathbf{w}_l\|^2$, which is equivalent to weight decay applied to all layers.

Summary

- Regularization is a common method for dealing with overfitting. It adds a penalty term to the loss function on the training set to reduce the complexity of the learned model.

- One particular choice for keeping the model simple is weight decay using an ℓ_2 penalty. This leads to weight decay in the update steps of the learning algorithm.

- Gluon provides automatic weight decay functionality in the optimizer by setting the hyperparameter `wd`.

- You can have different optimizers within the same training loop, e.g. for different sets of parameters.

Exercises

1. Experiment with the value of λ in the estimation problem in this page. Plot training and test accuracy as a function of λ. What do you observe?

2. Use a validation set to find the optimal value of λ. Is it really the optimal value? Does this matter?

3. What would the update equations look like if instead of $\|\mathbf{w}\|^2$ we used $\sum_i |w_i|$ as our penalty of choice (this is called ℓ_1 regularization).

4. We know that $\|\mathbf{w}\|^2 = \mathbf{w}^\top \mathbf{w}$. Can you find a similar equation for matrices (mathematicians call this the Frobenius norm)?

5. Review the relationship between training error and generalization error. In addition to weight decay, increased training, and the use of a model of suitable complexity, what other ways can you think of to deal with overfitting?

6. In Bayesian statistics we use the product of prior and likelihood to arrive at a posterior via $p(w|x) \propto p(x|w)p(w)$. How can you identify $p(w)$ with regularization?

Scan the QR Code to Discuss

4.6 Dropout

In the previous chapter, we introduced one classical approach to regularize statistical models. We penalized the size (the ℓ_2 norm) of the weights, coercing them to take smaller values. In probabilistic terms we might say that this imposes a Gaussian prior on the value of the weights. But in more intuitive, functional terms, we can say that this encourages the model to spread out its weights among many features and not to depend too much on a small number of potentially spurious associations.

4.6.1 Overfitting Revisited

With great flexibility comes overfitting liability. Given many more features than examples, linear models can overfit. But when there are many more examples than features, linear models can usually be counted on not to overfit. Unfortunately this propensity to generalize well comes at a cost. For every feature, a linear model has to assign it either positive or negative weight. Linear models can't take into account nuanced interactions between features. In more formal texts, you'll see this phenomena discussed as the bias-variance tradeoff. Linear models have high bias, (they can only represent a small class of functions), but low variance (they give similar results across different random samples of the data).

Deep neural networks, however, occupy the opposite end of the bias-variance spectrum. Neural networks are so flexible because they aren't confined to looking at each feature individually. Instead, they can learn complex interactions among groups of features. For example, they might infer that Nigeria and Western Union appearing together in an email indicates spam but that Nigeria without Western Union does not connote spam.

Even for a small number of features, deep neural networks are capable of overfitting. As one demonstration of the incredible flexibility of neural networks, researchers showed that neural networks perfectly classify randomly labeled data. Let's think about what means. If the labels are assigned uniformly at random, and there are 10 classes, then no classifier can get better than 10% accuracy on holdout data. Yet even in these situations, when there is no true pattern to be learned, neural networks can perfectly fit the training labels.

4.6.2 Robustness through Perturbations

Let's think briefly about what we expect from a good statistical model. Obviously we want it to do well on unseen test data. One way we can accomplish this is by asking for what amounts to a simple' model. Simplicity can come in the form of a small number of dimensions, which is what we did when discussing fitting a function with monomial basis functions. Simplicity can also come in the form of a small norm for the basis funtions. This is what led to weight decay and ℓ_2 regularization. Yet a third way to impose some notion of simplicity is that the function should be robust under modest changes in the input. For instance, when we classify images, we would expect that alterations of a few pixels are mostly harmless.

In fact, this notion was formalized by Bishop in 1995, when he proved that Training with Input Noise is Equivalent to Tikhonov Regularization. That is, he connected the notion of having a smooth (and thus simple) function with one that is resilient to perturbations in the input. Fast forward to 2014. Given

the complexity of deep networks with many layers, enforcing smoothness just on the input misses out on what is happening in subsequent layers. The ingenious idea of Srivastava et al., 2014 was to apply Bishop's idea to the *internal* layers of the network, too, namely to inject noise into the computational path of the network while it's training.

A key challenge in this context is how to add noise without introducing undue bias. In terms of inputs $\mathbf{x}$, this is relatively easy to accomplish: simply add some noise $\epsilon \sim \mathcal{N}(0, \sigma^2)$ to it and use this data during training via $\mathbf{x}' = \mathbf{x} + \epsilon$. A key property is that in expectation $\mathbf{E}[\mathbf{x}'] = \mathbf{x}$. For intermediate layers, though, this might not be quite so desirable since the scale of the noise might not be appropriate. The alternative is to perturb coordinates as follows:

$$
h' = \begin{cases} 0 & \text{with probability } p \\ \frac{h}{1-p} & \text{otherwise} \end{cases}
$$

By design, the expectation remains unchanged, i.e. $\mathbf{E}[h'] = h$. This idea is at the heart of dropout where intermediate activations h are replaced by a random variable h' with matching expectation. The name dropout' arises from the notion that some neurons drop out' of the computation for the purpose of computing the final result. During training we replace intermediate activations with random variables

4.6.3 Dropout in Practice

Recall the *multilayer perceptron* with a hidden layer and 5 hidden units. Its architecture is given by

$$
\begin{aligned}
h &= \sigma(W_1 x + b_1) \\
o &= W_2 h + b_2 \\
\hat{y} &= \text{softmax}(o)
\end{aligned}
$$

When we apply dropout to the hidden layer, it amounts to removing hidden units with probability p since their output is set to 0 with that probability. A possible result is the network shown below. Here h_2 and h_5 are removed. Consequently the calculation of y no longer depends on h_2 and h_5 and their respective gradient also vanishes when performing backprop. In this way, the calculation of the output layer cannot be overly dependent on any one element of $h_1, \dots, h_5$. This is exactly what we want for regularization purposes to cope with overfitting. At test time we typically do not use dropout to obtain more conclusive results.
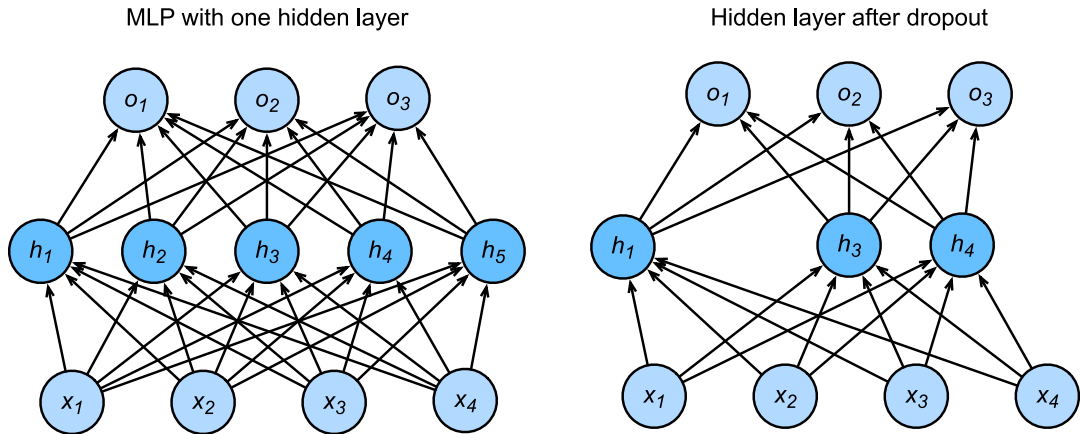
Fig. 4.4: MLP before and after dropout

4.6.4 Implementation from Scratch

To implement the dropout function we have to draw as many random variables as the input has dimensions from the uniform distribution $U[0, 1]$. According to the definition of dropout, we can implement it easily. The following `dropout` function will drop out the elements in the NDArray input X with the probability of `drop_prob`.

```
In [1]: import sys
        sys.path.insert(0, '..')

        import d2l
        from mxnet import autograd, gluon, init, nd
        from mxnet.gluon import loss as gloss, nn

        def dropout(X, drop_prob):
            assert 0 <= drop_prob <= 1
            # In this case, all elements are dropped out
            if drop_prob == 1:
                return X.zeros_like()
            mask = nd.random.uniform(0, 1, X.shape) > drop_prob
            return mask * X / (1.0-drop_prob)
```

Let us test how it works in a few examples. The dropout probability is 0, 0.5, and 1, respectively.

```
In [2]: X = nd.arange(16).reshape((2, 8))
        print(dropout(X, 0))
        print(dropout(X, 0.5))
        print(dropout(X, 1))

[[ 0.  1.  2.  3.  4.  5.  6.  7.]
 [ 8.  9. 10. 11. 12. 13. 14. 15.]]
<NDArray 2x8 @cpu(0)>
```

```
[[ 0.   0.   0.   0.   8. 10. 12.   0.]
 [16.   0. 20. 22.   0.   0.   0. 30.]]
<NDArray 2x8 @cpu(0)>

[[0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]
<NDArray 2x8 @cpu(0)>
```

Defining Model Parameters

Let's use the same dataset as used previously, namely Fashion-MNIST, described in the section Softmax Regression - Starting From Scratch. We will define a multilayer perceptron with two hidden layers. The two hidden layers both have 256 outputs.

```
In [3]: num_inputs, num_outputs, num_hiddens1, num_hiddens2 = 784, 10, 256, 256

        W1 = nd.random.normal(scale=0.01, shape=(num_inputs, num_hiddens1))
        b1 = nd.zeros(num_hiddens1)
        W2 = nd.random.normal(scale=0.01, shape=(num_hiddens1, num_hiddens2))
        b2 = nd.zeros(num_hiddens2)
        W3 = nd.random.normal(scale=0.01, shape=(num_hiddens2, num_outputs))
        b3 = nd.zeros(num_outputs)

        params = [W1, b1, W2, b2, W3, b3]
        for param in params:
            param.attach_grad()
```

Define the Model

The model defined below concatenates the fully connected layer and the activation function ReLU, using dropout for the output of each activation function. We can set the dropout probability of each layer separately. It is generally recommended to set a lower dropout probability closer to the input layer. Below we set it to 0.2 and 0.5 for the first and second hidden layer respectively. By using the is_training function described in the Autograd section we can ensure that dropout is only active during training.

```
In [4]: drop_prob1, drop_prob2 = 0.2, 0.5

        def net(X):
            X = X.reshape((-1, num_inputs))
            H1 = (nd.dot(X, W1) + b1).relu()
            # Use dropout only when training the model
            if autograd.is_training():
                # Add a dropout layer after the first fully connected layer
                H1 = dropout(H1, drop_prob1)
            H2 = (nd.dot(H1, W2) + b2).relu()
            if autograd.is_training():
                # Add a dropout layer after the second fully connected layer
                H2 = dropout(H2, drop_prob2)
            return nd.dot(H2, W3) + b3
```

Training and Testing

This is similar to the training and testing of multilayer perceptrons described previously.

```
In [5]: num_epochs, lr, batch_size = 10, 0.5, 256
        loss = gloss.SoftmaxCrossEntropyLoss()
        train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
        d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size,
                      params, lr)

epoch 1, loss 1.1241, train acc 0.562, test acc 0.791
epoch 2, loss 0.5772, train acc 0.786, test acc 0.789
epoch 3, loss 0.4924, train acc 0.820, test acc 0.844
epoch 4, loss 0.4475, train acc 0.837, test acc 0.843
epoch 5, loss 0.4207, train acc 0.847, test acc 0.858
epoch 6, loss 0.3939, train acc 0.856, test acc 0.863
epoch 7, loss 0.3773, train acc 0.861, test acc 0.873
epoch 8, loss 0.3610, train acc 0.867, test acc 0.877
epoch 9, loss 0.3553, train acc 0.870, test acc 0.881
epoch 10, loss 0.3444, train acc 0.874, test acc 0.869
```

4.6.5 Concise Implementation

In Gluon, we only need to add the `Dropout` layer after the fully connected layer and specify the dropout probability. When training the model, the `Dropout` layer will randomly drop out the output elements of the previous layer at the specified dropout probability; the `Dropout` layer simply passes the data through during testing.

```
In [6]: net = nn.Sequential()
        net.add(nn.Dense(256, activation="relu"),
                # Add a dropout layer after the first fully connected layer
                nn.Dropout(drop_prob1),
                nn.Dense(256, activation="relu"),
                # Add a dropout layer after the second fully connected layer
                nn.Dropout(drop_prob2),
                nn.Dense(10))
        net.initialize(init.Normal(sigma=0.01))
```

Next, we will train and test the model.

```
In [7]: trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
        d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size, None,
                      None, trainer)

epoch 1, loss 1.2010, train acc 0.544, test acc 0.750
epoch 2, loss 0.5851, train acc 0.783, test acc 0.813
epoch 3, loss 0.5015, train acc 0.817, test acc 0.856
epoch 4, loss 0.4511, train acc 0.835, test acc 0.861
epoch 5, loss 0.4217, train acc 0.846, test acc 0.855
epoch 6, loss 0.3997, train acc 0.855, test acc 0.858
epoch 7, loss 0.3817, train acc 0.860, test acc 0.877
epoch 8, loss 0.3719, train acc 0.865, test acc 0.880
epoch 9, loss 0.3589, train acc 0.869, test acc 0.868
epoch 10, loss 0.3455, train acc 0.874, test acc 0.879
```

Summary

- Beyond controlling the number of dimensions and the size of the weight vector, dropout is yet another tool to avoid overfitting. Often all three are used jointly.

- Dropout replaces an activation h with a random variable h' with expected value h and with variance given by the dropout probability p.

- Dropout is only used during training.

Exercises

1. Try out what happens if you change the dropout probabilities for layers 1 and 2. In particular, what happens if you switch the ones for both layers?

2. Increase the number of epochs and compare the results obtained when using dropout with those when not using it.

3. Compute the variance of the the activation random variables after applying dropout.

4. Why should you typically not using dropout?

5. If changes are made to the model to make it more complex, such as adding hidden layer units, will the effect of using dropout to cope with overfitting be more obvious?

6. Using the model in this section as an example, compare the effects of using dropout and weight decay. What if dropout and weight decay are used at the same time?

7. What happens if we apply dropout to the individual weights of the weight matrix rather than the activations?

8. Replace the dropout activation with a random variable that takes on values of $[0, \gamma/2, \gamma]$. Can you design something that works better than the binary dropout function? Why might you want to use it? Why not?

References

[1] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). JMLR

Scan the QR Code to Discuss

4.7 Forward Propagation, Back Propagation, and Computational Graphs

In the previous sections we used a mini-batch stochastic gradient descent optimization algorithm to train the model. During the implementation of the algorithm, we only calculated the forward propagation of the model, which is to say, we calculated the model output for the input, then called the auto-generated `backward` function to then finally calculate the gradient through the `autograd` module. The automatic gradient calculation, when based on back-propagation, significantly simplifies the implementation of the deep learning model training algorithm. In this section we will use both mathematical and computational graphs to describe forward and back propagation. More specifically, we will explain forward and back propagation through a sample model with a single hidden layer perceptron with ℓ_2 norm regularization. This section will help understand a bit better what goes on behind the scenes when we invoke a deep network.

4.7.1 Forward Propagation

Forward propagation refers to the calculation and storage of intermediate variables (including outputs) for the neural network within the models in the order from input layer to output layer. In the following we work in detail through the example of a deep network with one hidden layer step by step. This is a bit tedious but it will serve us well when discussing what really goes on when we call `backward`.

For the sake of simplicity, let's assume that the input example is $\mathbf{x} \in \mathbb{R}^d$ and there is no bias term. Here the intermediate variable is:

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}$$

$\mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ is the weight parameter of the hidden layer. After entering the intermediate variable $\mathbf{z} \in \mathbb{R}^h$ into the activation function ϕ operated by the basic elements, we will obtain a hidden layer variable with the vector length of h,

$$\mathbf{h} = \phi(\mathbf{z}).$$

The hidden variable $\mathbf{h}$ is also an intermediate variable. Assuming the parameters of the output layer only possess a weight of $\mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$, we can obtain an output layer variable with a vector length of q,

$$\mathbf{o} = \mathbf{W}^{(2)}\mathbf{h}.$$

Assuming the loss function is l and the example label is y, we can then calculate the loss term for a single data example,

$$L = l(\mathbf{o}, y).$$

According to the definition of ℓ_2 norm regularization, given the hyper-parameter λ, the regularization

term is

$$s = \frac{\lambda}{2} \left(\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2 \right),$$

where the Frobenius norm of the matrix is equivalent to the calculation of the L_2 norm after flattening the matrix to a vector. Finally, the model's regularized loss on a given data example is

$$J = L + s.$$

We refer to J as the objective function of a given data example and refer to it as the objective function' in the following discussion.

4.7.2 Computational Graph of Forward Propagation

Plotting computational graphs helps us visualize the dependencies of operators and variables within the calculation. The figure below contains the graph associated with the simple network described above. The lower left corner signifies the input and the upper right corner the output. Notice that the direction of the arrows (which illustrate data flow) are primarily rightward and upward.
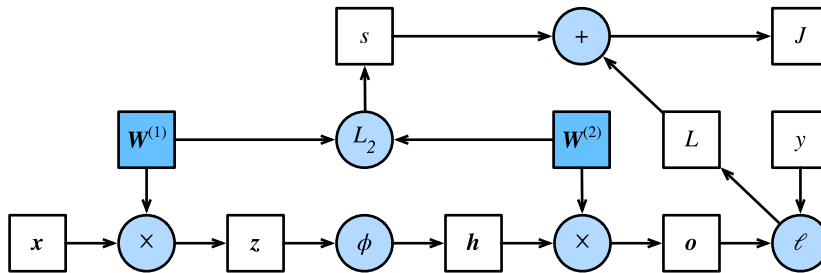
Fig. 4.5: Computational Graph

4.7.3 Back Propagation

Back propagation refers to the method of calculating the gradient of neural network parameters. In general, back propagation calculates and stores the intermediate variables of an objective function related to each layer of the neural network and the gradient of the parameters in the order of the output layer to the input layer according to the chain rule' in calculus. Assume that we have functions $\mathsf{Y} = f(\mathsf{X})$ and $\mathsf{Z} = g(\mathsf{Y}) = g \circ f(\mathsf{X})$, in which the input and the output $\mathsf{X}, \mathsf{Y}, \mathsf{Z}$ are tensors of arbitrary shapes. By using the chain rule we can compute the derivative of Z wrt. X via

$$\frac{\partial \mathsf{Z}}{\partial \mathsf{X}} = \mathrm{prod}\left(\frac{\partial \mathsf{Z}}{\partial \mathsf{Y}}, \frac{\partial \mathsf{Y}}{\partial \mathsf{X}} \right).$$

Here we use the prod operator to multiply its arguments after the necessary operations, such as transposition and swapping input positions have been carried out. For vectors this is straightforward: it is simply matrix-matrix multiplication and for higher dimensional tensors we use the appropriate counterpart. The operator prod hides all the notation overhead.

The parameters of the simple network with one hidden layer are $\mathbf{W}^{(1)}$ and $\mathbf{W}^{(2)}$. The objective of back propagation is to calculate the gradients $\partial J/\partial \mathbf{W}^{(1)}$ and $\partial J/\partial \mathbf{W}^{(2)}$. To accompish this we will apply the chain rule and calculate in turn the gradient of each intermediate variable and parameter. The order of calculations are reversed relative to those performed in forward propagation, since we need to start with the outcome of the compute graph and work our way towards the parameters. The first step is to calculate the gradients of the objective function $J = L + s$ with respect to the loss term L and the regularization term s.

$$\frac{\partial J}{\partial L} = 1 \text{ and } \frac{\partial J}{\partial s} = 1$$

Next we compute the gradient of the objective function with respect to variable of the output layer $\mathbf{o}$ according to the chain rule.

$$\frac{\partial J}{\partial \mathbf{o}} = \text{prod}\left(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}}\right) = \frac{\partial L}{\partial \mathbf{o}} \in \mathbb{R}^q$$

Next we calculate the gradients of the regularization term with respect to both parameters.

$$\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)} \text{ and } \frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}$$

Now we are able calculate the gradient $\partial J/\partial \mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$ of the model parameters closest to the output layer. Using the chain rule yields:

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}}\right) = \frac{\partial J}{\partial \mathbf{o}}\mathbf{h}^\top + \lambda \mathbf{W}^{(2)}$$

To obtain the gradient with respect to $\mathbf{W}^{(1)}$ we need to continue back propagation along the output layer to the hidden layer. The gradient $\partial J/\partial \mathbf{h} \in \mathbb{R}^h$ of the hidden layer variable is

$$\frac{\partial J}{\partial \mathbf{h}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}}\right) = \mathbf{W}^{(2)\top}\frac{\partial J}{\partial \mathbf{o}}.$$

Since the activation function ϕ applies element-wise, calculating the gradient $\partial J/\partial \mathbf{z} \in \mathbb{R}^h$ of the intermediate variable $\mathbf{z}$ requires the use of the element-wise multiplication operator. We denote it by $\odot$.

$$\frac{\partial J}{\partial \mathbf{z}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}}\right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'\left(\mathbf{z}\right).$$

Finally, we can obtain the gradient $\partial J/\partial \mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ of the model parameters closest to the input layer.

According to the chain rule, we get

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}}\right) = \frac{\partial J}{\partial \mathbf{z}}\mathbf{x}^\top + \lambda\mathbf{W}^{(1)}.$$

4.7.4 Training a Model

When training networks, forward and backward propagation depend on each other. In particular, for forward propagation we traverse the compute graph in the direction of dependencies and compute all the variables on its path. These are then used for backpropagation where the compute order on the graph is reversed. One of the consequences is that we need to retain the intermediate values until backpropagation is complete. This is also one of the reasons why backpropagation requires significantly more memory than plain inference' - we end up computing tensors as gradients and need to retain all the intermediate variables to invoke the chain rule. Another reason is that we typically train with minibatches containing more than one variable, thus more intermediate activations need to be stored.

Summary

- Forward propagation sequentially calculates and stores intermediate variables within the compute graph defined by the neural network. It proceeds from input to output layer.
- Back propagation sequentially calculates and stores the gradients of intermediate variables and parameters within the neural network in the reversed order.
- When training deep learning models, forward propagation and back propagation are interdependent.
- Training requires significantly more memory and storage.

Exercises

1. Assume that the inputs $\mathbf{x}$ are matrices. What is the dimensionality of the gradients?
2. Add a bias to the hidden layer of the model described in this chapter.
   - Draw the corresponding compute graph.
   - Derive the forward and backward propagation equations.
3. Compute the memory footprint for training and inference in model described in the current chapter.
4. Assume that you want to compute *second* derivatives. What happens to the compute graph? Is this a good idea?
5. Assume that the compute graph is too large for your GPU.
   - Can you partition it over more than one GPU?

- What are the advantages and disadvantages over training on a smaller minibatch?

Scan the QR Code to Discuss

4.8 Numerical Stability and Initialization

So far we covered the tools needed to implement multilayer perceptrons, how to solve regression and classification problems, and how to control capacity. However, we took initialization of the parameters for granted, or rather simply assumed that they would not be particularly relevant. In the following we will look at them in more detail and discuss some useful heuristics.

Secondly, we were not particularly concerned with the choice of activation. Indeed, for shallow networks this is not very relevant. For deep networks, however, design choices of nonlinearity and initialization play a crucial role in making the optimization algorithm converge relatively rapidly. Failure to be mindful of these issues can lead to either exploding or vanishing gradients.

4.8.1 Vanishing and Exploding Gradients

Consider a deep network with d layers, input $\mathbf{x}$ and output $\mathbf{o}$. Each layer satisfies:

$$\mathbf{h}^{t+1} = f_t(\mathbf{h}^t) \text{ and thus } \mathbf{o} = f_d \circ \ldots \circ f_1(\mathbf{x})$$

If all activations and inputs are vectors, we can write the gradient of $\mathbf{o}$ with respect to any set of parameters $\mathbf{W}_t$ associated with the function f_t at layer t simply as

$$\partial_{\mathbf{W}_t} \mathbf{o} = \underbrace{\partial_{\mathbf{h}^{d-1}} \mathbf{h}^d}_{:=\mathbf{M}_d} \cdot \ldots \cdot \underbrace{\partial_{\mathbf{h}^t} \mathbf{h}^{t+1}}_{:=\mathbf{M}_t} \underbrace{\partial_{\mathbf{W}_t} \mathbf{h}^t}_{:=\mathbf{v}_t}.$$

In other words, it is the product of $d - t$ matrices $\mathbf{M}_d \cdot \ldots \cdot \mathbf{M}_t$ and the gradient vector $\mathbf{v}_t$. What happens is quite similar to the situation when we experienced numerical underflow when multiplying too many probabilities. At the time we were able to mitigate the problem by switching from into log-space, i.e. by shifting the problem from the mantissa to the exponent of the numerical representation. Unfortunately the problem outlined in the equation above is much more serious: initially the matrices M_t may well have a wide variety of eigenvalues. They might be small, they might be large, and in particular, their product might well be *very large* or *very small*. This is not (only) a problem of numerical representation but it means that the optimization algorithm is bound to fail. It either receives gradients with excessively large

or excessively small steps. In the former case, the parameters explode and in the latter case we end up with vanishing gradients and no meaningful progress.

Exploding Gradients

To illustrate this a bit better, we draw 100 Gaussian random matrices and multiply them with some initial matrix. For the scaling that we picked, the matrix product explodes. If this were to happen to us with a deep network, we would have no meaningful chance of making the algorithm converge.

```
In [1]: %matplotlib inline
        import mxnet as mx
        from mxnet import nd, autograd
        from matplotlib import pyplot as plt

        M = nd.random.normal(shape=(4,4))
        print('A single matrix', M)
        for i in range(100):
            M = nd.dot(M, nd.random.normal(shape=(4,4)))

        print('After multiplying 100 matrices', M)

A single matrix
[[ 2.2122064   0.7740038   1.0434405   1.1839255 ]
 [ 1.8917114  -1.2347414  -1.771029   -0.45138445]
 [ 0.57938355 -1.856082   -1.9768796  -0.20801921]
 [ 0.2444218  -0.03716067 -0.48774993 -0.02261727]]
<NDArray 4x4 @cpu(0)>
After multiplying 100 matrices
[[ 3.1575275e+20 -5.0052276e+19  2.0565092e+21 -2.3741922e+20]
 [-4.6332600e+20  7.3445046e+19 -3.0176513e+21  3.4838066e+20]
 [-5.8487235e+20  9.2711797e+19 -3.8092853e+21  4.3977330e+20]
 [-6.2947415e+19  9.9783660e+18 -4.0997977e+20  4.7331174e+19]]
<NDArray 4x4 @cpu(0)>
```

Vanishing Gradients

The converse problem of vanishing gradients is just as bad. One of the major culprits in this context are the activation functions σ that are interleaved with the linear operations in each layer. Historically, a popular activation used to be the sigmoid function $(1 + \exp(-x))$ that was introduced in the section discussing Multilayer Perceptrons. Let us briefly review the function to see why picking it as nonlinear activation function might be problematic.

```
In [2]: x = nd.arange(-8.0, 8.0, 0.1)
        x.attach_grad()
        with autograd.record():
            y = x.sigmoid()
        y.backward()

        plt.figure(figsize=(8, 4))
        plt.plot(x.asnumpy(), y.asnumpy())
        plt.plot(x.asnumpy(), x.grad.asnumpy())
```

```
plt.legend(['sigmoid', 'gradient'])
plt.show()
```
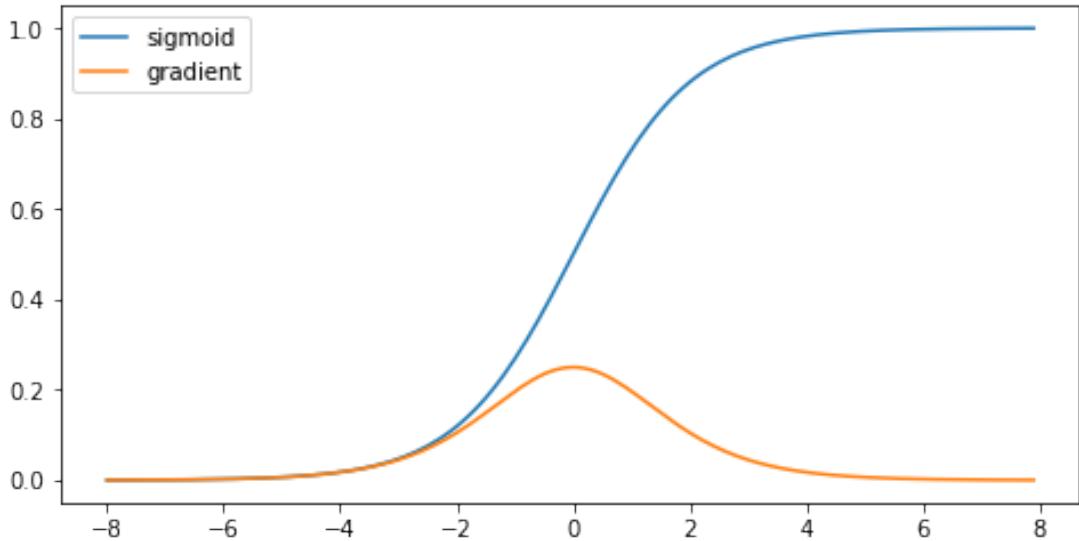

As we can see, the gradient of the sigmoid vanishes for very large or very small arguments. Due to the chain rule, this means that unless we are in the Goldilocks zone where the activations are in the range of, say $[-4, 4]$, the gradients of the overall product may vanish. When we have many layers this is likely to happen for *some* layer. Before ReLu $\max(0, x)$ was proposed, this problem used to be the bane of deep network training. As a consequence ReLu has become the default choice when designing activation functions in deep networks.

Symmetry

A last problem in deep network design is the symmetry inherent in their parametrization. Assume that we have a deep network with one hidden layer with two units, say h_1 and h_2. In this case, we could flip the weights $\mathbf{W}_1$ of the first layer and likewise the outputs of the second layer and we would obtain the same function. More generally, we have permutation symmetry between the hidden units of each layer. This is more than just a theoretical nuisance. Assume that we initialize the parameters of some layer as $\mathbf{W}_l = 0$ or even just assume that all entries of $\mathbf{W}_l$ are identical. In this case the gradients for all dimensions are identical and we will never be able to use the expressive power inherent in a given layer. In fact, the hidden layer behaves as if it had only a single unit.

4.8.2 Parameter Initialization

One way of addressing, or at least mitigating the issues raised above is through careful initialization of the weight vectors. This way we can ensure that at least initially the gradients do not vanish and that they are

within a reasonable scale where the network weights do not diverge. Additional care during optimization and suitable regularization ensures that things never get too bad. Let's get started.

Default Initialization

In the previous sections, e.g. in Concise Implementation of Linear Regression, we used `net.initialize(init.Normal(sigma=0.01))` as a way to pick normally distributed random numbers as initial values for the weights. If the initialization method is not specified, such as `net.initialize()`, MXNet will use the default random initialization method: each element of the weight parameter is randomly sampled with an uniform distribution $U[-0.07, 0.07]$ and the bias parameters are all set to 0. Both choices tend to work quite well in practice for moderate problem sizes.

Xavier Initialization

Let's look at the scale distribution of the activations of the hidden units h_i for some layer. They are given by

$$h_i = \sum_{j=1}^{n_{\text{in}}} W_{ij} x_j$$

The weights W_{ij} are all drawn independently from the same distribution. Let's furthermore assume that this distribution has zero mean and variance σ^2 (this doesn't mean that the distribution has to be Gaussian, just that mean and variance need to exist). We don't really have much control over the inputs into the layer x_j but let's proceed with the somewhat unrealistic assumption that they also have zero mean and variance γ^2 and that they're independent of $\mathbf{W}$. In this case we can compute mean and variance of h_i as follows:

$$\mathbf{E}[h_i] = \sum_{j=1}^{n_{\text{in}}} \mathbf{E}[W_{ij} x_j] = 0$$

$$\mathbf{E}[h_i^2] = \sum_{j=1}^{n_{\text{in}}} \mathbf{E}[W_{ij}^2 x_j^2]$$

$$= \sum_{j=1}^{n_{\text{in}}} \mathbf{E}[W_{ij}^2] \mathbf{E}[x_j^2]$$

$$= n_{\text{in}} \sigma^2 \gamma^2$$

One way to keep the variance fixed is to set $n_{\text{in}} \sigma^2 = 1$. Now consider backpropagation. There we face a similar problem, albeit with gradients being propagated from the top layers. That is, instead of $\mathbf{Ww}$ we need to deal with $\mathbf{W}^\top \mathbf{g}$, where $\mathbf{g}$ is the incoming gradient from the layer above. Using the same reasoning as for forward propagation we see that the gradients' variance can blow up unless $n_{\text{out}} \sigma^2 = 1$. This leaves us in a dilemma: we cannot possibly satisfy both conditions simultaneously. Instead, we

simply try to satisfy

$$\frac{1}{2}(n_{\text{in}} + n_{\text{out}})\sigma^2 = 1 \text{ or equivalently } \sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}$$

This is the reasoning underlying the eponymous Xavier initialization, proposed by Xavier Glorot and Yoshua Bengio in 2010. It works well enough in practice. For Gaussian random variables the Xavier initialization picks a normal distribution with zero mean and variance $\sigma^2 = 2/(n_{\text{in}} + n_{\text{out}})$. For uniformly distributed random variables $U[-a, a]$ note that their variance is given by $a^2/3$. Plugging $a^2/3$ into the condition on σ^2 yields that we should initialize uniformly with $U\left[-\sqrt{6/(n_{\text{in}} + n_{\text{out}})}, \sqrt{6/(n_{\text{in}} + n_{\text{out}})}\right]$.

Beyond

The reasoning above barely scratches the surface. In fact, MXNet has an entire `mxnet.initializer` module with over a dozen different heuristics. They can be used, e.g. when parameters are tied (i.e. when parameters of in different parts the network are shared), for superresolution, sequence models, and related problems. We recommend the reader to review what is offered as part of this module.

Summary

- Vanishing and exploding gradients are common issues in very deep networks, unless great care is taking to ensure that gradients and parameters remain well controlled.

- Initialization heuristics are needed to ensure that at least the initial gradients are neither too large nor too small.

- The ReLu addresses one of the vanishing gradient problems, namely that gradients vanish for very large inputs. This can accelerate convergence significantly.

- Random initialization is key to ensure that symmetry is broken before optimization.

Exercises

1. Can you design other cases of symmetry breaking besides the permutation symmetry?

2. Can we initialize all weight parameters in linear regression or in softmax regression to the same value?

3. Look up analytic bounds on the eigenvalues of the product of two matrices. What does this tell you about ensuring that gradients are well conditioned?

4. If we know that some terms diverge, can we fix this after the fact? Look at the paper on LARS by You, Gitman and Ginsburg, 2017 for inspiration.

4.9 Environment
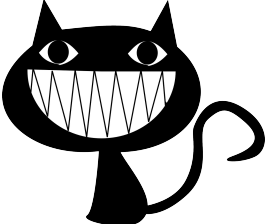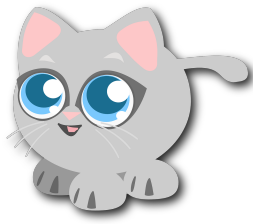
So far we did not worry very much about where the data came from and how the models that we build get deployed. Not caring about it can be problematic. Many failed machine learning deployments can be traced back to this situation. This chapter is meant to help with detecting such situations early and points out how to mitigate them. Depending on the case this might be rather simple (ask for the right' data) or really difficult (implement a reinforcement learning system).

4.9.1 Covariate Shift

At its heart is a problem that is easy to understand but also equally easy to miss. Consider being given the challenge of distinguishing cats and dogs. Our training data consists of images of the following kind:

cat	cat	dog	dog

At test time we are asked to classify the following images:

cat	cat	dog	dog

Obviously this is unlikely to work well. The training set consists of photos, while the test set contains only cartoons. The colors aren't even accurate. Training on a dataset that looks substantially different from the test set without some plan for how to adapt to the new domain is a bad idea. Unfortunately, this is a very common pitfall. Statisticians call this **Covariate Shift**, i.e. the situation where the distribution over the covariates (aka training data) is shifted on test data relative to the training case. Mathematically speaking, we are referring the case where $p(x)$ changes but $p(y|x)$ remains unchanged.

4.9.2 Concept Shift

A related problem is that of concept shift. This is the situation where the labels change. This sounds weird - after all, a cat is a cat. Well, cats maybe but not soft drinks. There is considerable concept shift throughout the USA, even for such a simple term:
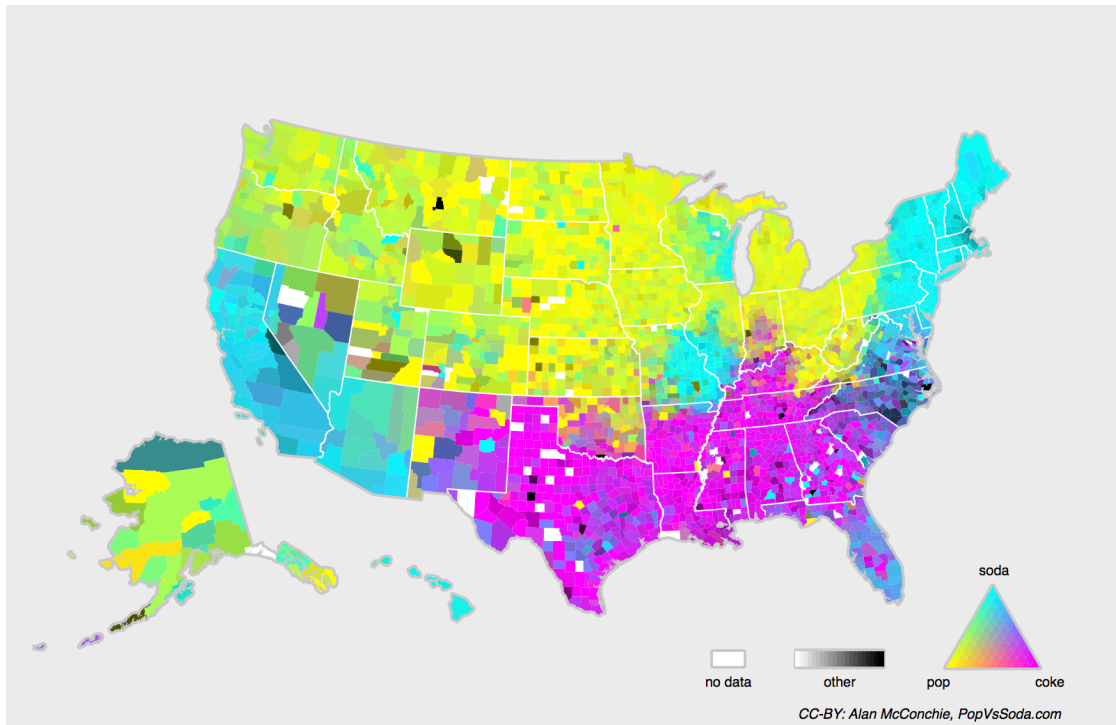
If we were to build a machine translation system, the distribution $p(y|x)$ would be different, e.g. depending on our location. This problem can be quite tricky to spot. A saving grace is that quite often the $p(y|x)$ only shifts gradually (e.g. the click-through rate for NOKIA phone ads). Before we go into further details, let us discuss a number of situations where covariate and concept shift are not quite as blatantly obvious.

4. Multilayer Perceptrons

4.9.3 Examples

Medical Diagnostics

Imagine you want to design some algorithm to detect cancer. You get data of healthy and sick people; you train your algorithm; it works fine, giving you high accuracy and you conclude that you're ready for a successful career in medical diagnostics. Not so fast

Many things could go wrong. In particular, the distributions that you work with for training and those in the wild might differ considerably. This happened to an unfortunate startup I had the opportunity to consult for many years ago. They were developing a blood test for a disease that affects mainly older men and they'd managed to obtain a fair amount of blood samples from patients. It is considerably more difficult, though, to obtain blood samples from healthy men (mainly for ethical reasons). To compensate for that, they asked a large number of students on campus to donate blood and they performed their test. Then they asked me whether I could help them build a classifier to detect the disease. I told them that it would be very easy to distinguish between both datasets with probably near perfect accuracy. After all, the test subjects differed in age, hormone level, physical activity, diet, alcohol consumption, and many more factors unrelated to the disease. This was unlikely to be the case with real patients: Their sampling procedure had caused an extreme case of covariate shift that couldn't be corrected by conventional means. In other words, training and test data were so different that nothing useful could be done and they had wasted significant amounts of money.

Self Driving Cars

A company wanted to build a machine learning system for self-driving cars. One of the key components is a roadside detector. Since real annotated data is expensive to get, they had the (smart and questionable) idea to use synthetic data from a game rendering engine as additional training data. This worked really well on test data' drawn from the rendering engine. Alas, inside a real car it was a disaster. As it turned out, the roadside had been rendered with a very simplistic texture. More importantly, *all* the roadside had been rendered with the *same* texture and the roadside detector learned about this feature' very quickly.

A similar thing happened to the US Army when they first tried to detect tanks in the forest. They took aerial photographs of the forest without tanks, then drove the tanks into the forest and took another set of pictures. The so-trained classifier worked perfectly'. Unfortunately, all it had learned was to distinguish trees with shadows from trees without shadows - the first set of pictures was taken in the early morning, the second one at noon.

Nonstationary distributions

A much more subtle situation is where the distribution changes slowly and the model is not updated adequately. Here are a number of typical cases:

- We train a computational advertising model and then fail to update it frequently (e.g. we forget to incorporate that an obscure new device called an iPad was just launched).

- We build a spam filter. It works well at detecting all spam that we've seen so far. But then the spammers wisen up and craft new messages that look quite unlike anything we've seen before.

- We build a product recommendation system. It works well for the winter. But then it keeps on recommending Santa hats after Christmas.

More Anecdotes

- We build a classifier for Not suitable/safe for work (NSFW) images. To make our life easy, we scrape a few seedy Subreddits. Unfortunately the accuracy on real life data is lacking (the pictures posted on Reddit are mostly remarkable' in some way, e.g. being taken by skilled photographers, whereas most real NSFW images are fairly unremarkable). Quite unsurprisingly the accuracy is not very high on real data.

- We build a face detector. It works well on all benchmarks. Unfortunately it fails on test data - the offending examples are close-ups where the face fills the entire image (no such data was in the training set).

- We build a web search engine for the USA market and want to deploy it in the UK.

In short, there are many cases where training and test distribution $p(x)$ are different. In some cases, we get lucky and the models work despite the covariate shift. We now discuss principled solution strategies. Warning - this will require some math and statistics.

4.9.4 Covariate Shift Correction

Assume that we want to estimate some dependency $p(y|x)$ for which we have labeled data (x_i, y_i). Alas, the observations x_i are drawn from some distribution $q(x)$ rather than the proper' distribution $p(x)$. To make progress, we need to reflect about what exactly is happening during training: we iterate over training data and associated labels $\{(x_1, y_1), \ldots (y_n, y_n)\}$ and update the weight vectors of the model after every minibatch.

Depending on the situation we also apply some penalty to the parameters, such as weight decay, dropout, zoneout, or anything similar. This means that we largely minimize the loss on the training.

$$\underset{w}{\text{minimize}} \frac{1}{n} \sum_{i=1}^{n} l(x_i, y_i, f(x_i)) + \text{some penalty}(w)$$

Statisticians call the first term an *empirical average*, that is an average computed over the data drawn from $p(x)p(y|x)$. If the data is drawn from the wrong' distribution q, we can correct for that by using the following simple identity:

$$\int p(x)f(x)dx = \int p(x)f(x)\frac{q(x)}{p(x)}dx$$
$$= \int q(x)f(x)\frac{p(x)}{q(x)}dx$$

In other words, we need to re-weight each instance by the ratio of probabilities that it would have been drawn from the correct distribution $\beta(x) := p(x)/q(x)$. Alas, we do not know that ratio, so before we can do anything useful we need to estimate it. Many methods are available, e.g. some rather fancy operator theoretic ones which try to recalibrate the expectation operator directly using a minimum-norm or a maximum entropy principle. Note that for any such approach, we need samples drawn from both distributions - the true' p, e.g. by access to training data, and the one used for generating the training set q (the latter is trivially available).

In this case there exists a very effective approach that will give almost as good results: logistic regression. This is all that is needed to compute estimate probability ratios. We learn a classifier to distinguish between data drawn from $p(x)$ and data drawn from $q(x)$. If it is impossible to distinguish between the two distributions then it means that the associated instances are equally likely to come from either one of the two distributions. On the other hand, any instances that can be well discriminated should be significantly over/underweighted accordingly. For simplicity's sake assume that we have an equal number of instances from both distributions, denoted by $x_i \sim p(x)$ and $x_i \sim q(x)$ respectively. Now denote by z_i labels which are 1 for data drawn from p and -1 for data drawn from q. Then the probability in a mixed dataset is given by

$$p(z = 1|x) = \frac{p(x)}{p(x) + q(x)} \text{ and hence } \frac{p(z = 1|x)}{p(z = -1|x)} = \frac{p(x)}{q(x)}$$

Hence, if we use a logistic regression approach where $p(z = 1|x) = \frac{1}{1+\exp(f(x))}$ it follows that

$$\beta(x) = \frac{1/(1 + \exp(-f(x)))}{\exp(-f(x)/(1 + \exp(-f(x))))} = \exp(f(x))$$

As a result, we need to solve two problems: first one to distinguish between data drawn from both distributions, and then a reweighted minimization problem where we weigh terms by β, e.g. via the head gradients. Here's a prototypical algorithm for that purpose which uses an unlabeled training set X and test set Z:

1. Generate training set with $\{(x_i, -1)...(z_j, 1)\}$

2. Train binary classifier using logistic regression to get function f

3. Weigh training data using $\beta_i = \exp(f(x_i))$ or better $\beta_i = \min(\exp(f(x_i)), c)$

4. Use weights β_i for training on X with labels Y

Generative Adversarial Networks use the very idea described above to engineer a *data generator* such that it cannot be distinguished from a reference dataset. For this, we use one network, say f to distinguish real and fake data and a second network g that tries to fool the discriminator f into accepting fake data as real. We will discuss this in much more detail later.

4.9.5 Concept Shift Correction

Concept shift is much harder to fix in a principled manner. For instance, in a situation where suddenly the problem changes from distinguishing cats from dogs to one of distinguishing white from black animals, it will be unreasonable to assume that we can do much better than just training from scratch using the new labels. Fortunately, in practice, such extreme shifts almost never happen. Instead, what usually happens is that the task keeps on changing slowly. To make things more concrete, here are some examples:

- In computational advertising, new products are launched, old products become less popular. This means that the distribution over ads and their popularity changes gradually and any click-through rate predictor needs to change gradually with it.

- Traffic cameras lenses degrade gradually due to environmental wear, affecting image quality progressively.

- News content changes gradually (i.e. most of the news remains unchanged but new stories appear).

In such cases, we can use the same approach that we used for training networks to make them adapt to the change in the data. In other words, we use the existing network weights and simply perform a few update steps with the new data rather than training from scratch.

4.9.6 A Taxonomy of Learning Problems

Armed with knowledge about how to deal with changes in $p(x)$ and in $p(y|x)$, let us consider a number of problems that we can solve using machine learning.

- **Batch Learning.** Here we have access to training data and labels $\{(x_1, y_1), \ldots (x_n, y_n)\}$, which we use to train a network $f(x, w)$. Later on, we deploy this network to score new data (x, y) drawn from the same distribution. This is the default assumption for any of the problems that we discuss here. For instance, we might train a cat detector based on lots of pictures of cats and dogs. Once we trained it, we ship it as part of a smart catdoor computer vision system that lets only cats in. This is then installed in a customer's home and is never updated again (barring extreme circumstances).

- **Online Learning.** Now imagine that the data (x_i, y_i) arrives one sample at a time. More specifically, assume that we first observe x_i, then we need to come up with an estimate $f(x_i, w)$ and only once we've done this, we observe y_i and with it, we receive a reward (or incur a loss), given our decision. Many real problems fall into this category. E.g. we need to predict tomorrow's stock price, this allows us to trade based on that estimate and at the end of the day we find out whether our estimate allowed us to make a profit. In other words, we have the following cycle where we are continuously improving our model given new observations.

$$\text{model } f_t \longrightarrow \text{data } x_t \longrightarrow \text{estimate } f_t(x_t) \longrightarrow \text{observation } y_t \longrightarrow \text{loss } l(y_t, f_t(x_t)) \longrightarrow \text{model } f_{t+1}$$

- **Bandits.** They are a *special case* of the problem above. While in most learning problems we have a continuously parametrized function f where we want to learn its parameters (e.g. a deep network), in a bandit problem we only have a finite number of arms that we can pull (i.e. a finite number of actions that we can take). It is not very surprising that for this simpler problem stronger theoretical

guarantees in terms of optimality can be obtained. We list it mainly since this problem is often (confusingly) treated as if it were a distinct learning setting.

- **Control (and nonadversarial Reinforcement Learning).** In many cases the environment remembers what we did. Not necessarily in an adversarial manner but it'll just remember and the response will depend on what happened before. E.g. a coffee boiler controller will observe different temperatures depending on whether it was heating the boiler previously. PID (proportional integral derivative) controller algorithms are a popular choice there. Likewise, a user's behavior on a news site will depend on what we showed him previously (e.g. he will read most news only once). Many such algorithms form a model of the environment in which they act such as to make their decisions appear less random (i.e. to reduce variance).

- **Reinforcement Learning.** In the more general case of an environment with memory, we may encounter situations where the environment is trying to *cooperate* with us (cooperative games, in particular for non-zero-sum games), or others where the environment will try to *win*. Chess, Go, Backgammon or StarCraft are some of the cases. Likewise, we might want to build a good controller for autonomous cars. The other cars are likely to respond to the autonomous car's driving style in nontrivial ways, e.g. trying to avoid it, trying to cause an accident, trying to cooperate with it, etc.

One key distinction between the different situations above is that the same strategy that might have worked throughout in the case of a stationary environment, might not work throughout when the environment can adapt. For instance, an arbitrage opportunity discovered by a trader is likely to disappear once he starts exploiting it. The speed and manner at which the environment changes determines to a large extent the type of algorithms that we can bring to bear. For instance, if we *know* that things may only change slowly, we can force any estimate to change only slowly, too. If we know that the environment might change instantaneously, but only very infrequently, we can make allowances for that. These types of knowledge are crucial for the aspiring data scientist to deal with concept shift, i.e. when the problem that he is trying to solve changes over time.

Summary

- In many cases training and test set do not come from the same distribution. This is called covariate shift.

- Covariate shift can be detected and corrected if the shift isn't too severe. Failure to do so leads to nasty surprises at test time.

- In some cases the environment *remembers* what we did and will respond in unexpected ways. We need to account for that when building models.

Exercises

1. What could happen when we change the behavior of a search engine? What might the users do? What about the advertisers?

2. Implement a covariate shift detector. Hint - build a classifier.

3. Implement a covariate shift corrector.

4. What could go wrong if training and test set are very different? What would happen to the sample weights?

Scan the QR Code to Discuss

4.10 Predicting House Prices on Kaggle

The previous chapters introduced a number of basic tools to build deep networks and to perform capacity control using dimensionality, weight decay and dropout. It's time to put our knowledge to good use by participating in a Kaggle competition. Predicting house prices is the perfect start, since its data is fairly generic and doesn't have much regular structure in the way text of images do. Note that the dataset is *not* the Boston housing dataset of Harrison and Rubinfeld, 1978. Instead, it consists of the larger and more fully-featured dataset of house prices in Ames, IA covering 2006-2010. It was collected by Bart de Cock in 2011. Due to its larger size it presents a slightly more interesting estimation problem.

In this chapter we will apply what we've learned so far. In particular, we will walk you through details of data preprocessing, model design, hyperparameter selection and tuning. We hope that through a hands-on approach you will be able to observe the effects of capacity control, feature extraction, etc. in practice. Such experience is vital if you want to become an experienced data scientist.

4.10.1 Kaggle

Kaggle is a popular platform for machine learning competitions. It combines data, code and users in a way to allow for both collaboration and competition. For instance, you can see the code that (some) competitors submitted and you can see how well you're doing relative to everyone else. If you want to participate in one of the competitions, you need to register for an account. So it's best to do this now.
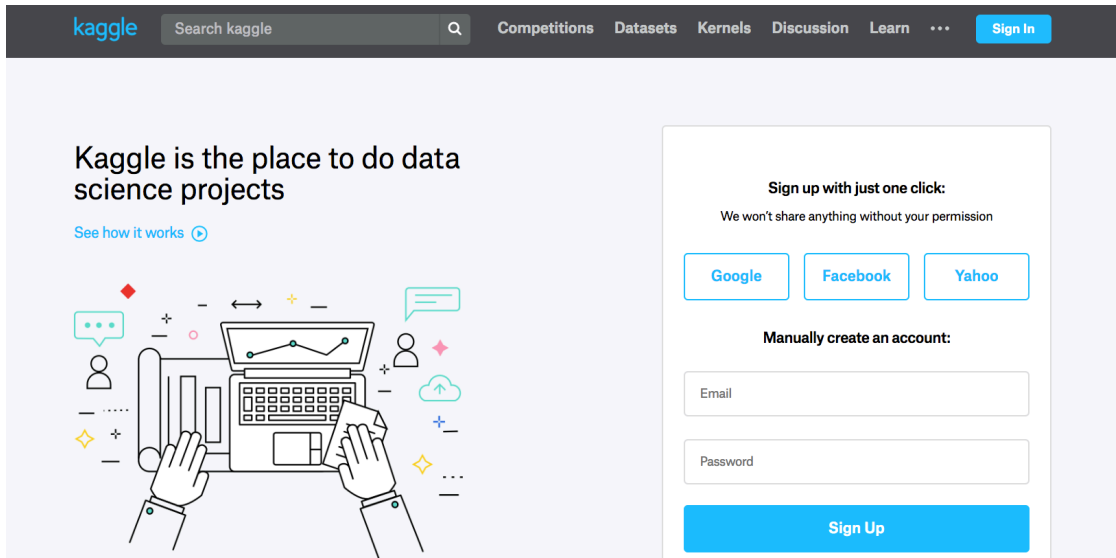
Fig. 4.6: Kaggle website

On the House Prices Prediction page you can find the data set (under the data tab), submit predictions, see your ranking, etc.; You can find it at the URL below:

https://www.kaggle.com/c/house-prices-advanced-regression-techniques
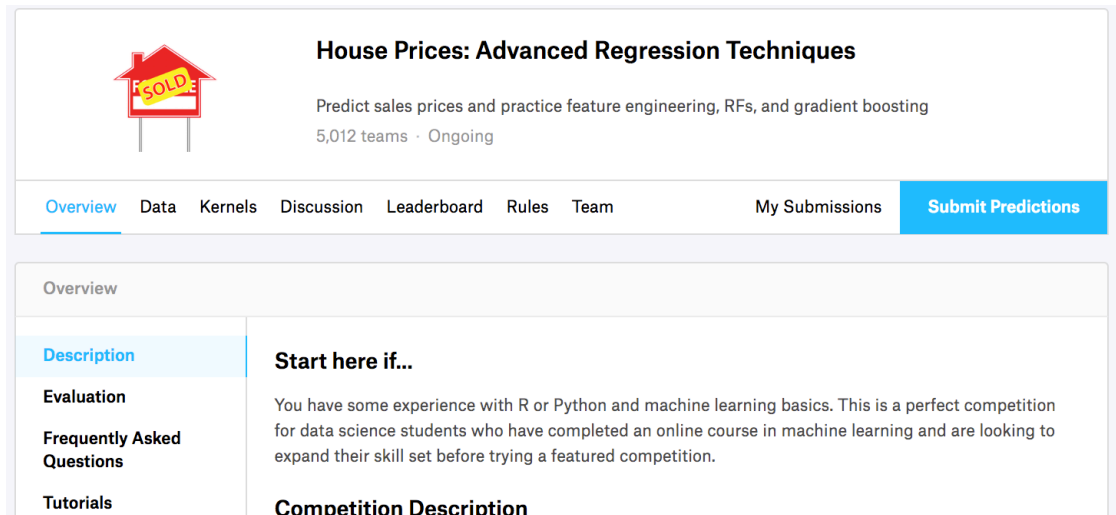
Fig. 4.7: House Price Prediction

4.10.2 Accessing and Reading Data Sets

The competition data is separated into training and test sets. Each record includes the property values of the house and attributes such as street type, year of construction, roof type, basement condition. The data includes multiple datatypes, including integers (year of construction), discrete labels (roof type), floating point numbers, etc.; Some data is missing and is thus labeled na'. The price of each house, namely the label, is only included in the training data set (it's a competition after all). The Data' tab on the competition tab has links to download the data.

We will read and process the data using `pandas`, an efficient data analysis toolkit. Make sure you have `pandas` installed for the experiments in this section.

```
In [1]: # If pandas is not installed, please uncomment the following line:
        # !pip install pandas

        import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet import autograd, gluon, init, nd
        from mxnet.gluon import data as gdata, loss as gloss, nn
        import numpy as np
        import pandas as pd
```

For convenience we already downloaded the data and stored it in the `../data` directory. To load the two CSV (Comma Separated Values) files containing training and test data respectively we use Pandas.

```
In [2]: train_data = pd.read_csv('../data/kaggle_house_pred_train.csv')
        test_data = pd.read_csv('../data/kaggle_house_pred_test.csv')
```

The training data set includes 1,460 examples, 80 features, and 1 label., the test data contains 1,459 examples and 80 features.

```
In [3]: print(train_data.shape)
        print(test_data.shape)

(1460, 81)
(1459, 80)
```

Let's take a look at the first 4 and last 2 features as well as the label (SalePrice) from the first 4 examples:

```
In [4]: train_data.iloc[0:4, [0, 1, 2, 3, -3, -2, -1]]

Out[4]:    Id  MSSubClass MSZoning  LotFrontage SaleType SaleCondition  SalePrice
        0   1          60       RL         65.0       WD        Normal     208500
        1   2          20       RL         80.0       WD        Normal     181500
        2   3          60       RL         68.0       WD        Normal     223500
        3   4          70       RL         60.0       WD       Abnorml     140000
```

We can see that in each example, the first feature is the ID. This helps the model identify each training example. While this is convenient, it doesn't carry any information for prediction purposes. Hence we remove it from the dataset before feeding the data into the network.

```
In [5]: all_features = pd.concat((train_data.iloc[:, 1:-1], test_data.iloc[:, 1:]))
```

4.10.3 Data Preprocessing

As stated above, we have a wide variety of datatypes. Before we feed it into a deep network we need to perform some amount of processing. Let's start with the numerical features. We begin by replacing missing values with the mean. This is a reasonable strategy if features are missing at random. To adjust them to a common scale we rescale them to zero mean and unit variance. This is accomplished as follows:

$$x \leftarrow \frac{x - \mu}{\sigma}$$

To check that this transforms x to data with zero mean and unit variance simply calculate $\mathbf{E}[(x - \mu)/\sigma] = (\mu - \mu)/\sigma = 0$. To check the variance we use $\mathbf{E}[(x - \mu)^2] = \sigma^2$ and thus the transformed variable has unit variance. The reason for normalizing' the data is that it brings all features to the same order of magnitude. After all, we do not know *a priori* which features are likely to be relevant. Hence it makes sense to treat them equally.

```
In [6]: numeric_features = all_features.dtypes[all_features.dtypes != 'object'].index
        all_features[numeric_features] = all_features[numeric_features].apply(
            lambda x: (x - x.mean()) / (x.std()))
        # After standardizing the data all means vanish, hence we can set missing
        # values to 0
        all_features = all_features.fillna(0)
```

Next we deal with discrete values. This includes variables such as MSZoning'. We replace them by a one-hot encoding in the same manner as how we transformed multiclass classification data into a vector of 0 and 1. For instance, MSZoning' assumes the values RL' and RM'. They map into vectors $(1, 0)$ and $(0, 1)$ respectively. Pandas does this automatically for us.

```
In [7]: # Dummy_na=True refers to a missing value being a legal eigenvalue, and
        # creates an indicative feature for it
        all_features = pd.get_dummies(all_features, dummy_na=True)
        all_features.shape
Out[7]: (2919, 354)
```

You can see that this conversion increases the number of features from 79 to 331. Finally, via the `values` attribute we can extract the NumPy format from the Pandas dataframe and convert it into MXNet's native representation - NDArray for training.

```
In [8]: n_train = train_data.shape[0]
        train_features = nd.array(all_features[:n_train].values)
        test_features = nd.array(all_features[n_train:].values)
        train_labels = nd.array(train_data.SalePrice.values).reshape((-1, 1))
```

4.10.4 Training

To get started we train a linear model with squared loss. This will obviously not lead to a competition winning submission but it provides a sanity check to see whether there's meaningful information in the data. It also amounts to a minimum baseline of how well we should expect any fancy' model to work.

```
In [9]: loss = gloss.L2Loss()
```

```
def get_net():
    net = nn.Sequential()
    net.add(nn.Dense(1))
    net.initialize()
    return net
```

House prices, like shares, are relative. That is, we probably care more about the relative error $\frac{y - \hat{y}}{y}$ than about the absolute error. For instance, getting a house price wrong by USD 100,000 is terrible in Rural Ohio, where the value of the house is USD 125,000. On the other hand, if we err by this amount in Los Altos Hills, California, we can be proud of the accuracy of our model (the median house price there exceeds 4 million).

One way to address this problem is to measure the discrepancy in the logarithm of the price estimates. In fact, this is also the error that is being used to measure the quality in this competition. After all, a small value δ of $\log y - \log \hat{y}$ translates into $e^{-\delta} \leq \frac{\hat{y}}{y} \leq e^{\delta}$. This leads to the following loss function:

$$L = \sqrt{\frac{1}{n} \sum_{i=1}^{n} \left(\log y_i - \log \hat{y}_i \right)^2}$$

```
In [10]: def log_rmse(net, features, labels):
             # To further stabilize the value when the logarithm is taken, set the
             # value less than 1 as 1
             clipped_preds = nd.clip(net(features), 1, float('inf'))
             rmse = nd.sqrt(2 * loss(clipped_preds.log(), labels.log()).mean())
             return rmse.asscalar()
```

Unlike in the previous sections, the following training functions use the Adam optimization algorithm. Compared to the previously used mini-batch stochastic gradient descent, the Adam optimization algorithm is relatively less sensitive to learning rates. This will be covered in further detail later on when we discuss the details on *Optimization Algorithms* in a separate chapter.

```
In [11]: def train(net, train_features, train_labels, test_features, test_labels,
                   num_epochs, learning_rate, weight_decay, batch_size):
             train_ls, test_ls = [], []
             train_iter = gdata.DataLoader(gdata.ArrayDataset(
                 train_features, train_labels), batch_size, shuffle=True)
             # The Adam optimization algorithm is used here
             trainer = gluon.Trainer(net.collect_params(), 'adam', {
                 'learning_rate': learning_rate, 'wd': weight_decay})
             for epoch in range(num_epochs):
                 for X, y in train_iter:
                     with autograd.record():
                         l = loss(net(X), y)
                     l.backward()
                     trainer.step(batch_size)
                 train_ls.append(log_rmse(net, train_features, train_labels))
                 if test_labels is not None:
                     test_ls.append(log_rmse(net, test_features, test_labels))
             return train_ls, test_ls
```

4.10.5 k-Fold Cross-Validation

The k-fold cross-validation was introduced in the section where we discussed how to deal with *Model Selection, Underfitting and Overfitting*. We will put this to good use to select the model design and to adjust the hyperparameters. We first need a function that returns the i-th fold of the data in a k-fold cross-validation procedure. It proceeds by slicing out the i-th segment as validation data and returning the rest as training data. Note - this is not the most efficient way of handling data and we would use something much smarter if the amount of data was considerably larger. But this would obscure the function of the code considerably and we thus omit it.

```
In [12]: def get_k_fold_data(k, i, X, y):
             assert k > 1
             fold_size = X.shape[0] // k
             X_train, y_train = None, None
             for j in range(k):
                 idx = slice(j * fold_size, (j + 1) * fold_size)
                 X_part, y_part = X[idx, :], y[idx]
                 if j == i:
                     X_valid, y_valid = X_part, y_part
                 elif X_train is None:
                     X_train, y_train = X_part, y_part
                 else:
                     X_train = nd.concat(X_train, X_part, dim=0)
                     y_train = nd.concat(y_train, y_part, dim=0)
             return X_train, y_train, X_valid, y_valid
```

The training and verification error averages are returned when we train k times in the k-fold cross-validation.

```
In [13]: def k_fold(k, X_train, y_train, num_epochs,
                    learning_rate, weight_decay, batch_size):
             train_l_sum, valid_l_sum = 0, 0
             for i in range(k):
                 data = get_k_fold_data(k, i, X_train, y_train)
                 net = get_net()
                 train_ls, valid_ls = train(net, *data, num_epochs, learning_rate,
                                            weight_decay, batch_size)
                 train_l_sum += train_ls[-1]
                 valid_l_sum += valid_ls[-1]
                 if i == 0:
                     d2l.semilogy(range(1, num_epochs + 1), train_ls, 'epochs', 'rmse',
                                  range(1, num_epochs + 1), valid_ls,
                                  ['train', 'valid'])
                 print('fold %d, train rmse: %f, valid rmse: %f' % (
                     i, train_ls[-1], valid_ls[-1]))
             return train_l_sum / k, valid_l_sum / k
```

4.10.6 Model Selection

We pick a rather un-tuned set of hyperparameters and leave it up to the reader to improve the model considerably. Finding a good choice can take quite some time, depending on how many things one wants to optimize over. Within reason the k-fold crossvalidation approach is resilient against multiple testing.

4.10. Predicting House Prices on Kaggle   **185**

However, if we were to try out an unreasonably large number of options it might fail since we might just get lucky on the validation split with a particular set of hyperparameters.

```
In [14]: k, num_epochs, lr, weight_decay, batch_size = 5, 100, 5, 0, 64
         train_l, valid_l = k_fold(k, train_features, train_labels, num_epochs, lr,
                                   weight_decay, batch_size)
         print('%d-fold validation: avg train rmse: %f, avg valid rmse: %f'
               % (k, train_l, valid_l))
```
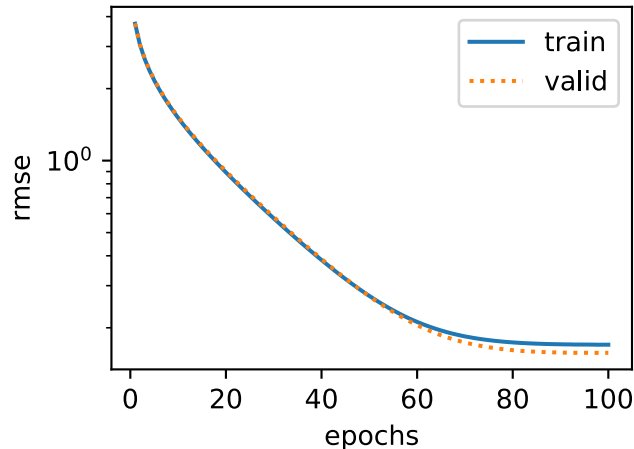


```
fold 0, train rmse: 0.169928, valid rmse: 0.157110
fold 1, train rmse: 0.162353, valid rmse: 0.189469
fold 2, train rmse: 0.163828, valid rmse: 0.167961
fold 3, train rmse: 0.167681, valid rmse: 0.154751
fold 4, train rmse: 0.162711, valid rmse: 0.182902
5-fold validation: avg train rmse: 0.165300, avg valid rmse: 0.170439
```

You will notice that sometimes the number of training errors for a set of hyper-parameters can be very low, while the number of errors for the K-fold cross validation may be higher. This is most likely a consequence of overfitting. Therefore, when we reduce the amount of training errors, we need to check whether the amount of errors in the k-fold cross-validation have also been reduced accordingly.

4.10.7 Predict and Submit

Now that we know what a good choice of hyperparameters should be, we might as well use all the data to train on it (rather than just $1 - 1/k$ of the data that is used in the crossvalidation slices). The model that we obtain in this way can then be applied to the test set. Saving the estimates in a CSV file will simplify uploading the results to Kaggle.

```
In [15]: def train_and_pred(train_features, test_feature, train_labels, test_data,
                           num_epochs, lr, weight_decay, batch_size):
             net = get_net()
             train_ls, _ = train(net, train_features, train_labels, None, None,
                                 num_epochs, lr, weight_decay, batch_size)
```

```
d2l.semilogy(range(1, num_epochs + 1), train_ls, 'epochs', 'rmse')
print('train rmse %f' % train_ls[-1])
# Apply the network to the test set
preds = net(test_features).asnumpy()
# Reformat it for export to Kaggle
test_data['SalePrice'] = pd.Series(preds.reshape(1, -1)[0])
submission = pd.concat([test_data['Id'], test_data['SalePrice']], axis=1)
submission.to_csv('submission.csv', index=False)
```

Let's invoke the model. A good sanity check is to see whether the predictions on the test set resemble those of the k-fold crossvalidation process. If they do, it's time to upload them to Kaggle.

```
In [16]: train_and_pred(train_features, test_features, train_labels, test_data,
                        num_epochs, lr, weight_decay, batch_size)
```
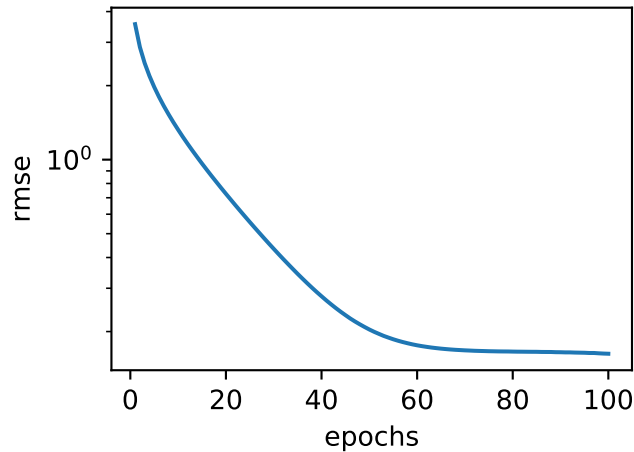


```
train rmse 0.162496
```

A file, `submission.csv` will be generated by the code above (CSV is one of the file formats accepted by Kaggle). Next, we can submit our predictions on Kaggle and compare them to the actual house price (label) on the testing data set, checking for errors. The steps are quite simple:

- Log in to the Kaggle website and visit the House Price Prediction Competition page.
- Click the Submit Predictions or Late Submission button on the right.
- Click the Upload Submission File button in the dashed box at the bottom of the page and select the prediction file you wish to upload.
- Click the Make Submission button at the bottom of the page to view your results.

Fig. 4.8: Submitting data to Kaggle

Summary

- Real data often contains a mix of different datatypes and needs to be preprocessed.

- Rescaling real-valued data to zero mean and unit variance is a good default. So is replacing missing values with their mean.

- Transforming categorical variables into indicator variables allows us to treat them like vectors.

- We can use k-fold cross validation to select the model and adjust the hyper-parameters.

- Logarithms are useful for relative loss.

Exercises

1. Submit your predictions for this tutorial to Kaggle. How good are your predictions?

2. Can you improve your model by minimizing the log-price directly? What happens if you try to predict the log price rather than the price?

3. Is it always a good idea to replace missing values by their mean? Hint - can you construct a situation where the values are not missing at random?

4. Find a better representation to deal with missing values. Hint - What happens if you add an indicator variable?

5. Improve the score on Kaggle by tuning the hyperparameters through k-fold crossvalidation.

6. Improve the score by improving the model (layers, regularization, dropout).

7. What happens if we do not standardize the continuous numerical features like we have done in this section?

Scan the QR Code to Discuss

5

Deep Learning Computation

The previous chapter introduced the principles and implementation for a simple deep learning model, including multi-layer perceptrons. In this chapter we will cover various key components of deep learning computation, such as model construction, parameter access and initialization, custom layers, and reading, storing, and using GPUs. Throughout this chapter, you will gain important insights into model implementation and computation details, which gives readers a solid foundation for implementing more complex models in the following chapters.

5.1 Layers and Blocks

One of the key components that helped propel deep learning is powerful software. In an analogous manner to semiconductor design where engineers went from specifying transistors to logical circuits to writing code we now witness similar progress in the design of deep networks. The previous chapters have seen us move from designing single neurons to entire layers of neurons. However, even network design by layers can be tedious when we have 152 layers, as is the case in ResNet-152, which was proposed by He et al. in 2016 for computer vision problems. Such networks have a fair degree of regularity and they consist of *blocks* of repeated (or at least similarly designed) layers. These blocks then form the basis of more complex network designs. In short, blocks are combinations of one or more layers. This design is aided by code that generates such blocks on demand, just like a Lego factory generates blocks which can be combined to produce terrific artifacts.

We start with very simple block, namely the block for a multilayer perceptron, such as the one we encountered previously. A common strategy would be to construct a two-layer network as follows:

```
In [1]: from mxnet import nd
        from mxnet.gluon import nn

        x = nd.random.uniform(shape=(2, 20))

        net = nn.Sequential()
        net.add(nn.Dense(256, activation='relu'))
        net.add(nn.Dense(10))
        net.initialize()
        net(x)
Out[1]:
        [[ 0.09543004  0.04614332 -0.00286654 -0.07790349 -0.05130243  0.02942037
           0.08696642 -0.0190793  -0.04122177  0.05088576]
         [ 0.0769287   0.03099705  0.00856576 -0.04467199 -0.06926839  0.09132434
           0.06786595 -0.06187842 -0.03436673  0.04234694]]
        <NDArray 2x10 @cpu(0)>
```

This generates a network with a hidden layer of 256 units, followed by a ReLu activation and another 10 units governing the output. In particular, we used the `nn.Sequential` constructor to generate an empty network into which we then inserted both layers. What exactly happens inside `nn.Sequential` has remained rather mysterious so far. In the following we will see that this really just constructs a block. These blocks can be combined into larger artifacts, often recursively. The diagram below shows how:
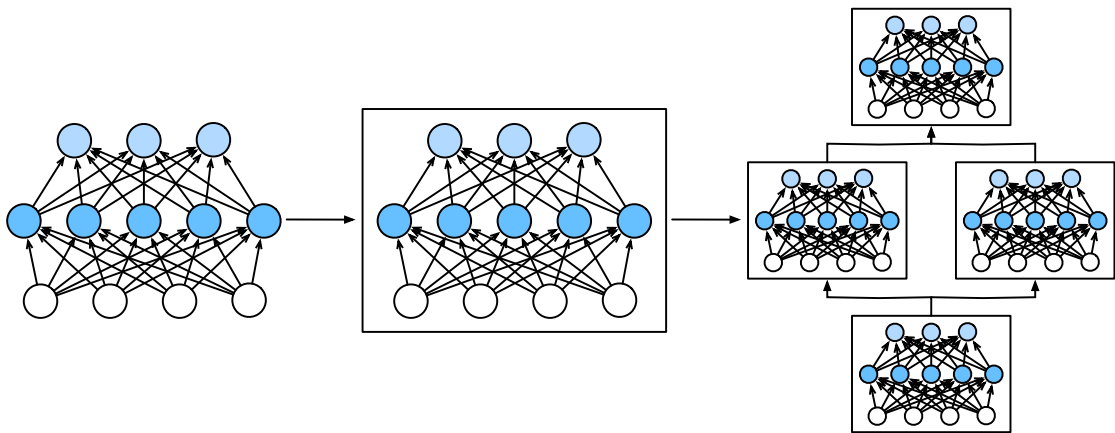
Fig. 5.1: Multiple layers are combined into blocks

In the following we will explain the various steps needed to go from defining layers to defining blocks (of one or more layers). To get started we need a bit of reasoning about software. For most intents and purposes a block behaves very much like a fancy layer. That is, it provides the following functionality:

1. It needs to ingest data (the input).

2. It needs to produce a meaningful output. This is typically encoded in what we will call the `forward` function. It allows us to invoke a block via `net(X)` to obtain the desired output. What happens behind the scenes is that it invokes `forward` to perform forward propagation.

3. It needs to produce a gradient with regard to its input when invoking `backward`. Typically this is automatic.

4. It needs to store parameters that are inherent to the block. For instance, the block above contains two hidden layers, and we need a place to store parameters for it.

5. Obviously it also needs to initialize these parameters as needed.

5.1.1 A Custom Block

The `nn.Block` class provides the functionality required for much of what we need. It is a model constructor provided in the `nn` module, which we can inherit to define the model we want. The following inherits the Block class to construct the multilayer perceptron mentioned at the beginning of this section. The `MLP` class defined here overrides the `__init__` and `forward` functions of the Block class. They are used to create model parameters and define forward computations, respectively. Forward computation is also forward propagation.

```
In [2]: from mxnet import nd
        from mxnet.gluon import nn

        class MLP(nn.Block):
            # Declare a layer with model parameters. Here, we declare two fully
            # connected layers
            def __init__(self, **kwargs):
                # Call the constructor of the MLP parent class Block to perform the
                # necessary initialization. In this way, other function parameters can
                # also be specified when constructing an instance, such as the model
                # parameter, params, described in the following sections
                super(MLP, self).__init__(**kwargs)
                self.hidden = nn.Dense(256, activation='relu')  # Hidden layer
                self.output = nn.Dense(10)  # Output layer

            # Define the forward computation of the model, that is, how to return the
            # required model output based on the input x
            def forward(self, x):
                return self.output(self.hidden(x))
```

Let's look at it a bit more closely. The `forward` method invokes a network simply by evaluating the hidden layer `self.hidden(x)` and subsequently by evaluating the output layer `self.output( . .. )`. This is what we expect in the forward pass of this block.

In order for the block to know what it needs to evaluate, we first need to define the layers. This is what the `__init__` method does. It first initializes all of the Block-related parameters and then constructs the requisite layers. This attached the corresponding layers and the required parameters to the class. Note that there is no need to define a backpropagation method in the class. The system automatically generates the `backward` method needed for back propagation by automatically finding the gradient. The same applies to the `initialize` method, which is generated automatically. Let's try this out:

```
In [3]: net = MLP()
        net.initialize()
        net(x)
```

```
Out[3]:
        [[ 0.00362228  0.00633332  0.03201144 -0.01369375  0.10336449 -0.03508018
          -0.00032164 -0.01676023  0.06978628  0.01303309]
         [ 0.03871715  0.02608213  0.03544959 -0.02521311  0.11005433 -0.0143066
          -0.03052466 -0.03852827  0.06321152  0.0038594 ]]
        <NDArray 2x10 @cpu(0)>
```

As explained above, the block class can be quite versatile in terms of what it does. For instance, its subclass can be a layer (such as the `Dense` class provided by Gluon), it can be a model (such as the `MLP` class we just derived), or it can be a part of a model (this is what typically happens when designing very deep networks). Throughout this chapter we will see how to use this with great flexibility.

5.1.2 A Sequential Block

The Block class is a generic component describing dataflow. In fact, the Sequential class is derived from the Block class: when the forward computation of the model is a simple concatenation of computations for each layer, we can define the model in a much simpler way. The purpose of the Sequential class is to provide some useful convenience functions. In particular, the `add` method allows us to add concatenated Block subclass instances one by one, while the forward computation of the model is to compute these instances one by one in the order of addition. Below, we implement a `MySequential` class that has the same functionality as the Sequential class. This may help you understand more clearly how the Sequential class works.

```
In [4]: class MySequential(nn.Block):
            def __init__(self, **kwargs):
                super(MySequential, self).__init__(**kwargs)

            def add(self, block):
                # Here, block is an instance of a Block subclass, and we assume it has
                # a unique name. We save it in the member variable _children of the
                # Block class, and its type is OrderedDict. When the MySequential
                # instance calls the initialize function, the system automatically
                # initializes all members of _children
                self._children[block.name] = block

            def forward(self, x):
                # OrderedDict guarantees that members will be traversed in the order
                # they were added
                for block in self._children.values():
                    x = block(x)
                return x
```

At its core is the `add` method. It adds any block to the ordered dictionary of children. These are then executed in sequence when forward propagation is invoked. Let's see what the MLP looks like now.

```
In [5]: net = MySequential()
        net.add(nn.Dense(256, activation='relu'))
        net.add(nn.Dense(10))
        net.initialize()
        net(x)
Out[5]:
        [[ 0.07787765  0.00216401  0.01682201  0.03059879 -0.00702019  0.01668714
```

```
       0.04822845   0.00394321 -0.09300036 -0.044943  ]
     [ 0.08891079 -0.00625484 -0.01619132  0.03807178 -0.01451489  0.02006172
       0.0303478    0.02463485 -0.07605445 -0.04389167]]
    <NDArray 2x10 @cpu(0)>
```

Indeed, it is no different than It can observed here that the use of the `MySequential` class is no different from the use of the Sequential class described in the Concise Implementation of Multilayer Perceptron section.

5.1.3 Blocks with Code

Although the Sequential class can make model construction easier, and you do not need to define the `forward` method, directly inheriting the Block class can greatly expand the flexibility of model construction. In particular, we will use Python's control flow within the forward method. While we're at it, we need to introduce another concept, that of the *constant* parameter. These are parameters that are not used when invoking backprop. This sounds very abstract but here's what's really going on. Assume that we have some function

$$f(\mathbf{x}, \mathbf{w}) = 3 \cdot \mathbf{w}^\top \mathbf{x}.$$

In this case 3 is a constant parameter. We could change 3 to something else, say c via

$$f(\mathbf{x}, \mathbf{w}) = c \cdot \mathbf{w}^\top \mathbf{x}.$$

Nothing has really changed, except that we can adjust the value of c. It is still a constant as far as $\mathbf{w}$ and $\mathbf{x}$ are concerned. However, since Gluon doesn't know about this beforehand, it's worth while to give it a hand (this makes the code go faster, too, since we're not sending the Gluon engine on a wild goose chase after a parameter that doesn't change). `get_constant` is the method that can be used to accomplish this. Let's see what this looks like in practice.

```python
In [6]: class FancyMLP(nn.Block):
            def __init__(self, **kwargs):
                super(FancyMLP, self).__init__(**kwargs)
                # Random weight parameters created with the get_constant are not
                # iterated during training (i.e. constant parameters)
                self.rand_weight = self.params.get_constant(
                    'rand_weight', nd.random.uniform(shape=(20, 20)))
                self.dense = nn.Dense(20, activation='relu')

            def forward(self, x):
                x = self.dense(x)
                # Use the constant parameters created, as well as the relu and dot
                # functions of NDArray
                x = nd.relu(nd.dot(x, self.rand_weight.data()) + 1)
                # Reuse the fully connected layer. This is equivalent to sharing
                # parameters with two fully connected layers
                x = self.dense(x)
                # Here in Control flow, we need to call asscalar to return the scalar
                # for comparison
                while x.norm().asscalar() > 1:
```

```
        x /= 2
    if x.norm().asscalar() < 0.8:
        x *= 10
    return x.sum()
```

In this `FancyMLP` model, we used constant weight `Rand_weight` (note that it is not a model parameter), performed a matrix multiplication operation (`nd.dot<`), and reused the *same* `Dense` layer. Note that this is very different from using two dense layers with different sets of parameters. Instead, we used the same network twice. Quite often in deep networks one also says that the parameters are *tied* when one wants to express that multiple parts of a network share the same parameters. Let's see what happens if we construct it and feed data through it.

```
In [7]: net = FancyMLP()
        net.initialize()
        net(x)

Out[7]:
        [25.522684]
        <NDArray 1 @cpu(0)>
```

There's no reason why we couldn't mix and match these ways of build a network. Obviously the example below resembles more a chimera, or less charitably, a Rube Goldberg Machine. That said, it combines examples for building a block from individual blocks, which in turn, may be blocks themselves. Furthermore, we can even combine multiple strategies inside the same forward function. To demonstrate this, here's the network.

```
In [8]: class NestMLP(nn.Block):
            def __init__(self, **kwargs):
                super(NestMLP, self).__init__(**kwargs)
                self.net = nn.Sequential()
                self.net.add(nn.Dense(64, activation='relu'),
                             nn.Dense(32, activation='relu'))
                self.dense = nn.Dense(16, activation='relu')

            def forward(self, x):
                return self.dense(self.net(x))

        chimera = nn.Sequential()
        chimera.add(NestMLP(), nn.Dense(20), FancyMLP())

        chimera.initialize()
        chimera(x)

Out[8]:
        [30.518448]
        <NDArray 1 @cpu(0)>
```

5.1.4 Compilation

The avid reader is probably starting to worry about the efficiency of this. After all, we have lots of dictionary lookups, code execution, and lots of other Pythonic things going on in what is supposed to be a high performance deep learning library. The problems of Python's Global Interpreter Lock are well known. In the context of deep learning it means that we have a super fast GPU (or multiple of them)

---

which might have to wait until a puny single CPU core running Python gets a chance to tell it what to do next. This is clearly awful and there are many ways around it. The best way to speed up Python is by avoiding it altogether.

Gluon does this by allowing for *Hybridization*. In it, the Python interpreter executes the block the first time it's invoked. The Gluon runtime records what is happening and the next time around it short circuits any calls to Python. This can accelerate things considerably in some cases but care needs to be taken with control flow. We suggest that the interested reader skip forward to the section covering hybridization and compilation after finishing the current chapter.

Summary

- Layers are blocks
- Many layers can be a block
- Many blocks can be a block
- Code can be a block
- Blocks take are of a lot of housekeeping, such as parameter initialization, backprop and related issues.
- Sequential concatenations of layers and blocks are handled by the eponymous `Sequential` block.

Exercises

1. What kind of error message will you get when calling an `__init__` method whose parent class not in the `__init__` function of the parent class?

2. What kinds of problems will occur if you remove the `asscalar` function in the `FancyMLP` class?

3. What kinds of problems will occur if you change `self.net` defined by the Sequential instance in the `NestMLP` class to `self.net = [nn.Dense(64, activation='relu'), nn. Dense(32, activation='relu')]`?

4. Implement a block that takes two blocks as an argument, say `net1` and `net2` and returns the concatenated output of both networks in the forward pass (this is also called a parallel block).

5. Assume that you want to concatenate multiple instances of the same network. Implement a factory function that generates multiple instances of the same block and build a larger network from it.

Scan the QR Code to Discuss

5.2 Parameter Management

The ultimate goal of training deep networks is to find good parameter values for a given architecture. When everything is standard, the `nn.Sequential` class is a perfectly good tool for it. However, very few models are entirely standard and most scientists want to build things that are novel. This section shows how to manipulate parameters. In particular we will cover the following aspects:

- Accessing parameters for debugging, diagnostics, to visualize them or to save them is the first step to understanding how to work with custom models.

- Secondly, we want to set them in specific ways, e.g. for initialization purposes. We discuss the structure of parameter initializers.

- Lastly, we show how this knowledge can be put to good use by building networks that share some parameters.

As always, we start from our trusty Multilayer Perceptron with a hidden layer. This will serve as our choice for demonstrating the various features.

```
In [1]: from mxnet import init, nd
        from mxnet.gluon import nn

        net = nn.Sequential()
        net.add(nn.Dense(256, activation='relu'))
        net.add(nn.Dense(10))
        net.initialize()  # Use the default initialization method

        x = nd.random.uniform(shape=(2, 20))
        net(x)  # Forward computation
Out[1]:
        [[ 0.09543004  0.04614332 -0.00286654 -0.07790349 -0.05130243  0.02942037
           0.08696642 -0.0190793  -0.04122177  0.05088576]
         [ 0.0769287   0.03099705  0.00856576 -0.04467199 -0.06926839  0.09132434
           0.06786595 -0.06187842 -0.03436673  0.04234694]]
        <NDArray 2x10 @cpu(0)>
```

5.2.1 Parameter Access

In the case of a Sequential class we can access the parameters with ease, simply by indexing each of the layers in the network. The params variable then contains the required data. Let's try this out in practice by inspecting the parameters of the first layer.

```
In [2]: print(net[0].params)
        print(net[1].params)

dense0_ (
  Parameter dense0_weight (shape=(256, 20), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
)
dense1_ (
  Parameter dense1_weight (shape=(10, 256), dtype=float32)
  Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

The output tells us a number of things. Firstly, the layer consists of two sets of parameters: dense0_weight and dense0_bias, as we would expect. They are both single precision and they have the necessary shapes that we would expect from the first layer, given that the input dimension is 20 and the output dimension 256. In particular the names of the parameters are very useful since they allow us to identify parameters *uniquely* even in a network of hundreds of layers and with nontrivial structure. The second layer is structured accordingly.

Targeted Parameters

In order to do something useful with the parameters we need to access them, though. There are several ways to do this, ranging from simple to general. Let's look at some of them.

```
In [3]: print(net[1].bias)
        print(net[1].bias.data())

Parameter dense1_bias (shape=(10,), dtype=float32)

[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
<NDArray 10 @cpu(0)>
```

The first returns the bias of the second layer. Since this is an object containing data, gradients, and additional information, we need to request the data explicitly. Note that the bias is all 0 since we initialized the bias to contain all zeros. Note that we can also access the parameters by name, such as dense0_weight. This is possible since each layer comes with its own parameter dictionary that can be accessed directly. Both methods are entirely equivalent but the first method leads to much more readable code.

```
In [4]: print(net[0].params['dense0_weight'])
        print(net[0].params['dense0_weight'].data())

Parameter dense0_weight (shape=(256, 20), dtype=float32)

[[ 0.06700657 -0.00369488  0.0418822  ... -0.05517294 -0.01194733
  -0.00369594]
 [-0.03296221 -0.04391347  0.03839272 ...  0.05636378  0.02545484
```

```
 -0.007007  ]
 [-0.0196689   0.01582889 -0.00881553 ...  0.01509629 -0.01908049
  -0.02449339]
 ...
 [ 0.00010955  0.0439323  -0.04911506 ...  0.06975312  0.0449558
  -0.03283203]
 [ 0.04106557  0.05671307 -0.00066976 ...  0.06387014 -0.01292654
   0.00974177]
 [ 0.00297424 -0.0281784  -0.06881659 ... -0.04047417  0.00457048
   0.05696651]]
<NDArray 256x20 @cpu(0)>
```

Note that the weights are nonzero. This is by design since they were randomly initialized when we constructed the network. `data` is not the only function that we can invoke. For instance, we can compute the gradient with respect to the parameters. It has the same shape as the weight. However, since we did not invoke backpropagation yet, the values are all 0.

```
In [5]: net[0].weight.grad()
```

```
Out[5]:
        [[0. 0. 0. ... 0. 0. 0.]
         [0. 0. 0. ... 0. 0. 0.]
         [0. 0. 0. ... 0. 0. 0.]
         ...
         [0. 0. 0. ... 0. 0. 0.]
         [0. 0. 0. ... 0. 0. 0.]
         [0. 0. 0. ... 0. 0. 0.]]
        <NDArray 256x20 @cpu(0)>
```

All Parameters at Once

Accessing parameters as described above can be a bit tedious, in particular if we have more complex blocks, or blocks of blocks (or even blocks of blocks of blocks), since we need to walk through the entire tree in reverse order to how the blocks were constructed. To avoid this, blocks come with a method `collect_params` which grabs all parameters of a network in one dictionary such that we can traverse it with ease. It does so by iterating over all constituents of a block and calls `collect_params` on subblocks as needed. To see the difference consider the following:

```
In [6]: # parameters only for the first layer
        print(net[0].collect_params())
        # parameters of the entire network
        print(net.collect_params())

dense0_ (
  Parameter dense0_weight (shape=(256, 20), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
)
sequential0_ (
  Parameter dense0_weight (shape=(256, 20), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
  Parameter dense1_weight (shape=(10, 256), dtype=float32)
  Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

This provides us with a third way of accessing the parameters of the network. If we wanted to get the value of the bias term of the second layer we could simply use this:

```
In [7]: net.collect_params()['dense1_bias'].data()
Out[7]:
        [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
        <NDArray 10 @cpu(0)>
```

Throughout the book we'll see how various blocks name their subblocks (Sequential simply numbers them). This makes it very convenient to use regular expressions to filter out the required parameters.

```
In [8]: print(net.collect_params('.*weight'))
        print(net.collect_params('dense0.*'))
sequential0_ (
  Parameter dense0_weight (shape=(256, 20), dtype=float32)
  Parameter dense1_weight (shape=(10, 256), dtype=float32)
)
sequential0_ (
  Parameter dense0_weight (shape=(256, 20), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
)
```

Rube Goldberg strikes again

Let's see how the parameter naming conventions work if we nest multiple blocks inside each other. For that we first define a function that produces blocks (a block factory, so to speak) and then we combine these inside yet larger blocks.

```
In [9]: def block1():
            net = nn.Sequential()
            net.add(nn.Dense(32, activation='relu'))
            net.add(nn.Dense(16, activation='relu'))
            return net

        def block2():
            net = nn.Sequential()
            for i in range(4):
                net.add(block1())
            return net

        rgnet = nn.Sequential()
        rgnet.add(block2())
        rgnet.add(nn.Dense(10))
        rgnet.initialize()
        rgnet(x)
Out[9]:
        [[ 1.0116727e-08 -9.4839003e-10 -1.1526797e-08  1.4917443e-08
          -1.5690811e-09 -3.9257650e-09 -4.1441655e-09  9.3013472e-09
           3.2393586e-09 -4.8612452e-09]
         [ 9.0111598e-09 -1.9115812e-10 -8.9595842e-09  1.0745880e-08
           1.4963460e-10 -2.2272872e-09 -3.9153973e-09  7.0595711e-09
           3.4854222e-09 -4.5807327e-09]]
        <NDArray 2x10 @cpu(0)>
```

Now that we are done designing the network, let's see how it is organized. `collect_params` provides us with this information, both in terms of naming and in terms of logical structure.

```
In [10]: print(rgnet.collect_params)
         print(rgnet.collect_params())

<bound method Block.collect_params of Sequential(
  (0): Sequential(
    (0): Sequential(
      (0): Dense(20 -> 32, Activation(relu))
      (1): Dense(32 -> 16, Activation(relu))
    )
    (1): Sequential(
      (0): Dense(16 -> 32, Activation(relu))
      (1): Dense(32 -> 16, Activation(relu))
    )
    (2): Sequential(
      (0): Dense(16 -> 32, Activation(relu))
      (1): Dense(32 -> 16, Activation(relu))
    )
    (3): Sequential(
      (0): Dense(16 -> 32, Activation(relu))
      (1): Dense(32 -> 16, Activation(relu))
    )
  )
  (1): Dense(16 -> 10, linear)
)>
sequential1_ (
  Parameter dense2_weight (shape=(32, 20), dtype=float32)
  Parameter dense2_bias (shape=(32,), dtype=float32)
  Parameter dense3_weight (shape=(16, 32), dtype=float32)
  Parameter dense3_bias (shape=(16,), dtype=float32)
  Parameter dense4_weight (shape=(32, 16), dtype=float32)
  Parameter dense4_bias (shape=(32,), dtype=float32)
  Parameter dense5_weight (shape=(16, 32), dtype=float32)
  Parameter dense5_bias (shape=(16,), dtype=float32)
  Parameter dense6_weight (shape=(32, 16), dtype=float32)
  Parameter dense6_bias (shape=(32,), dtype=float32)
  Parameter dense7_weight (shape=(16, 32), dtype=float32)
  Parameter dense7_bias (shape=(16,), dtype=float32)
  Parameter dense8_weight (shape=(32, 16), dtype=float32)
  Parameter dense8_bias (shape=(32,), dtype=float32)
  Parameter dense9_weight (shape=(16, 32), dtype=float32)
  Parameter dense9_bias (shape=(16,), dtype=float32)
  Parameter dense10_weight (shape=(10, 16), dtype=float32)
  Parameter dense10_bias (shape=(10,), dtype=float32)
)
```

Since the layers are hierarchically generated, we can also access them accordingly. For instance, to access the first major block, within it the second subblock and then within it, in turn the bias of the first layer, we perform the following.

```
In [11]: rgnet[0][1][0].bias.data()
Out[11]:
        [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
         0. 0. 0. 0. 0. 0. 0. 0.]
```

```
<NDArray 32 @cpu(0)>
```

5.2.2 Parameter Initialization

Now that we know how to access the parameters, let's look at how to initialize them properly. We discussed the need for Initialization in the previous chapter. By default, MXNet initializes the weight matrices uniformly by drawing from $U[-0.07, 0.07]$ and the bias parameters are all set to 0. However, we often need to use other methods to initialize the weights. MXNet's `init` module provides a variety of preset initialization methods, but if we want something out of the ordinary, we need a bit of extra work.

Built-in Initialization

Let's begin with the built-in initializers. The code below initializes all parameters with Gaussian random variables.

```
In [12]: # force_reinit ensures that the variables are initialized again, regardless of
         # whether they were already initialized previously
         net.initialize(init=init.Normal(sigma=0.01), force_reinit=True)
         net[0].weight.data()[0]

Out[12]:
         [-0.008166  -0.00159167 -0.00273115  0.00684697  0.01204039  0.01359703
           0.00776908 -0.00640936  0.00256858  0.00545601  0.0018105  -0.00914027
           0.00133803  0.01070259 -0.00368285  0.01432678  0.00558631 -0.01479764
           0.00879013  0.00460165]
         <NDArray 20 @cpu(0)>
```

If we wanted to initialize all parameters to 1, we could do this simply by changing the initializer to `Constant(1)`.

```
In [13]: net.initialize(init=init.Constant(1), force_reinit=True)
         net[0].weight.data()[0]

Out[13]:
         [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
         <NDArray 20 @cpu(0)>
```

If we want to initialize only a specific parameter in a different manner, we can simply set the initializer only for the appropriate subblock (or parameter) for that matter. For instance, below we initialize the second layer to a constant value of 42 and we use the `Xavier` initializer for the weights of the first layer.

```
In [14]: net[1].initialize(init=init.Constant(42), force_reinit=True)
         net[0].weight.initialize(init=init.Xavier(), force_reinit=True)
         print(net[1].weight.data()[0,0])
         print(net[0].weight.data()[0])


[42.]
<NDArray 1 @cpu(0)>

[-0.14511706 -0.01173057 -0.03754489 -0.14020921  0.00900492  0.01712246
  0.12447387 -0.04094418 -0.12105145  0.00079902 -0.0277361  -0.10213967
```

```
 -0.14027238 -0.02196661 -0.04641148  0.11977354  0.03604397 -0.14493202
 -0.06514931  0.13826048]
<NDArray 20 @cpu(0)>
```

Custom Initialization

Sometimes, the initialization methods we need are not provided in the `init` module. At this point, we can implement a subclass of the `Initializer` class so that we can use it like any other initialization method. Usually, we only need to implement the `_init_weight` function and modify the incoming NDArray according to the initial result. In the example below, we pick a decidedly bizarre and nontrivial distribution, just to prove the point. We draw the coefficients from the following distribution:

$$w \sim \begin{cases} U[5, 10] & \text{with probability } \frac{1}{4} \\ 0 & \text{with probability } \frac{1}{2} \\ U[-10, -5] & \text{with probability } \frac{1}{4} \end{cases}$$

```
In [15]: class MyInit(init.Initializer):
             def _init_weight(self, name, data):
                 print('Init', name, data.shape)
                 data[:] = nd.random.uniform(low=-10, high=10, shape=data.shape)
                 data *= data.abs() >= 5

         net.initialize(MyInit(), force_reinit=True)
         net[0].weight.data()[0]

Init dense0_weight (256, 20)
Init dense1_weight (10, 256)

Out[15]:
         [-5.44481   6.536484  -0.        0.        0.        7.7452965
          7.739216  7.6021366  0.       -0.       -7.3307705 -0.
          9.611603  0.         7.4357147  0.        0.       -0.
          8.446959  0.       ]
         <NDArray 20 @cpu(0)>
```

If even this functionality is insufficient, we can set parameters directly. Since `data()` returns an NDArray we can access it just like any other matrix. A note for advanced users - if you want to adjust parameters within an `autograd` scope you need to use `set_data` to avoid confusing the automatic differentiation mechanics.

```
In [16]: net[0].weight.data()[:] += 1
         net[0].weight.data()[0,0] = 42
         net[0].weight.data()[0]

Out[16]:
         [42.        7.536484  1.        1.        1.        8.7452965
          8.739216  8.602137  1.        1.       -6.3307705 1.
          10.611603 1.        8.435715  1.        1.        1.
          9.446959  1.       ]
         <NDArray 20 @cpu(0)>
```

5.2.3 Tied Parameters

In some cases, we want to share model parameters across multiple layers. For instance when we want to find good word embeddings we may decide to use the same parameters both for encoding and decoding of words. We discussed one such case when we introduced *Blocks*. Let's see how to do this a bit more elegantly. In the following we allocate a dense layer and then use its parameters specifically to set those of another layer.

```
In [17]: net = nn.Sequential()
         # We need to give the shared layer a name such that we can reference its
         # parameters
         shared = nn.Dense(8, activation='relu')
         net.add(nn.Dense(8, activation='relu'),
                 shared,
                 nn.Dense(8, activation='relu', params=shared.params),
                 nn.Dense(10))
         net.initialize()

         x = nd.random.uniform(shape=(2, 20))
         net(x)

         # Check whether the parameters are the same
         print(net[1].weight.data()[0] == net[2].weight.data()[0])
         net[1].weight.data()[0,0] = 100
         # Make sure that they're actually the same object rather than just having the
         # same value
         print(net[1].weight.data()[0] == net[2].weight.data()[0])

[1. 1. 1. 1. 1. 1. 1. 1.]
<NDArray 8 @cpu(0)>

[1. 1. 1. 1. 1. 1. 1. 1.]
<NDArray 8 @cpu(0)>
```

The above example shows that the parameters of the second and third layer are tied. They are identical rather than just being equal. That is, by changing one of the parameters the other one changes, too. What happens to the gradients is quite ingenious. Since the model parameters contain gradients, the gradients of the second hidden layer and the third hidden layer are accumulated in the `shared.params.grad()` during backpropagation.

Summary

- We have several ways to access, initialize, and tie model parameters.
- We can use custom initialization.
- Gluon has a sophisticated mechanism for accessing parameters in a unique and hierarchical manner.

Exercises

1. Use the FancyMLP definition of the *previous section* and access the parameters of the various layers.

2. Look at the MXNet documentation and explore different initializers.

3. Try accessing the model parameters after `net.initialize()` and before `net(x)` to observe the shape of the model parameters. What changes? Why?

4. Construct a multilayer perceptron containing a shared parameter layer and train it. During the training process, observe the model parameters and gradients of each layer.

5. Why is sharing parameters a good idea?

Scan the QR Code to Discuss

5.3 Deferred Initialization

In the previous examples we played fast and loose with setting up our networks. In particular we did the following things that *shouldn't* work:

- We defined the network architecture with no regard to the input dimensionality.

- We added layers without regard to the output dimension of the previous layer.

- We even initialized' these parameters without knowing how many parameters were were to initialize.

All of those things sound impossible and indeed, they are. After all, there's no way MXNet (or any other framework for that matter) could predict what the input dimensionality of a network would be. Later on, when working with convolutional networks and images this problem will become even more pertinent, since the input dimensionality (i.e. the resolution of an image) will affect the dimensionality of subsequent layers at a long range. Hence, the ability to set parameters without the need to know at the time of writing the code what the dimensionality is can greatly simplify statistical modeling. In what follows, we will discuss how this works using initialization as an example. After all, we cannot initialize variables that we don't know exist.

5.3.1 Instantiating a Network

Let's see what happens when we instantiate a network. We start with our trusty MLP as before.

```
In [1]: from mxnet import init, nd
        from mxnet.gluon import nn

        def getnet():
            net = nn.Sequential()
            net.add(nn.Dense(256, activation='relu'))
            net.add(nn.Dense(10))
            return net

        net = getnet()
```

At this point the network doesn't really know yet what the dimensionalities of the various parameters should be. All one could tell at this point is that each layer needs weights and bias, albeit of unspecified dimensionality. If we try accessing the parameters, that's exactly what happens.

```
In [2]: print(net.collect_params)
        print(net.collect_params())

<bound method Block.collect_params of Sequential(
  (0): Dense(None -> 256, Activation(relu))
  (1): Dense(None -> 10, linear)
)>
sequential0_ (
  Parameter dense0_weight (shape=(256, 0), dtype=float32)
  Parameter dense0_bias (shape=(256,), dtype=float32)
  Parameter dense1_weight (shape=(10, 0), dtype=float32)
  Parameter dense1_bias (shape=(10,), dtype=float32)
)
```

In particular, trying to access `net[0].weight.data()` at this point would trigger a runtime error stating that the network needs initializing before it can do anything. Let's see whether anything changes after we initialize the parameters:

```
In [3]: net.initialize()
        net.collect_params()

Out[3]: sequential0_ (
            Parameter dense0_weight (shape=(256, 0), dtype=float32)
            Parameter dense0_bias (shape=(256,), dtype=float32)
            Parameter dense1_weight (shape=(10, 0), dtype=float32)
            Parameter dense1_bias (shape=(10,), dtype=float32)
        )
```

As we can see, nothing really changed. Only once we provide the network with some data do we see a difference. Let's try it out.

```
In [4]: x = nd.random.uniform(shape=(2, 20))
        net(x)  # Forward computation

        net.collect_params()

Out[4]: sequential0_ (
            Parameter dense0_weight (shape=(256, 20), dtype=float32)
```

```
        Parameter dense0_bias (shape=(256,), dtype=float32)
        Parameter dense1_weight (shape=(10, 256), dtype=float32)
        Parameter dense1_bias (shape=(10,), dtype=float32)
    )
```

The main difference to before is that as soon as we knew the input dimensionality, $\mathbf{x} \in \mathbb{R}^{20}$ it was possible to define the weight matrix for the first layer, i.e. $\mathbf{W}_1 \in \mathbb{R}^{256 \times 20}$. With that out of the way, we can progress to the second layer, define its dimensionality to be 10×256 and so on through the computational graph and bind all the dimensions as they become available. Once this is known, we can proceed by initializing parameters. This is the solution to the three problems outlined above.

5.3.2 Deferred Initialization in Practice

Now that we know how it works in theory, let's see when the initialization is actually triggered. In order to do so, we mock up an initializer which does nothing but report a debug message stating when it was invoked and with which parameters.

```
In [5]: class MyInit(init.Initializer):
            def _init_weight(self, name, data):
                print('Init', name, data.shape)
                # The actual initialization logic is omitted here

        net = getnet()
        net.initialize(init=MyInit())
```

Note that, although `MyInit` will print information about the model parameters when it is called, the above `initialize` function does not print any information after it has been executed. Therefore there is no real initialization parameter when calling the `initialize` function. Next, we define the input and perform a forward calculation.

```
In [6]: x = nd.random.uniform(shape=(2, 20))
        y = net(x)

Init dense2_weight (256, 20)
Init dense3_weight (10, 256)
```

At this time, information on the model parameters is printed. When performing a forward calculation based on the input `x`, the system can automatically infer the shape of the weight parameters of all layers based on the shape of the input. Once the system has created these parameters, it calls the `MyInit` instance to initialize them before proceeding to the forward calculation.

Of course, this initialization will only be called when completing the initial forward calculation. After that, we will not re-initialize when we run the forward calculation `net(x)`, so the output of the `MyInit` instance will not be generated again.

```
In [7]: y = net(x)
```

As mentioned at the beginning of this section, deferred initialization can also cause confusion. Before the first forward calculation, we were unable to directly manipulate the model parameters, for example, we could not use the `data` and `set_data` functions to get and modify the parameters. Therefore, we often force initialization by sending a sample observation through the network.

5.3.3 Forced Initialization

Deferred initialization does not occur if the system knows the shape of all parameters when calling the `initialize` function. This can occur in two cases:

- We've already seen some data and we just want to reset the parameters.

- We specified all input and output dimensions of the network when defining it.

The first case works just fine, as illustrated below.

```
In [8]: net.initialize(init=MyInit(), force_reinit=True)

Init dense2_weight (256, 20)
Init dense3_weight (10, 256)
```

The second case requires us to specify the remaining set of parameters when creating the layer. For instance, for dense layers we also need to specify the `in_units` so that initialization can occur immediately once `initialize` is called.

```
In [9]: net = nn.Sequential()
        net.add(nn.Dense(256, in_units=20, activation='relu'))
        net.add(nn.Dense(10, in_units=256))

        net.initialize(init=MyInit())

Init dense4_weight (256, 20)
Init dense5_weight (10, 256)
```

Summary

- Deferred initialization is a good thing. It allows Gluon to set many things automagically and it removes a great source of errors from defining novel network architectures.

- We can override this by specifying all implicitly defined variables.

- Initialization can be repeated (or forced) by setting the `force_reinit=True` flag.

Exercises

1. What happens if you specify only parts of the input dimensions. Do you still get immediate initialization?

2. What happens if you specify mismatching dimensions?

3. What would you need to do if you have input of varying dimensionality? Hint - look at parameter tying.

5.4 Custom Layers

One of the reasons for the success of deep learning can be found in the wide range of layers that can be used in a deep network. This allows for a tremendous degree of customization and adaptation. For instance, scientists have invented layers for images, text, pooling, loops, dynamic programming, even for computer programs. Sooner or later you will encounter a layer that doesn't exist yet in Gluon, or even better, you will eventually invent a new layer that works well for your problem at hand. This is when it's time to build a custom layer. This section shows you how.

5.4.1 Layers without Parameters

Since this is slightly intricate, we start with a custom layer (aka Block) that doesn't have any inherent parameters. Our first step is very similar to when we *introduced blocks* previously. The following `CenteredLayer` class constructs a layer that subtracts the mean from the input. We build it by inheriting from the Block class and implementing the `forward` method.

```
In [1]: from mxnet import gluon, nd
        from mxnet.gluon import nn

        class CenteredLayer(nn.Block):
            def __init__(self, **kwargs):
                super(CenteredLayer, self).__init__(**kwargs)

            def forward(self, x):
                return x - x.mean()
```

To see how it works let's feed some data into the layer.

```
In [2]: layer = CenteredLayer()
        layer(nd.array([1, 2, 3, 4, 5]))

Out[2]:
        [-2. -1.  0.  1.  2.]
        <NDArray 5 @cpu(0)>
```

We can also use it to construct more complex models.

```
In [3]: net = nn.Sequential()
        net.add(nn.Dense(128), CenteredLayer())
        net.initialize()
```

Let's see whether the centering layer did its job. For that we send random data through the network and check whether the mean vanishes. Note that since we're dealing with floating point numbers, we're going to see a very small albeit typically nonzero number.

```
In [4]: y = net(nd.random.uniform(shape=(4, 8)))
        y.mean().asscalar()
```

```
Out[4]: -7.212293e-10
```

5.4.2 Layers with Parameters

Now that we know how to define layers in principle, let's define layers with parameters. These can be adjusted through training. In order to simplify things for an avid deep learning researcher the `Parameter` class and the `ParameterDict` dictionary provide some basic housekeeping functionality. In particular, they govern access, initialization, sharing, saving and loading model parameters. For instance, this way we don't need to write custom serialization routines for each new custom layer.

For instance, we can use the member variable `params` of the `ParameterDict` type that comes with the Block class. It is a dictionary that maps string type parameter names to model parameters in the `Parameter` type. We can create a `Parameter` instance from `ParameterDict` via the `get` function.

```
In [5]: params = gluon.ParameterDict()
        params.get('param2', shape=(2, 3))
        params
```

```
Out[5]: (
          Parameter param2 (shape=(2, 3), dtype=<class 'numpy.float32'>)
        )
```

Let's use this to implement our own version of the dense layer. It has two parameters - bias and weight. To make it a bit nonstandard, we bake in the ReLu activation as default. Next, we implement a fully connected layer with both weight and bias parameters. It uses ReLU as an activation function, where `in_units` and `units` are the number of inputs and the number of outputs, respectively.

```
In [6]: class MyDense(nn.Block):
            # units: the number of outputs in this layer; in_units: the number of
            # inputs in this layer
            def __init__(self, units, in_units, **kwargs):
                super(MyDense, self).__init__(**kwargs)
                self.weight = self.params.get('weight', shape=(in_units, units))
                self.bias = self.params.get('bias', shape=(units,))

            def forward(self, x):
                linear = nd.dot(x, self.weight.data()) + self.bias.data()
                return nd.relu(linear)
```

Naming the parameters allows us to access them by name through dictionary lookup later. It's a good idea to give them instructive names. Next, we instantiate the `MyDense` class and access its model parameters.

```
In [7]: dense = MyDense(units=3, in_units=5)
        dense.params
```

```
Out[7]: mydense0_ (
          Parameter mydense0_weight (shape=(5, 3), dtype=<class 'numpy.float32'>)
```

```
          Parameter mydense0_bias (shape=(3,), dtype=<class 'numpy.float32'>)
        )
```

We can directly carry out forward calculations using custom layers.

```
In [8]: dense.initialize()
        dense(nd.random.uniform(shape=(2, 5)))
Out[8]:
        [[0.06917784 0.01627153 0.01029644]
         [0.02602214 0.0453731  0.        ]]
        <NDArray 2x3 @cpu(0)>
```

We can also construct models using custom layers. Once we have that we can use it just like the built-in dense layer. The only exception is that in our case size inference is not automagic. Please consult the MXNet documentation for details on how to do this.

```
In [9]: net = nn.Sequential()
        net.add(MyDense(8, in_units=64),
                MyDense(1, in_units=8))
        net.initialize()
        net(nd.random.uniform(shape=(2, 64)))
Out[9]:
        [[0.03820474]
         [0.04035058]]
        <NDArray 2x1 @cpu(0)>
```

Summary

- We can design custom layers via the Block class. This is more powerful than defining a block factory, since it can be invoked in many contexts.
- Blocks can have local parameters.

Exercises

1. Design a layer that learns an affine transform of the data, i.e. it removes the mean and learns an additive parameter instead.

2. Design a layer that takes an input and computes a tensor reduction, i.e. it returns $y_k = \sum_{i,j} W_{ijk} x_i x_j$.

3. Design a layer that returns the leading half of the Fourier coefficients of the data. Hint - look up the `fft` function in MXNet.

Scan the QR Code to Discuss

5.5 File I/O

So far we discussed how to process data, how to build, train and test deep learning models. However, at some point we are likely happy with what we obtained and we want to save the results for later use and distribution. Likewise, when running a long training process it is best practice to save intermediate results (checkpointing) to ensure that we don't lose several days worth of computation when tripping over the power cord of our server. At the same time, we might want to load a pretrained model (e.g. we might have word embeddings for English and use it for our fancy spam classifier). For all of these cases we need to load and store both individual weight vectors and entire models. This section addresses both issues.

5.5.1 NDArray

In its simplest form, we can directly use the `save` and `load` functions to store and read NDArrays separately. This works just as expected.

```
In [1]: from mxnet import nd
        from mxnet.gluon import nn

        x = nd.arange(4)
        nd.save('x-file', x)
```

Then, we read the data from the stored file back into memory.

```
In [2]: x2 = nd.load('x-file')
        x2
Out[2]: [
         [0. 1. 2. 3.]
         <NDArray 4 @cpu(0)>]
```

We can also store a list of NDArrays and read them back into memory.

```
In [3]: y = nd.zeros(4)
        nd.save('x-files', [x, y])
        x2, y2 = nd.load('x-files')
        (x2, y2)
Out[3]: (
         [0. 1. 2. 3.]
         <NDArray 4 @cpu(0)>,
```

```
      [0. 0. 0. 0.]
      <NDArray 4 @cpu(0)>)
```

We can even write and read a dictionary that maps from a string to an NDArray. This is convenient, for instance when we want to read or write all the weights in a model.

```
In [4]: mydict = {'x': x, 'y': y}
        nd.save('mydict', mydict)
        mydict2 = nd.load('mydict')
        mydict2
Out[4]: {'x':
         [0. 1. 2. 3.]
         <NDArray 4 @cpu(0)>, 'y':
         [0. 0. 0. 0.]
         <NDArray 4 @cpu(0)>}
```

5.5.2 Gluon Model Parameters

Saving individual weight vectors (or other NDArray tensors) is useful but it gets very tedious if we want to save (and later load) an entire model. After all, we might have hundreds of parameter groups sprinkled throughout. Writing a script that collects all the terms and matches them to an architecture is quite some work. For this reason Gluon provides built-in functionality to load and save entire networks rather than just single weight vectors. An important detail to note is that this saves model *parameters* and not the entire model. I.e. if we have a 3 layer MLP we need to specify the *architecture* separately. The reason for this is that the models themselves can contain arbitrary code, hence they cannot be serialized quite so easily (there is a way to do this for compiled models - please refer to the MXNet documentation for the technical details on it). The result is that in order to reinstate a model we need to generate the architecture in code and then load the parameters from disk. The *deferred initialization* is quite advantageous here since we can simply define a model without the need to put actual values in place. Let's start with our favorite MLP.

```
In [5]: class MLP(nn.Block):
            def __init__(self, **kwargs):
                super(MLP, self).__init__(**kwargs)
                self.hidden = nn.Dense(256, activation='relu')
                self.output = nn.Dense(10)

            def forward(self, x):
                return self.output(self.hidden(x))

        net = MLP()
        net.initialize()
        x = nd.random.uniform(shape=(2, 20))
        y = net(x)
```

Next, we store the parameters of the model as a file with the name mlp.params'.

```
In [6]: net.save_parameters('mlp.params')
```

To check whether we are able to recover the model we instantiate a clone of the original MLP model. Unlike the random initialization of model parameters, here we read the parameters stored in the file

---

directly.

```
In [7]: clone = MLP()
        clone.load_parameters('mlp.params')
```

Since both instances have the same model parameters, the computation result of the same input x should
be the same. Let's verify this.

```
In [8]: yclone = clone(x)
        yclone == y
Out[8]:
        [[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
         [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]]
        <NDArray 2x10 @cpu(0)>
```

Summary

- The `save` and `load` functions can be used to perform File I/O for NDArray objects.
- The `load_parameters` and `save_parameters` functions allow us to save entire sets of
  parameters for a network in Gluon.
- Saving the architecture has to be done in code rather than in parameters.

Exercises

1. Even if there is no need to deploy trained models to a different device, what are the practical benefits
   of storing model parameters?

2. Assume that we want to reuse only parts of a network to be incorporated into a network of a *different*
   architecture. How would you go about using, say the first two layers from a previous network in a
   new network.

3. How would you go about saving network architecture and parameters? What restrictions would
   you impose on the architecture?

Scan the QR Code to Discuss

5.6 GPUs

In the introduction to this book we discussed the rapid growth of computation over the past two decades. In a nutshell, GPU performance has increased by a factor of 1000 every decade since 2000. This offers great opportunity but it also suggests a significant need to provide such performance.

Decade	Dataset	Memory	Floating Point Calculations per Second
1970	100 (Iris)	1 KB	100 KF (Intel 8080)
1980	1 K (House prices in Boston)	100 KB	1 MF (Intel 80186)
1990	10 K (optical character recognition)	10 MB	10 MF (Intel 80486)
2000	10 M (web pages)	100 MB	1 GF (Intel Core)
2010	10 G (advertising)	1 GB	1 TF (NVIDIA C2050)
2020	1 T (social network)	100 GB	1 PF (NVIDIA DGX-2)

In this section we begin to discuss how to harness this compute performance for your research. First by using single GPUs and at a later point, how to use multiple GPUs and multiple servers (with multiple GPUs). You might have noticed that MXNet NDArray looks almost identical to NumPy. But there are a few crucial differences. One of the key features that differentiates MXNet from NumPy is its support for diverse hardware devices.

In MXNet, every array has a context. In fact, whenever we displayed an NDArray so far, it added a cryptic `@cpu(0)` notice to the output which remained unexplained so far. As we will discover, this just indicates that the computation is being executed on the CPU. Other contexts might be various GPUs. Things can get even hairier when we deploy jobs across multiple servers. By assigning arrays to contexts intelligently, we can minimize the time spent transferring data between devices. For example, when training neural networks on a server with a GPU, we typically prefer for the model's parameters to live on the GPU.

In short, for complex neural networks and large-scale data, using only CPUs for computation may be inefficient. In this section, we will discuss how to use a single NVIDIA GPU for calculations. First, make sure you have at least one NVIDIA GPU installed. Then, download CUDA and follow the prompts to set the appropriate path. Once these preparations are complete, the `nvidia-smi` command can be used to view the graphics card information.

```
In [1]: !nvidia-smi

Mon Mar 11 22:59:12 2019
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 396.37                 Driver Version: 396.37                     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla M60           Off  | 00000000:00:1D.0 Off |                    0 |
| N/A   34C    P0    38W / 150W |   7044MiB /  7618MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla M60           Off  | 00000000:00:1E.0 Off |                    0 |
```

```
| N/A   37C   P8    14W / 150W |     11MiB /  7618MiB |      0%      Default |
+-----------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|    0    103927     C   /home/ubuntu/miniconda2/bin/python           7033MiB |
+-----------------------------------------------------------------------------+
```

Next, we need to confirm that the GPU version of MXNet is installed. If a CPU version of MXNet is already installed, we need to uninstall it first. For example, use the `pip uninstall mxnet` command, then install the corresponding MXNet version according to the CUDA version. Assuming you have CUDA 9.0 installed, you can install the MXNet version that supports CUDA 9.0 by `pip install mxnet-cu90`. To run the programs in this section, you need at least two GPUs.

Note that this might be extravagant for most desktop computers but it is easily available in the cloud, e.g. by using the AWS EC2 multi-GPU instances. Almost all other sections do *not* require multiple GPUs. Instead, this is simply to illustrate how data flows between different devices.

5.6.1 Computing Devices

MXNet can specify devices, such as CPUs and GPUs, for storage and calculation. By default, MXNet creates data in the main memory and then uses the CPU to calculate it. In MXNet, the CPU and GPU can be indicated by `cpu()` and `gpu()`. It should be noted that `mx.cpu()` (or any integer in the parentheses) means all physical CPUs and memory. This means that MXNet's calculations will try to use all CPU cores. However, `mx.gpu()` only represents one graphic card and the corresponding graphic memory. If there are multiple GPUs, we use `mx.gpu(i)` to represent the i-th GPU (i starts from 0). Also, `mx.gpu(0)` and `mx.gpu()` are equivalent.

```
In [2]: import mxnet as mx
        from mxnet import nd
        from mxnet.gluon import nn

        mx.cpu(), mx.gpu(), mx.gpu(1)
Out[2]: (cpu(0), gpu(0), gpu(1))
```

5.6.2 NDArray and GPUs

By default, NDArray objects are created on the CPU. Therefore, we will see the `@cpu(0)` identifier each time we print an NDArray.

```
In [3]: x = nd.array([1, 2, 3])
        x
Out[3]:
        [1. 2. 3.]
        <NDArray 3 @cpu(0)>
```

We can use the `context` property of NDArray to view the device where the NDArray is located. It is important to note that whenever we want to operate on multiple terms they need to be in the same context. For instance, if we sum two variables, we need to make sure that both arguments are on the same device - otherwise MXNet would not know where to store the result or even how to decide where to perform the computation.

```
In [4]: x.context

Out[4]: cpu(0)
```

Storage on the GPU

There are several ways to store an NDArray on the GPU. For example, we can specify a storage device with the `ctx` parameter when creating an NDArray. Next, we create the NDArray variable `a` on `gpu(0)`. Notice that when printing `a`, the device information becomes `@gpu(0)`. The NDArray created on a GPU only consumes the memory of this GPU. We can use the `nvidia-smi` command to view GPU memory usage. In general, we need to make sure we do not create data that exceeds the GPU memory limit.

```
In [5]: x = nd.ones((2, 3), ctx=mx.gpu())
        x

Out[5]:
        [[1. 1. 1.]
         [1. 1. 1.]]
        <NDArray 2x3 @gpu(0)>
```

Assuming you have at least two GPUs, the following code will create a random array on `gpu(1)`.

```
In [6]: y = nd.random.uniform(shape=(2, 3), ctx=mx.gpu(1))
        y

Out[6]:
        [[0.59119    0.313164   0.76352036]
         [0.9731786  0.35454726 0.11677533]]
        <NDArray 2x3 @gpu(1)>
```

Copying

If we want to compute $\mathbf{x} + \mathbf{y}$ we need to decide where to perform this operation. For instance, we can transfer $\mathbf{x}$ to `gpu(1)` and perform the operation there. **Do not** simply add `x + y` since this will result in an exception. The runtime engine wouldn't know what to do, it cannot find data on the same device and it fails.
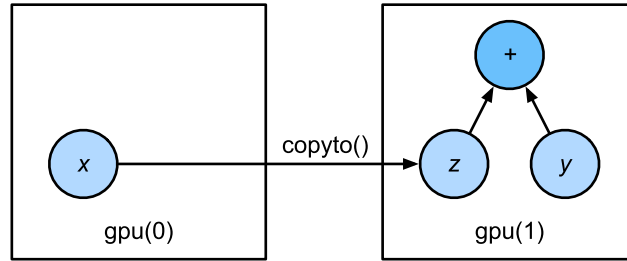
Fig. 5.2: Copyto copies arrays to the target device

`copyto` copies the data to another device such that we can add them. Since **y** lives on the second GPU we need to move **x** there before we can add the two.

```
In [7]: z = x.copyto(mx.gpu(1))
        print(x)
        print(z)


[[1. 1. 1.]
 [1. 1. 1.]]
<NDArray 2x3 @gpu(0)>

[[1. 1. 1.]
 [1. 1. 1.]]
<NDArray 2x3 @gpu(1)>
```

Now that the data is on the same GPU (both **z** and **y** are), we can add them up. In such cases MXNet places the result on the same device as its constituents. In our case that is `@gpu(1)`.

```
In [8]: y + z

Out[8]:
        [[1.59119   1.313164  1.7635204]
         [1.9731786 1.3545473 1.1167753]]
        <NDArray 2x3 @gpu(1)>
```

Imagine that your variable z already lives on your second GPU (gpu(0)). What happens if we call z.copyto(gpu(0))? It will make a copy and allocate new memory, even though that variable already lives on the desired device! There are times where depending on the environment our code is running in, two variables may already live on the same device. So we only want to make a copy if the variables currently lives on different contexts. In these cases, we can call `as_in_context()`. If the variable is already the specified context then this is a no-op. In fact, unless you specifically want to make a copy, `as_in_context()` is the method of choice.

```
In [9]: z = x.as_in_context(mx.gpu(1))
        z

Out[9]:
        [[1. 1. 1.]
         [1. 1. 1.]]
        <NDArray 2x3 @gpu(1)>
```

It is important to note that, if the `context` of the source variable and the target variable are consistent, then the `as_in_context` function causes the target variable and the source variable to share the memory of the source variable.

```
In [10]: y.as_in_context(mx.gpu(1)) is y
Out[10]: True
```

The `copyto` function always creates new memory for the target variable.

```
In [11]: y.copyto(mx.gpu()) is y
Out[11]: False
```

Watch Out

People use GPUs to do machine learning because they expect them to be fast. But transferring variables between contexts is slow. So we want you to be 100% certain that you want to do something slow before we let you do it. If MXNet just did the copy automatically without crashing then you might not realize that you had written some slow code.

Also, transferring data between devices (CPU, GPUs, other machines) is something that is *much slower* than computation. It also makes parallelization a lot more difficult, since we have to wait for data to be sent (or rather to be received) before we can proceed with more operations. This is why copy operations should be taken with great care. As a rule of thumb, many small operations are much worse than one big operation. Moreover, several operations at a time are much better than many single operations interspersed in the code (unless you know what you're doing). This is the case since such operations can block if one device has to wait for the other before it can do something else. It's a bit like ordering your coffee in a queue rather than pre-ordering it by phone and finding out that it's ready when you are.

Lastly, when we print NDArray data or convert NDArrays to NumPy format, if the data is not in main memory, MXNet will copy it to the main memory first, resulting in additional transmission overhead. Even worse, it is now subject to the dreaded Global Interpreter Lock which makes everything wait for Python to complete.

5.6.3 Gluon and GPUs

Similar to NDArray, Gluon's model can specify devices through the `ctx` parameter during initialization. The following code initializes the model parameters on the GPU (we will see many more examples of how to run models on GPUs in the following, simply since they will become somewhat more compute intensive).

```
In [12]: net = nn.Sequential()
         net.add(nn.Dense(1))
         net.initialize(ctx=mx.gpu())
```

When the input is an NDArray on the GPU, Gluon will calculate the result on the same GPU.

```
In [13]: net(x)
```

```
Out[13]:
        [[0.04995865]
         [0.04995865]]
        <NDArray 2x1 @gpu(0)>
```

Let us confirm that the model parameters are stored on the same GPU.

```
In [14]: net[0].weight.data()
```
```
Out[14]:
        [[0.0068339  0.01299825 0.0301265 ]]
        <NDArray 1x3 @gpu(0)>
```

In short, as long as all data and parameters are on the same device, we can learn models efficiently. In the following we will see several such examples.

Summary

- MXNet can specify devices for storage and calculation, such as CPU or GPU. By default, MXNet creates data in the main memory and then uses the CPU to calculate it.

- MXNet requires all input data for calculation to be **on the same device**, be it CPU or the same GPU.

- You can lose significant performance by moving data without care. A typical mistake is as follows: computing the loss for every minibatch on the GPU and reporting it back to the user on the commandline (or logging it in a NumPy array) will trigger a global interpreter lock which stalls all GPUs. It is much better to allocate memory for logging inside the GPU and only move larger logs.

Exercises

1. Try a larger computation task, such as the multiplication of large matrices, and see the difference in speed between the CPU and GPU. What about a task with a small amount of calculations?

2. How should we read and write model parameters on the GPU?

3. Measure the time it takes to compute 1000 matrix-matrix multiplications of 100×100 matrices and log the matrix norm $\mathrm{tr} M M^\top$ one result at a time vs. keeping a log on the GPU and transferring only the final result.

4. Measure how much time it takes to perform two matrix-matrix multiplications on two GPUs at the same time vs. in sequence on one GPU (hint - you should see almost linear scaling).

References

[1] CUDA download address. https://developer.nvidia.com/cuda-downloads

Scan the QR Code to Discuss

6

Convolutional Neural Networks

In this chapter we introduce convolutional neural networks. They are the first nontrivial *architecture* beyond the humble multilayer perceptron. In their design scientists used inspiration from biology, group theory, and lots of experimentation to achieve stunning results in object recognition, segmentation, image synthesis and related computer vision tasks. Convnets', as they are often called, have become a cornerstone for deep learning research. Their applications reach beyond images to audio, text, video, time series analysis, graphs and recommender systems.

We will first describe the operating principles of the convolutional layer and pooling layer in a convolutional neural network, and then explain padding, stride, input channels, and output channels. Next we will explore the design concepts of several representative deep convolutional neural networks. These models include the AlexNet, the first such network proposed, and later networks that use repeating elements (VGG), network in network (NiN), networks with parallel concatenations (GoogLeNet), residual networks (ResNet), and densely connected networks (DenseNet). Many of these networks have led to significant progress in the ImageNet competition (a famous computer vision contest) within the past few years.

Over time the networks have increased in depth significantly, exceeding hundreds of layers. To train on them efficiently tools for capacity control, reparametrization and training acceleration are needed. Batch normalization and residual networks are both used to address these problems. We will describe them in this chapter.

6.1 From Dense Layers to Convolutions

So far we learned the basics of designing Deep Networks. Indeed, for someone dealing only with generic data, the previous sections are probably sufficient to train and deploy such a network sufficiently. There is one caveat, though - just like most problems in statistics, networks with many parameters either require a lot of data or a lot of regularization. As a result we cannot hope to design sophisticated models in most cases.

For instance, consider the seemingly task of distinguishing cats from dogs. We decide to use a good camera and take 1 megapixel photos to ensure that we can really distinguish both species accurately. This means that the *input* into a network has 1 million dimensions. Even an aggressive reduction to 1,000 dimensions after the first layer means that we need 10^9 parameters. Unless we have copious amounts of data (billions of images of cats and dogs), this is mission impossible. Add in subsequent layers and it is clear that this approach is infeasible.

The avid reader might object to the rather absurd implications of this argument by stating that 1 megapixel resolution is not necessary. But even if we reduce it to a paltry 100,000 pixels, that's still 10^8 parameters. A corresponding number of images of training data is beyond the reach of most statisticians. In fact, the number exceeds the population of dogs and cats in all but the largest countries! Yet both humans and computers are able to distinguish cats from dogs quite well, often after only a few hundred images. This seems to contradict our conclusions above. There is clearly something wrong in the way we are approaching the problem. Let's find out.

6.1.1 Invariances

Imagine that you want to detect an object in an image. It is only reasonable to assume that the location of the object shouldn't matter too much to determine whether the object is there. We should assume that we would recognize an object wherever it is in an image. This is true within reason - pigs usually don't fly and planes usually don't swim. Nonetheless, we would still recognize a flying pig, albeit possibly after a double-take. This fact manifests itself e.g. in the form of the children's game Where is Waldo'. In it, the goal is to find a boy with red and white striped clothes, a striped hat and black glasses within a panoply of activity in an image. Despite the rather characteristic outfit this tends to be quite difficult, due to the large amount of confounders. The image below, on the other hand, makes the problem particularly easy.

There are two key principles that we can deduce from this slightly frivolous reasoning:

1. Object detectors should work the same regardless of where in the image an object can be found. In other words, the waldoness' of a location in the image can be assessed (in first approximation) without regard of the position within the image. (Translation Invariance)

2. Object detection can be answered by considering only local information. In other words, the waldoness' of a location can be assessed (in first approximation) without regard of what else happens in the image at large distances. (Locality)

Let's see how this translates into mathematics.

6.1.2 Constraining the MLP

In the following we will treat images and hidden layers as two-dimensional arrays. I.e. $x[i, j]$ and $h[i, j]$ denote the position (i, j) in an image. Consequently we switch from weight matrices to four-dimensional weight tensors. In this case a dense layer can be written as follows:

$$h[i, j] = \sum_{k,l} W[i, j, k, l] \cdot x[k, l] = \sum_{a,b} V[i, j, a, b] \cdot x[i + a, j + b]$$

The switch from W to V is entirely cosmetic (for now) since there is a one to one correspondence between coefficients in both tensors. We simply re-index the subscripts (k, l) such that $k = i + a$ and $l = j + b$. In other words we set $V[i, j, a, b] = W[i, j, i+a, j+b]$. The indices a, b run over both positive and negative offsets, covering the entire image. For any given location (i, j) in the hidden layer $h[i, j]$ we compute its

value by summing over pixels in x, centered around (i, j) and weighted by $V[i, j, a, b]$.

Now let's invoke the first principle we established above - *translation invariance*. This implies that a shift in the inputs x should simply lead to a shift in the activations h. This is only possible if V doesn't actually depend on (i, j), that is, we have $V[i, j, a, b] = V[a, b]$. As a result we can simplify the definition for h.

$$h[i, j] = \sum_{a, b} V[a, b] \cdot x[i + a, j + b]$$

This is a convolution! We are effectively weighting pixels $(i + a, j + b)$ in the vicinity of (i, j) with coefficients $V[a, b]$ to obtain the value $h[i, j]$. Note that $V[a, b]$ needs a lot fewer coefficients than $V[i, j, a, b]$. For a 1 megapixel image it has at most 1 million coefficients. This is 1 million fewer parameters since it no longer depends on the location within the image. We have made significant progress!

Now let's invoke the second principle - *locality*. In the problem of detecting Waldo we shouldn't have to look very far away from (i, j) in order to glean relevant information to assess what is going on at $h[i, j]$. This means that outside some range $|a|, |b| > \Delta$ we should set $V[a, b] = 0$. Equivalently we can simply rewrite $h[i, j]$ as

$$h[i, j] = \sum_{a = -\Delta}^{\Delta} \sum_{b = -\Delta}^{\Delta} V[a, b] \cdot x[i + a, j + b]$$

This, in a nutshell is the convolutional layer. The difference to the fully connected network is dramatic. While previously we might have needed 10^8 or more coefficients, we now only need $O(\Delta^2)$ terms. The price that we pay for this drastic simplification is that our network will be translation invariant and that we are only able to take local information into account.

6.1.3 Convolutions

Let's briefly review why the above operation is called a convolution. In math the convolution between two functions, say $f, g : \mathbb{R}^d \to R$ is defined as

$$[f \circledast g](x) = \int_{\mathbb{R}^d} f(z) g(x - z) dz$$

That is, we measure the overlap beween f and g when both functions are shifted by x and flipped'. Whenever we have discrete objects the integral turns into a sum. For instance, for vectors defined on ℓ_2, i.e. the set of square summable infinite dimensional vectors with index running over $\mathbb{Z}$ we obtain the following definition.

$$[f \circledast g](i) = \sum_a f(a) g(i - a)$$

For two-dimensional arrays we have a corresponding sum with indices (i, j) for f and $(i - a, j - b)$ for g respectively. This looks almost the same in the definition above, with one major difference. Rather than using $(i + a, j + b)$ we are using the difference instead. Note, though, that this distinction is mostly

cosmetic since we can always match the notation by using $\tilde{V}[a,b] = V[-a,-b]$ to obtain $h = x \circledast \tilde{V}$. Note that the original definition is actually a *cross correlation*. We will come back to this in the following section.

6.1.4 Waldo Revisited

Let's see what this looks like if we want to build an improved Waldo detector. The convolutional layer picks windows of a given size and weighs intensities according to the mask V. We expect that wherever the waldoness' is highest, we will also find a peak in the hidden layer activations.

There's just a problem with this approach: so far we blissfully ignored that images consist of 3 channels - red, green and blue. In reality images are thus not two-dimensional objects but three-dimensional tensors, e.g. of $1024 \times 1024 \times 3$ pixels. We thus index $\mathbf{x}$ as $x[i,j,k]$. The convolutional mask has to adapt accordingly. Instead of $V[a,b]$ we now have $V[a,b,c]$.

The last flaw in our reasoning is that this approach generates only one set of activations. This might not be great if we want to detect Waldo in several steps. We might need edge detectors, detectors for different colors, etc.; In short, we want to retain some information about edges, color gradients, combinations of colors, and a great many other things. An easy way to address this is to allow for *output channels*. We can take care of this by adding a fourth coordinate to V via $V[a,b,c,d]$. Putting all together we have:

$$h[i,j,k] = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_{c} V[a,b,c,k] \cdot x[i+a,j+b,c]$$

This is the definition of a convolutional neural network layer. There are still many operations that we need to address. For instance, we need to figure out how to combine all the activations to a single output (e.g. whether there's a Waldo in the image). We also need to decide how to compute things efficiently, how to combine multiple layers, and whether it is a good idea to have many narrow or a few wide layers. All of this will be addressed in the remainder of the chapter. For now we can bask in the glory having understood why convolutions exist in principle.

Summary

- Translation invariance in images implies that all patches of an image will be treated in the same manner.
- Locality means that only a small neighborhood of pixels will be used for computation.
- Channels on input and output allows for meaningful feature analysis.

Exercises

1. Assume that the size of the convolution mask is $\Delta = 0$. Show that in this case the convolutional mask implements an MLP independently for each set of channels.
2. Why might translation invariance not be a good idea after all? Does it make sense for pigs to fly?
3. What happens at the boundary of an image?
4. Derive an analogous convolutional layer for audio.
5. What goes wrong when you apply the above reasoning to text? Hint - what is the structure of language?
6. Prove that $f \circledast g = g \circledast f$.

Scan the QR Code to Discuss

6.2 Convolutions for Images

Now that we understand how to design convolutional networks in theory, let's see how this works in practice. For the next chapters we will stick to dealing with images, since they constitute one of the

most common use cases for convolutional networks. That is, we will discuss the two-dimensional case (image height and width). We begin with the cross-correlation operator that we introduced in the previous section. Strictly speaking, convolutional networks are a slight misnomer (but for notation only), since the operations are typically expressed as cross correlations.

6.2.1 The Cross-Correlation Operator

In a convolutional layer, an input array and a correlation kernel array output an array through a cross-correlation operation. Let's see how this works for two dimensions. As shown below, the input is a two-dimensional array with a height of 3 and width of 3. We mark the shape of the array as 3×3 or (3, 3). The height and width of the kernel array are both 2. This array is also called a kernel or filter in convolution computations. The shape of the kernel window (also known as the convolution window) depends on the height and width of the kernel, which is 2×2.

Fig. 6.1: Two-dimensional cross-correlation operation. The shaded portions are the first output element and the input and kernel array elements used in its computation: $0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$.

In the two-dimensional cross-correlation operation, the convolution window starts from the top-left of the input array, and slides in the input array from left to right and top to bottom. When the convolution window slides to a certain position, the input subarray in the window and kernel array are multiplied and summed by element to get the element at the corresponding location in the output array. The output array has a height of 2 and width of 2, and the four elements are derived from a two-dimensional cross-correlation operation:

$$0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19,$$
$$1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 = 25,$$
$$3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 = 37,$$
$$4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 = 43.$$

Note that the output size is *smaller* than the input. In particular, the output size is given by the input size $H \times W$ minus the size of the convolutional kernel $h \times w$ via $(H - h + 1) \times (W - w + 1)$. This is the case since we need enough space to shift' the convolutional kernel across the image (later we will see how to keep the size unchanged by padding the image with zeros around its boundary such that there's enough space to shift the kernel). Next, we implement the above process in the `corr2d` function. It accepts the input array `X` with the kernel array `K` and outputs the array `Y`.

```
In [1]: from mxnet import autograd, nd
        from mxnet.gluon import nn

        # This function has been saved in the d2l package for future use
        def corr2d(X, K):
            h, w = K.shape
            Y = nd.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
            for i in range(Y.shape[0]):
                for j in range(Y.shape[1]):
                    Y[i, j] = (X[i: i + h, j: j + w] * K).sum()
            return Y
```

We can construct the input array X and the kernel array K of the figure above to validate the output of the two-dimensional cross-correlation operation.

```
In [2]: X = nd.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
        K = nd.array([[0, 1], [2, 3]])
        corr2d(X, K)

Out[2]:
        [[19. 25.]
         [37. 43.]]
        <NDArray 2x2 @cpu(0)>
```

6.2.2 Convolutional Layers

The convolutional layer cross-correlates the input and kernels and adds a scalar bias to get the output. The model parameters of the convolutional layer include the kernel and scalar bias. When training the model, we usually randomly initialize the kernel and then continuously iterate the kernel and bias.

Next, we implement a custom two-dimensional convolutional layer based on the corr2d function. In the __init__ constructor function, we declare weight and bias as the two model parameters. The forward computation function forward directly calls the corr2d function and adds the bias. Just like the $h \times w$ cross-correlation we also refer to convolutional layers as $h \times w$ dimensional convolutions.

```
In [3]: class Conv2D(nn.Block):
            def __init__(self, kernel_size, **kwargs):
                super(Conv2D, self).__init__(**kwargs)
                self.weight = self.params.get('weight', shape=kernel_size)
                self.bias = self.params.get('bias', shape=(1,))

            def forward(self, x):
                return corr2d(x, self.weight.data()) + self.bias.data()
```

6.2.3 Object Edge Detection in Images

Let's look at a simple application of a convolutional layer: detecting the edge of an object in an image by finding the location of the pixel change. First, we construct an image' of 6×8 pixels. The middle four columns are black (0), and the rest are white (1).

```
In [4]: X = nd.ones((6, 8))
        X[:, 2:6] = 0
        X
Out[4]:
        [[1. 1. 0. 0. 0. 0. 1. 1.]
         [1. 1. 0. 0. 0. 0. 1. 1.]
         [1. 1. 0. 0. 0. 0. 1. 1.]
         [1. 1. 0. 0. 0. 0. 1. 1.]
         [1. 1. 0. 0. 0. 0. 1. 1.]
         [1. 1. 0. 0. 0. 0. 1. 1.]]
        <NDArray 6x8 @cpu(0)>
```

Next we construct a kernel `K` with a height and width of 1 and 2. When this performs the cross-correlation operation with the input, if the horizontally adjacent elements are the same, the output is 0. Otherwise, the output is non-zero.

```
In [5]: K = nd.array([[1, -1]])
```

Enter `X` and our designed kernel `K` to perform the cross-correlation operations. As you can see, we will detect 1 for the edge from white to black and -1 for the edge from black to white. The rest of the outputs are 0.

```
In [6]: Y = corr2d(X, K)
        Y
Out[6]:
        [[ 0.  1.  0.  0.  0. -1.  0.]
         [ 0.  1.  0.  0.  0. -1.  0.]
         [ 0.  1.  0.  0.  0. -1.  0.]
         [ 0.  1.  0.  0.  0. -1.  0.]
         [ 0.  1.  0.  0.  0. -1.  0.]
         [ 0.  1.  0.  0.  0. -1.  0.]]
        <NDArray 6x7 @cpu(0)>
```

Let's apply the kernel to the transposed image'. As expected, it vanishes. The kernel `K` only detects vertical edges.

```
In [7]: corr2d(X.T, K)
Out[7]:
        [[0. 0. 0. 0. 0.]
         [0. 0. 0. 0. 0.]
         [0. 0. 0. 0. 0.]
         [0. 0. 0. 0. 0.]
         [0. 0. 0. 0. 0.]
         [0. 0. 0. 0. 0.]
         [0. 0. 0. 0. 0.]
         [0. 0. 0. 0. 0.]]
        <NDArray 8x5 @cpu(0)>
```

6.2.4 Learning a Kernel

Designing an edge detector by finite differences `[1, -1]` is neat if we know what we are looking for. However, as we look at larger kernels, or possibly multiple layers, it is next to impossible to specify such

filters manually. Let's see whether we can learn the kernel that generated `Y` from `X` by looking at the (input, output) pairs only. We first construct a convolutional layer and initialize its kernel into a random array. Next, in each iteration, we use the squared error to compare `Y` and the output of the convolutional layer, then calculate the gradient to update the weight. For the sake of simplicity, the convolutional layer here ignores the bias.

We previously constructed the `Conv2D` class. However, since we used single-element assignments, Gluon has some trouble finding the gradient. Instead, we use the built-in `Conv2D` class provided by Gluon below.

```
In [8]: # Construct a convolutional layer with 1 output channel (channels will be
        # introduced in the following section) and a kernel array shape of (1, 2)
        conv2d = nn.Conv2D(1, kernel_size=(1, 2))
        conv2d.initialize()

        # The two-dimensional convolutional layer uses four-dimensional input and
        # output in the format of (example channel, height, width), where the batch
        # size (number of examples in the batch) and the number of channels are both 1
        X = X.reshape((1, 1, 6, 8))
        Y = Y.reshape((1, 1, 6, 7))

        for i in range(10):
            with autograd.record():
                Y_hat = conv2d(X)
                l = (Y_hat - Y) ** 2
            l.backward()
            # For the sake of simplicity, we ignore the bias here
            conv2d.weight.data()[:] -= 3e-2 * conv2d.weight.grad()
            if (i + 1) % 2 == 0:
                print('batch %d, loss %.3f' % (i + 1, l.sum().asscalar()))
batch 2, loss 4.949
batch 4, loss 0.831
batch 6, loss 0.140
batch 8, loss 0.024
batch 10, loss 0.004
```

As you can see, the error has dropped to a relatively small value after 10 iterations. Now we will take a look at the kernel array we learned.

```
In [9]: conv2d.weight.data().reshape((1, 2))
Out[9]:
        [[ 0.9895    -0.9873705]]
        <NDArray 1x2 @cpu(0)>
```

We find that the kernel array we learned is very close to the kernel array `K` we defined earlier.

6.2.5 Cross-correlation and Convolution

Recall the observation from the previous section that cross-correlation and convolution are equivalent. In the figure above it is easy to see this correspondence. Simply flip the kernel from the bottom left to the top right. In this case the indexing in the sum is reverted, yet the same result can be obtained. In keeping

with standard terminology with deep learning literature we will continue to refer to the cross-correlation operation as a convolution even though it is strictly speaking something slightly different.

Summary

- The core computation of a two-dimensional convolutional layer is a two-dimensional cross-correlation operation. In its simplest form, this performs a cross-correlation operation on the two-dimensional input data and the kernel, and then adds a bias.
- We can design a kernel to detect edges in images.
- We can learn the kernel through data.

Exercises

1. Construct an image X with diagonal edges.
    - What happens if you apply the kernel K to it?
    - What happens if you transpose X?
    - What happens if you transpose K?
2. When you try to automatically find the gradient for the Conv2D class we created, what kind of error message do you see?
3. How do you represent a cross-correlation operation as a matrix multiplication by changing the input and kernel arrays?
4. Design some kernels manually.
    - What is the form of a kernel for the second derivative?
    - What is the kernel for the Laplace operator?
    - What is the kernel for an integral?
    - What is the minimum size of a kernel to obtain a derivative of degree d?

Scan the QR Code to Discuss

6.3 Padding and Stride

In the example in the previous section, we used an input with a height and width of 3 and a convolution kernel with a height and width of 2 to get an output with a height and a width of 2. In general, assuming the input shape is $n_h \times n_w$ and the convolution kernel window shape is $k_h \times k_w$, then the output shape will be

$$(n_h - k_h + 1) \times (n_w - k_w + 1).$$

Therefore, the output shape of the convolutional layer is determined by the shape of the input and the shape of the convolution kernel window. In several cases we might want to change the dimensionality of the output:

- Multiple layers of convolutions reduce the information available at the boundary, often by much more than what we would want. If we start with a 240x240 pixel image, 10 layers of 5x5 convolutions reduce the image to 200x200 pixels, effectively slicing off 30% of the image and with it obliterating anything interesting on the boundaries. Padding mitigates this problem.

- In some cases we want to reduce the resolution drastically, e.g. halving it if we think that such a high input dimensionality is not required. In this case we might want to subsample the output. Strides address this.

- In some cases we want to increase the resolution, e.g. for image superresolution or for audio generation. Again, strides come to our rescue.

- In some cases we want to increase the length gently to a given size (mostly for sentences of variable length or for filling in patches). Padding addresses this.

6.3.1 Padding

As we saw so far, convolutions are quite useful. Alas, on the boundaries we encounter the problem that we keep on losing pixels. For any given convolution it's only a few pixels but this adds up as we discussed above. If the image was larger things would be easier - we could simply record one that's larger. Unfortunately, that's not what we get in reality. One solution to this problem is to add extra pixels around the boundary of the image, thus increasing the effective size of the image (the extra pixels typically assume the value 0). In the figure below we pad the 3×5 to increase to 5×7. The corresponding output then increases to a 4×6 matrix.
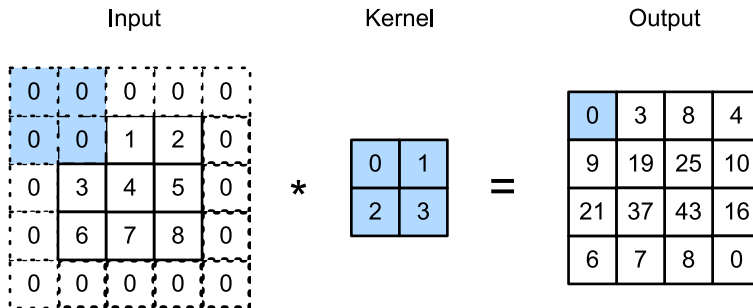
Fig. 6.2: Two-dimensional cross-correlation with padding. The shaded portions are the input and kernel array elements used by the first output element: $0 \times 0 + 0 \times 1 + 0 \times 2 + 0 \times 3 = 0$.

In general, if a total of p_h rows are padded on both sides of the height and a total of p_w columns are padded on both sides of width, the output shape will be

$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1),$$

This means that the height and width of the output will increase by p_h and p_w respectively.

In many cases, we will want to set $p_h = k_h - 1$ and $p_w = k_w - 1$ to give the input and output the same height and width. This will make it easier to predict the output shape of each layer when constructing the network. Assuming that k_h is odd here, we will pad $p_h/2$ rows on both sides of the height. If k_h is even, one possibility is to pad $\lceil p_h/2 \rceil$ rows on the top of the input and $\lfloor p_h/2 \rfloor$ rows on the bottom. We will pad both sides of the width in the same way.

Convolutional neural networks often use convolution kernels with odd height and width values, such as 1, 3, 5, and 7, so the number of padding rows or columns on both sides are the same. For any two-dimensional array X, assume that the element in its ith row and jth column is X[i,j]. When the number of padding rows or columns on both sides are the same so that the input and output have the same height and width, we know that the output Y[i,j] is calculated by cross-correlation of the input and convolution kernel with the window centered on X[i,j].

In the following example we create a two-dimensional convolutional layer with a height and width of 3, and then assume that the padding number on both sides of the input height and width is 1. Given an input with a height and width of 8, we find that the height and width of the output is also 8.

```
In [1]: from mxnet import nd
        from mxnet.gluon import nn

        # We define a convenience function to calculate the convolutional layer. This
        # function initializes the convolutional layer weights and performs
        # corresponding dimensionality elevations and reductions on the input and
        # output
        def comp_conv2d(conv2d, X):
            conv2d.initialize()
            # (1,1) indicates that the batch size and the number of channels
```

```
          # (described in later chapters) are both 1
          X = X.reshape((1, 1) + X.shape)
          Y = conv2d(X)
          # Exclude the first two dimensions that do not interest us: batch and
          # channel
          return Y.reshape(Y.shape[2:])

      # Note that here 1 row or column is padded on either side, so a total of 2
      # rows or columns are added
      conv2d = nn.Conv2D(1, kernel_size=3, padding=1)
      X = nd.random.uniform(shape=(8, 8))
      comp_conv2d(conv2d, X).shape

Out[1]: (8, 8)
```

When the height and width of the convolution kernel are different, we can make the output and input have the same height and width by setting different padding numbers for height and width.

```
In [2]: # Here, we use a convolution kernel with a height of 5 and a width of 3. The
        # padding numbers on both sides of the height and width are 2 and 1,
        # respectively
        conv2d = nn.Conv2D(1, kernel_size=(5, 3), padding=(2, 1))
        comp_conv2d(conv2d, X).shape

Out[2]: (8, 8)
```

6.3.2 Stride

When computing the cross-correlation the convolution window starts from the top-left of the input array, and slides in the input array from left to right and top to bottom. We refer to the number of rows and columns per slide as the stride.

In the current example, the stride is 1, both in terms of height and width. We can also use a larger stride. The figure below shows a two-dimensional cross-correlation operation with a stride of 3 vertically and 2 horizontally. We can see that when the second element of the first column is output, the convolution window slides down three rows. The convolution window slides two columns to the right when the second element of the first row is output. When the convolution window slides two columns to the right on the input, there is no output because the input element cannot fill the window (unless we add padding).
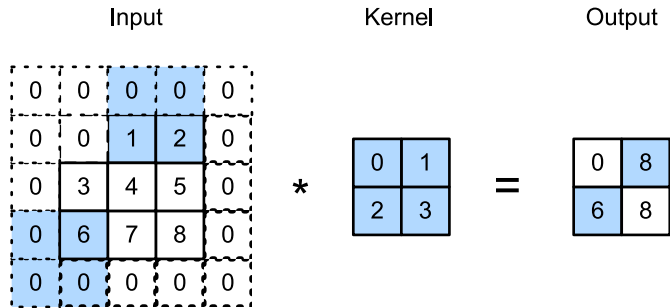
Fig. 6.3: Cross-correlation with strides of 3 and 2 for height and width respectively. The shaded portions are the output element and the input and core array elements used in its computation: $0 \times 0 + 0 \times 1 + 1 \times 2 + 2 \times 3 = 8, 0 \times 0 + 6 \times 1 + 0 \times 2 + 0 \times 3 = 6$.

In general, when the stride for the height is s_h and the stride for the width is s_w, the output shape is

$$\lfloor (n_h - k_h + p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w)/s_w \rfloor.$$

If we set $p_h = k_h - 1$ and $p_w = k_w - 1$, then the output shape will be simplified to $\lfloor (n_h + s_h - 1)/s_h \rfloor \times \lfloor (n_w + s_w - 1)/s_w \rfloor$. Going a step further, if the input height and width are divisible by the strides on the height and width, then the output shape will be $(n_h/s_h) \times (n_w/s_w)$.

Below, we set the strides on both the height and width to 2, thus halving the input height and width.

```
In [3]: conv2d = nn.Conv2D(1, kernel_size=3, padding=1, strides=2)
        comp_conv2d(conv2d, X).shape
```

```
Out[3]: (4, 4)
```

Next, we will look at a slightly more complicated example.

```
In [4]: conv2d = nn.Conv2D(1, kernel_size=(3, 5), padding=(0, 1), strides=(3, 4))
        comp_conv2d(conv2d, X).shape
```

```
Out[4]: (2, 2)
```

For the sake of brevity, when the padding number on both sides of the input height and width are p_h and p_w respectively, we call the padding (p_h, p_w). Specifically, when $p_h = p_w = p$, the padding is p. When the strides on the height and width are s_h and s_w, respectively, we call the stride (s_h, s_w). Specifically, when $s_h = s_w = s$, the stride is s. By default, the padding is 0 and the stride is 1. In practice we rarely use inhomogeneous strides or padding, i.e. we usually have $p_h = p_w$ and $s_h = s_w$.

Summary

- Padding can increase the height and width of the output. This is often used to give the output the same height and width as the input.

6.3. Padding and Stride

- The stride can reduce the resolution of the output, for example reducing the height and width of the output to only $1/n$ of the height and width of the input (n is an integer greater than 1).

- Padding and stride can be used to adjust the dimensionality of the data effectively.

Exercises

1. For the last example in this section, use the shape calculation formula to calculate the output shape to see if it is consistent with the experimental results.

2. Try other padding and stride combinations on the experiments in this section.

3. For audio signals, what does a stride of 2 correspond to?

4. What are the computational benefits of a stride larger than 1.

Scan the QR Code to Discuss

6.4 Multiple Input and Output Channels

In the previous section, we used two-dimensional arrays are the inputs and outputs, but real data often has higher dimensions. For example, a color image has three color channels of RGB (red, green, and blue) in addition to the height and width dimensions. Assuming that the height and width of the color image are h and w (pixels), it can be represented in the memory as a multi-dimensional array of $3 \times h \times w$. We refer to this dimension, with a size of 3, as the channel dimension. In this section, we will introduce convolution kernels with multiple input and multiple output channels.

6.4.1 Multiple Input Channels

When the input data contains multiple channels, we need to construct a convolution kernel with the same number of input channels as the input data, so that it can perform cross-correlation with the input data. Assuming that the number of channels for the input data is c_i, the number of input channels of the convolution kernel also needs to be c_i. We set the convolution kernel window shape to $k_h \times k_w$. This way, when $c_i = 1$, we know that the convolution kernel contains only a two-dimensional array of the shape $k_h \times k_w$. When $c_i > 1$, we will assign a kernel array of shape $k_h \times k_w$ to each input channel. Concatenating these c_i arrays to the input channel dimension yields a convolution kernel of

shape $c_i \times k_h \times k_w$. Since the input and convolution kernel each have c_i channels, we can perform a cross-correlation operation on the two-dimensional array of the input and the two-dimensional kernel array of the convolution kernel on each channel, and then add the c_i cross-correlated two-dimensional outputs by channel to get a two-dimensional array. This is the output of a two-dimensional cross-correlation operation between the multi-channel input data and the multi-input channel convolution kernel.

The figure below shows an example of a two-dimensional cross-correlation computation with two input channels. On each channel, the two-dimensional input array and the two-dimensional kernel array are cross-correlated, and then added together by channel to obtain the output. The shaded portions are the first output element as well as the input and kernel array elements used in its computation: $(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56$.
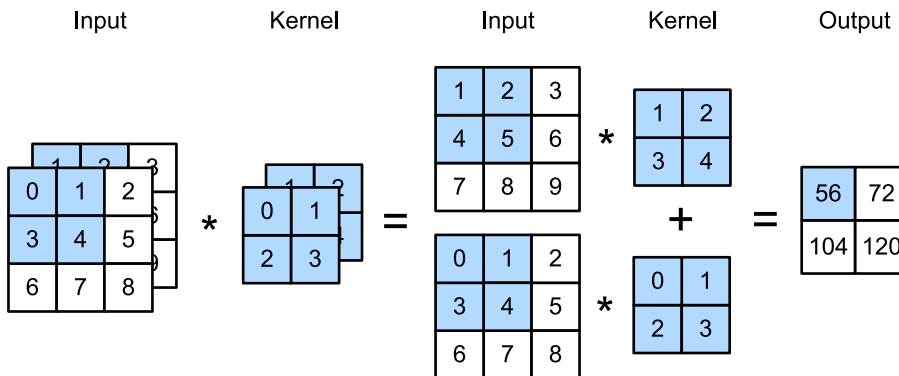
Fig. 6.4: Cross-correlation computation with 2 input channels. The shaded portions are the first output element as well as the input and kernel array elements used in its computation: $(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56$.

Let's implement cross-correlation operations with multiple input channels. We simply need to perform a cross-correlation operation for each channel, and then add them up using the add_n function.

```
In [1]: import sys
        sys.path.insert(0, '..')

        import d2l
        from mxnet import nd

        def corr2d_multi_in(X, K):
            # First, traverse along the 0th dimension (channel dimension) of X and K.
            # Then, add them together by using * to turn the result list into a
            # positional argument of the add_n function
            return nd.add_n(*[d2l.corr2d(x, k) for x, k in zip(X, K)])
```

We can construct the input array X and the kernel array K of the above diagram to validate the output of the cross-correlation operation.

```
In [2]: X = nd.array([[[0, 1, 2], [3, 4, 5], [6, 7, 8]],
                      [[1, 2, 3], [4, 5, 6], [7, 8, 9]]])
```

```
         K = nd.array([[[0, 1], [2, 3]], [[1, 2], [3, 4]]])

         corr2d_multi_in(X, K)

Out[2]:
         [[ 56.  72.]
          [104. 120.]]
         <NDArray 2x2 @cpu(0)>
```

6.4.2 Multiple Output Channels

Regardless of the number of input channels, so far we always ended up with one output channel. However, it is quite reasonable to assume that we might need more than one output, e.g. for edge detection in different directions or for more advanced filters. Denote by c_i and c_o the number of input and output channels respectively and let k_h and k_w be the height and width of the kernel. To get an output with multiple channels, we can create a kernel array of shape $c_i \times k_h \times k_w$ for each output channel. We concatenate them on the output channel dimension, so that the shape of the convolution kernel is $c_o \times c_i \times k_h \times k_w$. In cross-correlation operations, the result on each output channel is calculated from the kernel array of the convolution kernel on the same output channel and the entire input array.

We implement a cross-correlation function to calculate the output of multiple channels as shown below.

```
In [3]: def corr2d_multi_in_out(X, K):
            # Traverse along the 0th dimension of K, and each time, perform
            # cross-correlation operations with input X. All of the results are merged
            # together using the stack function
            return nd.stack(*[corr2d_multi_in(X, k) for k in K])
```

We construct a convolution kernel with 3 output channels by concatenating the kernel array K with K+1 (plus one for each element in K) and K+2.

```
In [4]: K = nd.stack(K, K + 1, K + 2)
        K.shape

Out[4]: (3, 2, 2, 2)
```

Below, we perform cross-correlation operations on the input array X with the kernel array K. Now the output contains 3 channels. The result of the first channel is consistent with the result of the previous input array X and the multi-input channel, single-output channel kernel.

```
In [5]: corr2d_multi_in_out(X, K)

Out[5]:
         [[[ 56.  72.]
           [104. 120.]]

          [[ 76. 100.]
           [148. 172.]]

          [[ 96. 128.]
           [192. 224.]]]
         <NDArray 3x2x2 @cpu(0)>
```

6.4.3 1×1 **Convolutional Layer**

At first a 1×1 convolution, i.e. $k_h = k_w = 1$, doesn't seem to make much sense. After all, a convolution correlates adjacent pixels. A 1×1 convolution obviously doesn't. Nonetheless, it is a popular choice when designing complex and deep networks. Let's see in some detail what it actually does.

Because the minimum window is used, the 1×1 convolution loses the ability of the convolutional layer to recognize patterns composed of adjacent elements in the height and width dimensions. The main computation of the 1×1 convolution occurs on the channel dimension. The figure below shows the cross-correlation computation using the 1×1 convolution kernel with 3 input channels and 2 output channels. It is worth noting that the inputs and outputs have the same height and width. Each element in the output is derived from a linear combination of elements in the same position in the height and width of the input between different channels. Assuming that the channel dimension is considered a feature dimension and that the elements in the height and width dimensions are considered data examples, then the 1×1 convolutional layer is equivalent to the fully connected layer.
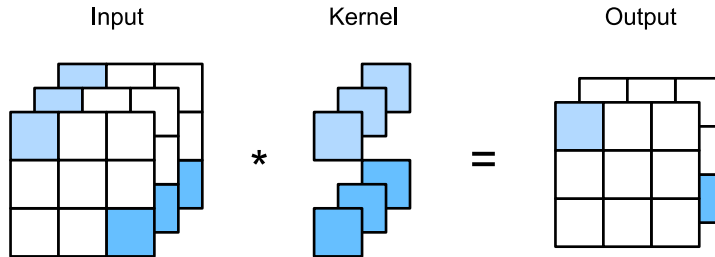
Fig. 6.5: The cross-correlation computation uses the 1×1 convolution kernel with 3 input channels and 2 output channels. The inputs and outputs have the same height and width.

Let's check whether this works in practice: we implement the 1×1 convolution using a fully connected layer. The only thing is that we need to make some adjustments to the data shape before and after the matrix multiplication.

```
In [6]: def corr2d_multi_in_out_1x1(X, K):
            c_i, h, w = X.shape
            c_o = K.shape[0]
            X = X.reshape((c_i, h * w))
            K = K.reshape((c_o, c_i))
            Y = nd.dot(K, X)  # Matrix multiplication in the fully connected layer
            return Y.reshape((c_o, h, w))
```

When performing 1×1 convolution, the above function is equivalent to the previously implemented cross-correlation function `corr2d_multi_in_out`. Let's check this with some reference data.

```
In [7]: X = nd.random.uniform(shape=(3, 3, 3))
        K = nd.random.uniform(shape=(2, 3, 1, 1))

        Y1 = corr2d_multi_in_out_1x1(X, K)
        Y2 = corr2d_multi_in_out(X, K)
```

```
        (Y1 - Y2).norm().asscalar() < 1e-6
```
Out[7]: True

Summary

- Multiple channels can be used to extend the model parameters of the convolutional layer.
- The 1×1 convolutional layer is equivalent to the fully connected layer, when applied on a per pixel basis.
- The 1×1 convolutional layer is typically used to adjust the number of channels between network layers and to control model complexity.

Exercises

1. Assume that we have two convolutional kernels of size k_1 and k_2 respectively (with no nonlinearity in between).
   - Prove that the result of the operation can be expressed by a single convolution.
   - What is the dimensionality of the equivalent single convolution?
   - Is the converse true?

2. Assume an input shape of $c_i \times h \times w$ and a convolution kernel with the shape $c_o \times c_i \times k_h \times k_w$, padding of (p_h, p_w), and stride of (s_h, s_w).
   - What is the computational cost (multiplications and additions) for the forward computation?
   - What is the memory footprint?
   - What is the memory footprint for the backward computation?
   - What is the computational cost for the backward computation?

3. By what factor does the number of calculations increase if we double the number of input channels c_i and the number of output channels c_o? What happens if we double the padding?

4. If the height and width of the convolution kernel is $k_h = k_w = 1$, what is the complexity of the forward computation?

5. Are the variables `Y1` and `Y2` in the last example of this section exactly the same? Why?

6. How would you implement convolutions using matrix multiplication when the convolution window is not 1×1?

6.5 Pooling

As we process images (or other data sources) we will eventually want to reduce the resolution of the images. After all, we typically want to output an estimate that does not depend on the dimensionality of the original image. Secondly, when detecting lower-level features, such as edge detection (we covered this in the section on *convolutional layers*), we often want to have some degree of invariance to translation. For instance, if we take the image X with a sharp delineation between black and white and if we shift it by one pixel to the right, i.e. `Z[i,j] = X[i,j+1]`, then the output for for the new image Z will be vastly different. The edge will have shifted by one pixel and with it all the activations. In reality objects hardly ever occur exactly at the same place. In fact, even with a tripod and a stationary object, vibration of the camera due to the movement of the shutter might shift things by a pixel or so (this is why high end cameras have a special option to fix this). Given that, we need a mathematical device to address the problem.

This section introduces pooling layers, which were proposed to alleviate the excessive sensitivity of the convolutional layer to location and to reduce the resolution of images through the processing pipeline.

6.5.1 Maximum Pooling and Average Pooling

Like convolutions, pooling computes the output for each element in a fixed-shape window (also known as a pooling window) of input data. Different from the cross-correlation computation of the inputs and kernels in the convolutional layer, the pooling layer directly calculates the maximum or average value of the elements in the pooling window. These operations are called maximum pooling or average pooling respectively. In maximum pooling, the pooling window starts from the top left of the input array, and slides in the input array from left to right and top to bottom. When the pooling window slides to a certain position, the maximum value of the input subarray in the window is the element at the corresponding location in the output array.
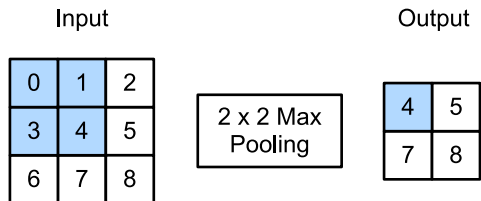
Fig. 6.6: Maximum pooling with a pooling window shape of 2×2. The shaded portions represent the first output element and the input element used for its computation: $\max(0, 1, 3, 4) = 4$

The output array in the figure above has a height of 2 and a width of 2. The four elements are derived from the maximum value of max:

$$\max(0, 1, 3, 4) = 4,$$
$$\max(1, 2, 4, 5) = 5,$$
$$\max(3, 4, 6, 7) = 7,$$
$$\max(4, 5, 7, 8) = 8.$$

Average pooling works like maximum pooling, only with the maximum operator replaced by the average operator. The pooling layer with a pooling window shape of $p \times q$ is called the $p \times q$ pooling layer. The pooling operation is called $p \times q$ pooling.

Let us return to the object edge detection example mentioned at the beginning of this section. Now we will use the output of the convolutional layer as the input for 2×2 maximum pooling. Set the convolutional layer input as X and the pooling layer output as Y. Whether or not the values of X[i, j] and X[i, j+1] are different, or X[i, j+1] and X[i, j+2] are different, the pooling layer outputs all include Y[i, j]=1. That is to say, using the 2×2 maximum pooling layer, we can still detect if the pattern recognized by the convolutional layer moves no more than one element in height and width.

As shown below, we implement the forward computation of the pooling layer in the pool2d function. This function is very similar to the corr2d function in the section on *convolutions*. The only difference lies in the computation of the output Y.

```
In [1]: from mxnet import nd
        from mxnet.gluon import nn

        def pool2d(X, pool_size, mode='max'):
            p_h, p_w = pool_size
            Y = nd.zeros((X.shape[0] - p_h + 1, X.shape[1] - p_w + 1))
            for i in range(Y.shape[0]):
                for j in range(Y.shape[1]):
                    if mode == 'max':
                        Y[i, j] = X[i: i + p_h, j: j + p_w].max()
                    elif mode == 'avg':
                        Y[i, j] = X[i: i + p_h, j: j + p_w].mean()
            return Y
```

We can construct the input array X in the above diagram to validate the output of the two-dimensional

maximum pooling layer.

```
In [2]: X = nd.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
        pool2d(X, (2, 2))
Out[2]:
        [[4. 5.]
         [7. 8.]]
        <NDArray 2x2 @cpu(0)>
```

At the same time, we experiment with the average pooling layer.

```
In [3]: pool2d(X, (2, 2), 'avg')
Out[3]:
        [[2. 3.]
         [5. 6.]]
        <NDArray 2x2 @cpu(0)>
```

6.5.2 Padding and Stride

Like the convolutional layer, the pooling layer can also change the output shape by padding the two sides of the input height and width and adjusting the window stride. The pooling layer works in the same way as the convolutional layer in terms of padding and strides. We will demonstrate the use of padding and stride in the pooling layer through the two-dimensional maximum pooling layer MaxPool2D in the nn module. We first construct an input data of shape (1, 1, 4, 4), where the first two dimensions are batch and channel.

```
In [4]: X = nd.arange(16).reshape((1, 1, 4, 4))
        X
Out[4]:
        [[[[ 0.  1.  2.  3.]
           [ 4.  5.  6.  7.]
           [ 8.  9. 10. 11.]
           [12. 13. 14. 15.]]]]
        <NDArray 1x1x4x4 @cpu(0)>
```

By default, the stride in the MaxPool2D class has the same shape as the pooling window. Below, we use a pooling window of shape (3, 3), so we get a stride shape of (3, 3) by default.

```
In [5]: pool2d = nn.MaxPool2D(3)
        # Because there are no model parameters in the pooling layer, we do not need
        # to call the parameter initialization function
        pool2d(X)
Out[5]:
        [[[[10.]]]]
        <NDArray 1x1x1x1 @cpu(0)>
```

The stride and padding can be manually specified.

```
In [6]: pool2d = nn.MaxPool2D(3, padding=1, strides=2)
        pool2d(X)
```

```
Out[6]:
     [[[[ 5.   7.]
       [13. 15.]]]]
     <NDArray 1x1x2x2 @cpu(0)>
```

Of course, we can specify an arbitrary rectangular pooling window and specify the padding and stride for height and width, respectively.

```
In [7]: pool2d = nn.MaxPool2D((2, 3), padding=(1, 2), strides=(2, 3))
        pool2d(X)
Out[7]:
     [[[[ 0.   3.]
       [ 8. 11.]
       [12. 15.]]]]
     <NDArray 1x1x3x2 @cpu(0)>
```

6.5.3 Multiple Channels

When processing multi-channel input data, the pooling layer pools each input channel separately, rather than adding the inputs of each channel by channel as in a convolutional layer. This means that the number of output channels for the pooling layer is the same as the number of input channels. Below, we will concatenate arrays X and X+1 on the channel dimension to construct an input with 2 channels.

```
In [8]: X = nd.concat(X, X + 1, dim=1)
        X
Out[8]:
     [[[[ 0.  1.  2.  3.]
       [ 4.  5.  6.  7.]
       [ 8.  9. 10. 11.]
       [12. 13. 14. 15.]]

      [[ 1.  2.  3.  4.]
       [ 5.  6.  7.  8.]
       [ 9. 10. 11. 12.]
       [13. 14. 15. 16.]]]]
     <NDArray 1x2x4x4 @cpu(0)>
```

As we can see, the number of output channels is still 2 after pooling.

```
In [9]: pool2d = nn.MaxPool2D(3, padding=1, strides=2)
        pool2d(X)
Out[9]:
     [[[[ 5.   7.]
       [13. 15.]]

      [[ 6.   8.]
       [14. 16.]]]]
     <NDArray 1x2x2x2 @cpu(0)>
```

Summary

- Taking the input elements in the pooling window, the maximum pooling operation assigns the maximum value as the output and the average pooling operation assigns the average value as the output.

- One of the major functions of a pooling layer is to alleviate the excessive sensitivity of the convolutional layer to location.

- We can specify the padding and stride for the pooling layer.

- Maximum pooling, combined with a stride larger than 1 can be used to reduce the resolution.

- The pooling layer's number of output channels is the same as the number of input channels.

Exercises

1. Implement average pooling as a convolution.

2. What is the computational cost of the pooling layer? Assume that the input to the pooling layer is of size $c \times h \times w$, the pooling window has a shape of $p_h \times p_w$ with a padding of (p_h, p_w) and a stride of (s_h, s_w).

3. Why do you expect maximum pooling and average pooling to work differently?

4. Do we need a separate minimum pooling layer? Can you replace it with another operation?

5. Is there another operation between average and maximum pooling that you could consider (hint - recall the softmax)? Why might it not be so popular?

Scan the QR Code to Discuss

6.6 Convolutional Neural Networks (LeNet)

In our first encounter with image data we applied a Multilayer Perceptron to pictures of clothing in the Fashion-MNIST data set. Both the height and width of each image were 28 pixels. We expanded the pixels in the image line by line to get a vector of length 784, and then used them as inputs to the fully connected layer. However, this classification method has certain limitations:

1. The adjacent pixels in the same column of an image may be far apart in this vector. The patterns they create may be difficult for the model to recognize. In fact, the vectorial representation ignores position entirely - we could have permuted all 28×28 pixels at random and obtained the same results.

2. For large input images, using a fully connected layer can easily cause the model to become too large, as we discussed previously.

As discussed in the previous sections, the convolutional layer attempts to solve both problems. On the one hand, the convolutional layer retains the input shape, so that the correlation of image pixels in the directions of both height and width can be recognized effectively. On the other hand, the convolutional layer repeatedly calculates the same kernel and the input of different positions through the sliding window, thereby avoiding excessively large parameter sizes.

A convolutional neural network is a network with convolutional layers. In this section, we will introduce an early convolutional neural network used to recognize handwritten digits in images - LeNet5. Convolutional networks were invented by Yann LeCun and coworkers at AT&T Bell Labs in the early 90s. LeNet showed that it was possible to use gradient descent to train the convolutional neural network for handwritten digit recognition. It achieved outstanding results at the time (only matched by Support Vector Machines at the time).

6.6.1 LeNet

LeNet is divided into two parts: a block of convolutional layers and one of fully connected ones. Below, we will introduce these two modules separately. Before going into details, let's briefly review the model in pictures. To illustrate the issue of channels and the specific layers we will use a rather description (later we will see how to convey the same information more concisely).
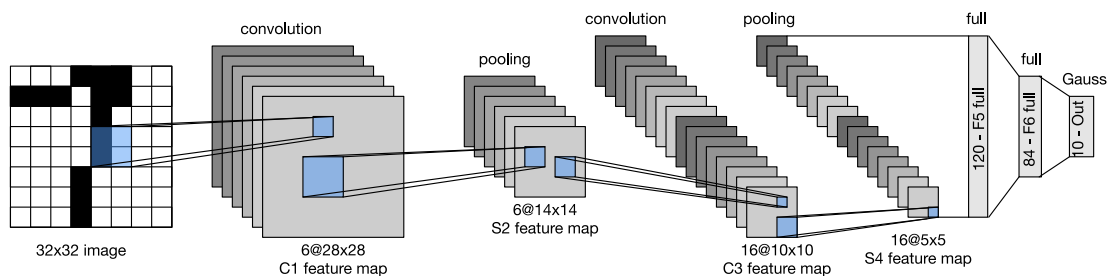
Fig. 6.7: Data flow in LeNet 5. The input is a handwritten digit, the output a probabilitiy over 10 possible outcomes.

The basic units in the convolutional block are a convolutional layer and a subsequent average pooling layer (note that max-pooling works better, but it had not been invented in the 90s yet). The convolutional layer is used to recognize the spatial patterns in the image, such as lines and the parts of objects, and the subsequent average pooling layer is used to reduce the dimensionality. The convolutional layer block is

composed of repeated stacks of these two basic units. In the convolutional layer block, each convolutional layer uses a 5×5 window and a sigmoid activation function for the output (note that ReLu works better, but it had not been invented in the 90s yet). The number of output channels for the first convolutional layer is 6, and the number of output channels for the second convolutional layer is increased to 16. This is because the height and width of the input of the second convolutional layer is smaller than that of the first convolutional layer. Therefore, increasing the number of output channels makes the parameter sizes of the two convolutional layers similar. The window shape for the two average pooling layers of the convolutional layer block is 2×2 and the stride is 2. Because the pooling window has the same shape as the stride, the areas covered by the pooling window sliding on each input do not overlap. In other words, the pooling layer performs downsampling.

The output shape of the convolutional layer block is (batch size, channel, height, width). When the output of the convolutional layer block is passed into the fully connected layer block, the fully connected layer block flattens each example in the mini-batch. That is to say, the input shape of the fully connected layer will become two dimensional: the first dimension is the example in the mini-batch, the second dimension is the vector representation after each example is flattened, and the vector length is the product of channel, height, and width. The fully connected layer block has three fully connected layers. They have 120, 84, and 10 outputs, respectively. Here, 10 is the number of output classes.

Next, we implement the LeNet model through the Sequential class.

```
In [1]: import sys
        sys.path.insert(0, '..')

        import d2l
        import mxnet as mx
        from mxnet import autograd, gluon, init, nd
        from mxnet.gluon import loss as gloss, nn
        import time

        net = nn.Sequential()
        net.add(nn.Conv2D(channels=6, kernel_size=5, padding=2, activation='sigmoid'),
                nn.AvgPool2D(pool_size=2, strides=2),
                nn.Conv2D(channels=16, kernel_size=5, activation='sigmoid'),
                nn.AvgPool2D(pool_size=2, strides=2),
                # Dense will transform the input of the shape (batch size, channel,
                # height, width) into the input of the shape (batch size,
                # channel * height * width) automatically by default
                nn.Dense(120, activation='sigmoid'),
                nn.Dense(84, activation='sigmoid'),
                nn.Dense(10))
```

We took the liberty of replacing the Gaussian activation in the last layer by a regular dense network since this is rather much more convenient to train. Other than that the network matches the historical definition of LeNet5. Next, we feed a single-channel example of size 28×28 into the network and perform a forward computation layer by layer to see the output shape of each layer.

```
In [2]: X = nd.random.uniform(shape=(1, 1, 28, 28))
        net.initialize()
        for layer in net:
            X = layer(X)
            print(layer.name, 'output shape:\t', X.shape)
```

```
conv0 output shape:      (1, 6, 28, 28)
pool0 output shape:      (1, 6, 14, 14)
conv1 output shape:      (1, 16, 10, 10)
pool1 output shape:      (1, 16, 5, 5)
dense0 output shape:     (1, 120)
dense1 output shape:     (1, 84)
dense2 output shape:     (1, 10)
```

We can see that the height and width of the input in the convolutional layer block is reduced, layer by layer. The convolutional layer uses a kernel with a height and width of 5 to reduce the height and width by 4, while the pooling layer halves the height and width, but the number of channels increases from 1 to 16. The fully connected layer reduces the number of outputs layer by layer, until the number of image classes becomes 10.
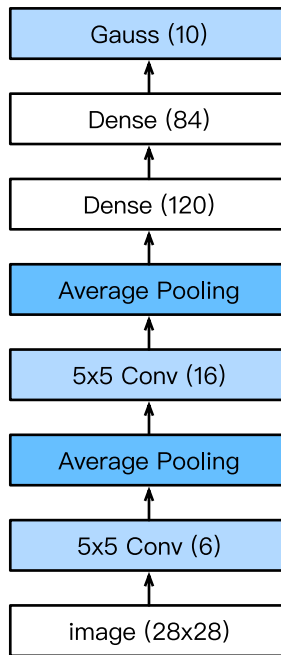
Fig. 6.8: Compressed notation for LeNet5

6.6.2 Data Acquisition and Training

Now, we will experiment with the LeNet model. We still use Fashion-MNIST as the training data set since the problem is rather more difficult than OCR (even in the 1990s the error rates were in the 1% range).

```
In [3]: batch_size = 256
        train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size=batch_size)
```

Since convolutional networks are significantly more expensive to compute than multilayer perceptrons we recommend using GPUs to speed up training. Time to introduce a convenience function that allows us to detect whether we have a GPU: it works by trying to allocate an NDArray on `gpu(0)`, and use `gpu(0)` if this is successful. Otherwise, we catch the resulting exception and we stick with the CPU.

```
In [4]:  # This function has been saved in the d2l package for future use
         def try_gpu():
             try:
                 ctx = mx.gpu()
                 _ = nd.zeros((1,), ctx=ctx)
             except mx.base.MXNetError:
                 ctx = mx.cpu()
             return ctx

         ctx = try_gpu()
         ctx
```

```
Out[4]: gpu(0)
```

Accordingly, we slightly modify the `evaluate_accuracy` function described when implementing the SoftMax from scratch. Since the data arrives in the CPU when loading we need to copy it to the GPU before any computation can occur. This is accomplished via the `as_in_context` function described in the *GPU Computing* section. Note that we accumulate the errors on the same device as where the data eventually lives (in `acc`). This avoids intermediate copy operations that would destroy performance.

```
In [5]:  # This function has been saved in the d2l package for future use. The function
         # will be gradually improved. Its complete implementation will be discussed in
         # the "Image Augmentation" section
         def evaluate_accuracy(data_iter, net, ctx):
             acc_sum, n = nd.array([0], ctx=ctx), 0
             for X, y in data_iter:
                 # If ctx is the GPU, copy the data to the GPU.
                 X, y = X.as_in_context(ctx), y.as_in_context(ctx).astype('float32')
                 acc_sum += (net(X).argmax(axis=1) == y).sum()
                 n += y.size
             return acc_sum.asscalar() / n
```

Just like the data loader we need to update the training function to deal with GPUs. Unlike `train_ch3 <../chapter_deep-learning-basics/softmax-regression-scratch.md>`__ we now move data prior to computation.

```
In [6]:  # This function has been saved in the d2l package for future use
         def train_ch5(net, train_iter, test_iter, batch_size, trainer, ctx,
                       num_epochs):
             print('training on', ctx)
             loss = gloss.SoftmaxCrossEntropyLoss()
             for epoch in range(num_epochs):
                 train_l_sum, train_acc_sum, n, start = 0.0, 0.0, 0, time.time()
                 for X, y in train_iter:
                     X, y = X.as_in_context(ctx), y.as_in_context(ctx)
                     with autograd.record():
                         y_hat = net(X)
                         l = loss(y_hat, y).sum()
                     l.backward()
                     trainer.step(batch_size)
                     y = y.astype('float32')
```

```
                   train_l_sum += l.asscalar()
                   train_acc_sum += (y_hat.argmax(axis=1) == y).sum().asscalar()
                   n += y.size
               test_acc = evaluate_accuracy(test_iter, net, ctx)
               print('epoch %d, loss %.4f, train acc %.3f, test acc %.3f, '
                     'time %.1f sec'
                     % (epoch + 1, train_l_sum / n, train_acc_sum / n, test_acc,
                        time.time() - start))
```

We initialize the model parameters on the device indicated by `ctx`, this time using Xavier. The loss function and the training algorithm still use the cross-entropy loss function and mini-batch stochastic gradient descent.

```
In [7]: lr, num_epochs = 0.9, 5
        net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
        train_ch5(net, train_iter, test_iter, batch_size, trainer, ctx, num_epochs)

training on gpu(0)
epoch 1, loss 2.3186, train acc 0.103, test acc 0.100, time 4.2 sec
epoch 2, loss 2.2261, train acc 0.145, test acc 0.428, time 3.9 sec
epoch 3, loss 1.0771, train acc 0.565, test acc 0.647, time 3.9 sec
epoch 4, loss 0.7970, train acc 0.688, test acc 0.735, time 3.8 sec
epoch 5, loss 0.6760, train acc 0.735, test acc 0.751, time 3.8 sec
```

Summary

- A convolutional neural network (in short, ConvNet) is a network using convolutional layers.

- In a ConvNet we alternate between convolutions, nonlinearities and often also pooling operations.

- Ultimately the resolution is reduced prior to emitting an output via one (or more) dense layers.

- LeNet was the first successful deployment of such a network.

Exercises

1. Replace the average pooling with max pooling. What happens?

2. Try to construct a more complex network based on LeNet to improve its accuracy.

   - Adjust the convolution window size.

   - Adjust the number of output channels.

   - Adjust the activation function (ReLu?).

   - Adjust the number of convolution layers.

   - Adjust the number of fully connected layers.

   - Adjust the learning rates and other training details (initialization, epochs, etc.)

3. Try out the improved network on the original MNIST dataset.

4. Display the activations of the first and second layer of LeNet for different inputs (e.g. sweaters, coats).

References

[1] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.

Scan the QR Code to Discuss

6.7 Deep Convolutional Neural Networks (AlexNet)

In nearly two decades since LeNet was proposed, for a time, neural networks were surpassed by other machine learning methods, such as Support Vector Machines. Although LeNet achieved good results on early small data sets, its performance on larger real data sets was not satisfactory. Neural network computing is complex. Although some neural network accelerators were available in the 1990s, they were not sufficiently powerful. Therefore, it was difficult to train a multichannel, multilayer convolutional neural network with a large number of parameters in those years. Secondly, datasets were still relatively small. As a result, deep learning research lay mostly dormant. Key techniques such as parameter initialization, non-convex optimization algorithms, activation functions and effective regularization were still missing. The lack of such research was another reason why the training of complex neural networks was very difficult.

One of the key differences to classical computer vision is that we trained the OCR system *end-to-end*. That is, we modeled the entire data analysis pipeline from raw pixels to the classifier output as a single trainable and deformable model. In training the model end-to-end it does away with a lot of the engineering typically required to build a machine learning model. This is quite different from what was the dominant paradigm for machine learning in the 1990s and 2000s.

1. Obtain an interesting dataset that captures relevant aspects e.g. of computer vision. Typically such datasets were hand generated using very expensive sensors (at the time 1 megapixel images were state of the art).

2. Preprocess the dataset using a significant amount of optics, geometry, and analytic tools.

3. Dump the data into a standard set of feature extractors such as SIFT, the Scale-Invariant Feature Transform, or SURF, the Speeded-Up Robust Features, or any number of other hand-tuned pipelines.

4. Dump the resulting representations into a linear model (or a kernel method which generates a linear model in feature space) to solve the machine learning part.

If you spoke to machine learning researchers, they believed that machine learning was both important and beautiful. Elegant theories proved the properties of various classifiers. The field of machine learning was thriving, rigorous and eminently useful. However, if you spoke to a computer vision researcher, you'd hear a very different story. The dirty truth of image recognition, they'd tell you, is that the really important aspects of the ML for CV pipeline were data and features. A slightly cleaner dataset, or a slightly better hand-tuned feature mattered a lot to the final accuracy. However, the specific choice of classifier was little more than an afterthought. At the end of the day you could throw your features in a logistic regression model, a support vector machine, or any other classifier of choice, and they would all perform roughly the same.

6.7.1 Learning Feature Representation

Another way to cast the state of affairs is that the most important part of the pipeline was the representation. And up until 2012, this part was done mechanically, based on some hard-fought intuition. In fact, engineering a new set of feature functions, improving results, and writing up the method was a prominent genre of paper. SIFT, SURF, HOG, Bags of visual words and similar feature extractors ruled the roost.

Another group of researchers had different plans. They believed that features themselves ought to be learned. Moreover they believed that to be reasonably complex, the features ought to be hierarchically composed. These researchers, including Yann LeCun, Geoff Hinton, Yoshua Bengio, Andrew Ng, Shun-ichi Amari, and Juergen Schmidhuber believed that by jointly training many layers of a neural network, they might come to learn hierarchical representations of data. In the case of an image, the lowest layers might come to detect edges, colors, and textures. Indeed, Krizhevski, Sutskever and Hinton, 2012 designed a new variant of a convolutional neural network which achieved excellent performance in the ImageNet challenge. Indeed, it learned good feature extractors in the lower layers. The figure below is reproduced from this paper and it describes lower level image descriptors.
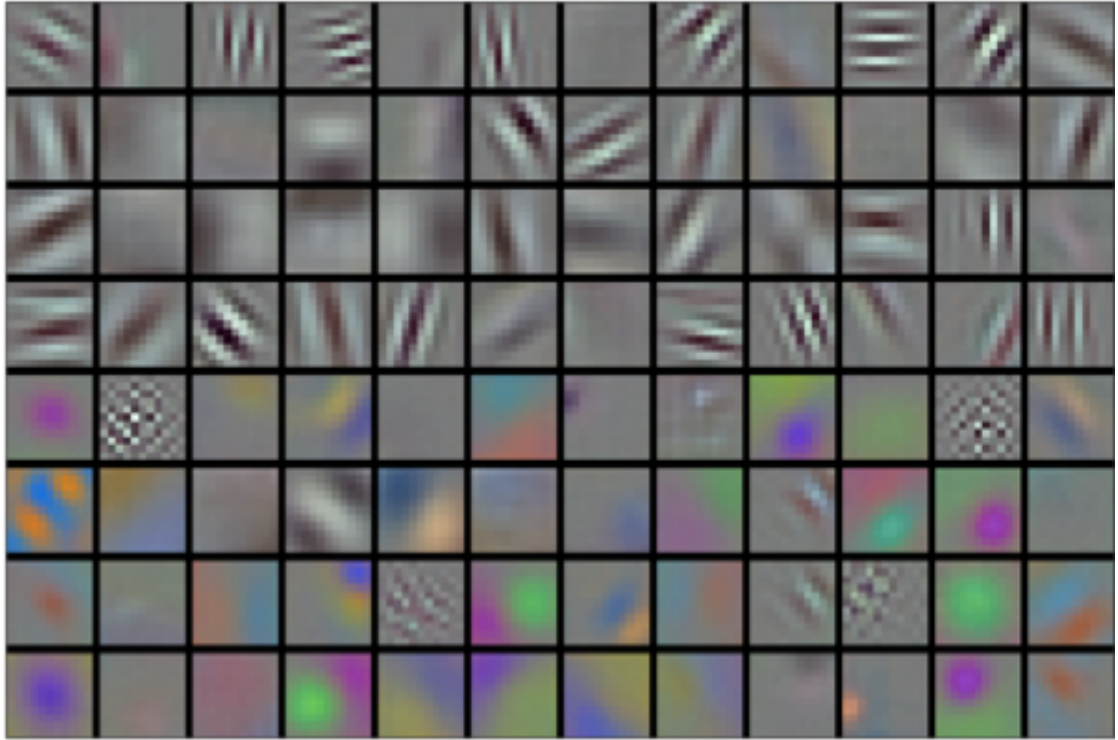
Fig. 6.9: Image filters learned by the first layer of AlexNet

Higher layers might build upon these representations to represent larger structures, like eyes, noses, blades of grass, and features. Yet higher layers might represent whole objects like people, airplanes, dogs, or frisbees. And ultimately, before the classification layer, the final hidden state might represent a compact representation of the image that summarized the contents in a space where data belonging to different categories would be linearly separable. It should be emphasized that the hierarchical representation of the input is determined by the parameters in the multilayer model, and these parameters are all obtained from learning.

Indeed, the visual processing system of animals (and humans) works a bit like that. At its lowest level it contains mostly edge detectors, followed by more structured features. Although researchers dedicated themselves to this idea and attempted to study the hierarchical representation of visual data, their ambitions went unrewarded until 2012. This was due to two key factors.

Missing Ingredient - Data

A deep model with many layers requires a large amount of data to achieve better results than convex models, such as kernel methods. However, given the limited storage capacity of computers, the fact that

sensors were expensive and the comparatively tighter research budgets in the 1990s, most research relied on tiny datasets. For example, many research papers relied on the UCI corpus of datasets, many of which contained hundreds or a few thousand images of low resolution, which were taken in unnatural settings. This situation was improved by the advent of big data around 2010. In particular, the ImageNet data set, which was released in 2009, contains 1,000 categories of objects, each with thousands of different images. This scale was unprecedented. It pushed both computer vision and machine learning research towards deep nonconvex models.

Missing Ingredient - Hardware

Deep Learning has a voracious appetite for computation. This is one of the main reasons why in the 90s and early 2000s algorithms based on convex optimization were the preferred way of solving problems. After all, convex algorithms have fast rates of convergence, global minima, and efficient algorithms can be found.

The game changer was the availability of GPUs. They had long been tuned for graphics processing in computer games. In particular, they were optimized for high throughput 4x4 matrix-vector products, since these are needed for many computer graphics tasks. Fortunately, the math required for that is very similar to convolutional layers in deep networks. Furthermore, around that time, NVIDIA and ATI had begun optimizing GPUs for general compute operations, going as far as renaming them GPGPU (General Purpose GPUs).

To provide some intuition, consider the cores of a modern microprocessor. Each of the cores is quite powerful, running at a high clock frequency, it has quite advanced and large caches (up to several MB of L3). Each core is very good at executing a very wide range of code, with branch predictors, a deep pipeline and lots of other things that make it great at executing regular programs. This apparent strength, however, is also its Achilles heel: general purpose cores are very expensive to build. They require lots of chip area, a sophisticated support structure (memory interfaces, caching logic between cores, high speed interconnects, etc.), and they're comparatively bad at any single task. Modern laptops have up to 4 cores, and even high end servers rarely exceed 64 cores, simply because it is not cost effective.

Compare that with GPUs. They consist of 100-1000 small processing elements (the details differ somewhat between NVIDIA, ATI, ARM and other chip vendors), often grouped into larger groups (NVIDIA calls them warps). While each core is relatively weak, sometimes even running at sub-1GHz clock frequency, it is the total number of such cores that makes GPUs orders of magnitude faster than CPUs. For instance, NVIDIA's latest Volta generation offers up to 120 TFlops per chip for specialized instructions (and up to 24 TFlops for more general purpose ones), while floating point performance of CPUs has not exceeded 1 TFlop to date. The reason for why this is possible is actually quite simple: firstly, power consumption tends to grow *quadratically* with clock frequency. Hence, for the power budget of a CPU core that runs 4x faster (a typical number) you can use 16 GPU cores at 1/4 the speed, which yields 16 x 1/4 = 4x the performance. Furthermore GPU cores are much simpler (in fact, for a long time they weren't even *able* to execute general purpose code), which makes them more energy efficient. Lastly, many operations in deep learning require high memory bandwidth. Again, GPUs shine here with buses that are at least 10x as wide as many CPUs.

Back to 2012. A major breakthrough came when Alex Krizhevsky and Ilya Sutskever implemented a deep

convolutional neural network that could run on GPU hardware. They realized that the computational bottlenecks in CNNs (convolutions and matrix multiplications) are all operations that could be parallelized in hardware. Using two NIVIDA GTX 580s with 3GB of memory they implemented fast convolutions. The code cuda-convnet was good enough that for several years it was the industry standard and powered the first couple years of the deep learning boom.

6.7.2 AlexNet

AlexNet was introduced in 2012, named after Alex Krizhevsky, the first author of the eponymous paper. AlexNet uses an 8-layer convolutional neural network and won the ImageNet Large Scale Visual Recognition Challenge 2012 with a large margin. This network proved, for the first time, that the features obtained by learning can transcend manually-design features, breaking the previous paradigm in computer vision. The architectures of AlexNet and LeNet are *very similar*, as the diagram below illustrates. Note that we provide a slightly streamlined version of AlexNet which removes the quirks that were needed in 2012 to make the model fit on two small GPUs.
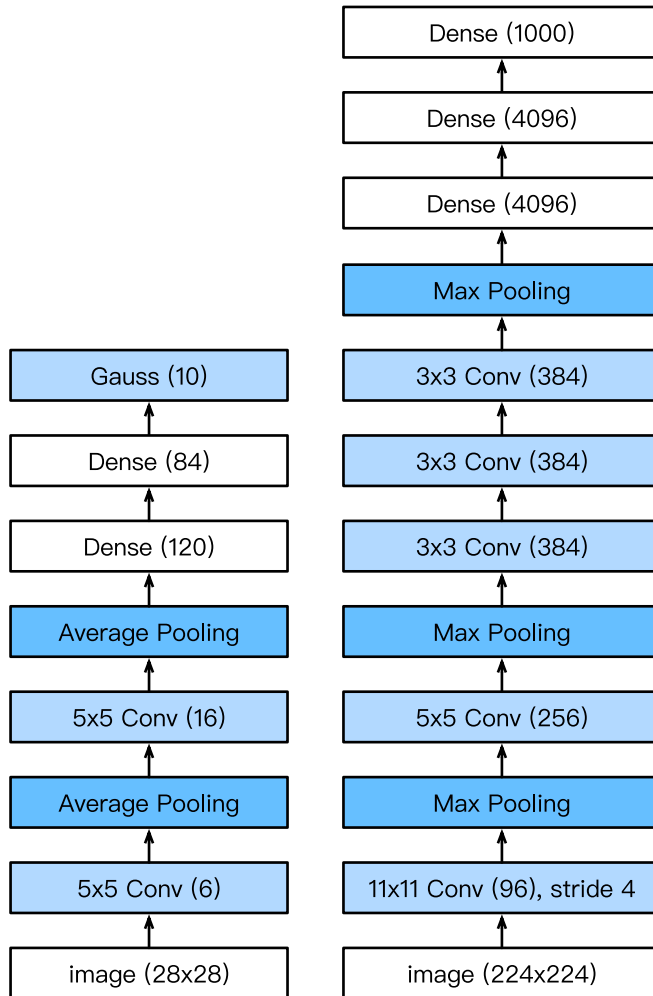
Fig. 6.10: LeNet (left) and AlexNet (right)

The design philosophies of AlexNet and LeNet are very similar, but there are also significant differences. First, AlexNet is much deeper than the comparatively small LeNet5. AlexNet consists of eight layers, five convolutional layers, two fully connected hidden layers, and one fully connected output layer. Second, AlexNet used the ReLu instead of the sigmoid as its activation function. This improved convergence during training significantly. Let's delve into the details below.

Architecture

In AlexNet's first layer, the convolution window shape is 11×11. Since most images in ImageNet are more than ten times higher and wider than the MNIST images, objects in ImageNet images take up more pixels. Consequently, a larger convolution window is needed to capture the object. The convolution window shape in the second layer is reduced to 5×5, followed by 3×3. In addition, after the first, second, and fifth convolutional layers, the network adds maximum pooling layers with a window shape of 3×3 and a stride of 2. Moreover, AlexNet has ten times more convolution channels than LeNet.

After the last convolutional layer are two fully connected layers with 4096 outputs. These two huge fully connected layers produce model parameters of nearly 1 GB. Due to the limited memory in early GPUs, the original AlexNet used a dual data stream design, so that one GPU only needs to process half of the model. Fortunately, GPU memory has developed tremendously over the past few years, so we usually do not need this special design anymore (our model deviates from the original paper in this aspect).

Activation Functions

Second, AlexNet changed the sigmoid activation function to a simpler ReLU activation function. On the one hand, the computation of the ReLU activation function is simpler. For example, it does not have the exponentiation operation found in the sigmoid activation function. On the other hand, the ReLU activation function makes model training easier when using different parameter initialization methods. This is because, when the output of the sigmoid activation function is very close to 0 or 1, the gradient of these regions is almost 0, so that back propagation cannot continue to update some of the model parameters. In contrast, the gradient of the ReLU activation function in the positive interval is always 1. Therefore, if the model parameters are not properly initialized, the sigmoid function may obtain a gradient of almost 0 in the positive interval, so that the model cannot be effectively trained.

Capacity Control and Preprocessing

AlexNet controls the model complexity of the fully connected layer by dropout section), while LeNet only uses weight decay. To augment the data even further, the training loop of AlexNet added a great deal of image augmentation, such as flipping, clipping, and color changes. This makes the model more robust and the larger sample size effectively reduces overfitting. We will discuss preprocessing in detail in a *subsequent section*.

```
In [1]: import sys
        sys.path.insert(0, '..')

        import d2l
        from mxnet import gluon, init, nd
        from mxnet.gluon import data as gdata, nn
        import os
        import sys

        net = nn.Sequential()
        # Here, we use a larger 11 x 11 window to capture objects. At the same time,
        # we use a stride of 4 to greatly reduce the height and width of the output.
```

```
            # Here, the number of output channels is much larger than that in LeNet
    net.add(nn.Conv2D(96, kernel_size=11, strides=4, activation='relu'),
            nn.MaxPool2D(pool_size=3, strides=2),
            # Make the convolution window smaller, set padding to 2 for consistent
            # height and width across the input and output, and increase the
            # number of output channels
            nn.Conv2D(256, kernel_size=5, padding=2, activation='relu'),
            nn.MaxPool2D(pool_size=3, strides=2),
            # Use three successive convolutional layers and a smaller convolution
            # window. Except for the final convolutional layer, the number of
            # output channels is further increased. Pooling layers are not used to
            # reduce the height and width of input after the first two
            # convolutional layers
            nn.Conv2D(384, kernel_size=3, padding=1, activation='relu'),
            nn.Conv2D(384, kernel_size=3, padding=1, activation='relu'),
            nn.Conv2D(256, kernel_size=3, padding=1, activation='relu'),
            nn.MaxPool2D(pool_size=3, strides=2),
            # Here, the number of outputs of the fully connected layer is several
            # times larger than that in LeNet. Use the dropout layer to mitigate
            # overfitting
            nn.Dense(4096, activation="relu"), nn.Dropout(0.5),
            nn.Dense(4096, activation="relu"), nn.Dropout(0.5),
            # Output layer. Since we are using Fashion-MNIST, the number of
            # classes is 10, instead of 1000 as in the paper
            nn.Dense(10))
```

We construct a single-channel data instance with both height and width of 224 to observe the output shape of each layer. It matches our diagram above.

```
In [2]: X = nd.random.uniform(shape=(1, 1, 224, 224))
        net.initialize()
        for layer in net:
            X = layer(X)
            print(layer.name, 'output shape:\t', X.shape)
```

```
conv0 output shape:        (1, 96, 54, 54)
pool0 output shape:        (1, 96, 26, 26)
conv1 output shape:        (1, 256, 26, 26)
pool1 output shape:        (1, 256, 12, 12)
conv2 output shape:        (1, 384, 12, 12)
conv3 output shape:        (1, 384, 12, 12)
conv4 output shape:        (1, 256, 12, 12)
pool2 output shape:        (1, 256, 5, 5)
dense0 output shape:       (1, 4096)
dropout0 output shape:     (1, 4096)
dense1 output shape:       (1, 4096)
dropout1 output shape:     (1, 4096)
dense2 output shape:       (1, 10)
```

6.7.3 Reading Data

Although AlexNet uses ImageNet in the paper, we use Fashion-MNIST. This is simply since training on ImageNet would take hours even on modern GPUs. One of the problems with applying AlexNet directly is that the images are simply too low resolution at 28×28 pixels. To make things work we upsample them

to 244×244 (this is generally not very smart but we do so to illustrate network performance). This can be done with the `Resize` class. We insert it into the processing pipeline before using the `ToTensor` class. The `Compose` class to concatenates these two changes for easy invocation.

```
In [3]: # This function has been saved in the d2l package for future use
        def load_data_fashion_mnist(batch_size, resize=None, root=os.path.join(
                '~', '.mxnet', 'datasets', 'fashion-mnist')):
            root = os.path.expanduser(root)  # Expand the user path '~'.
            transformer = []
            if resize:
                transformer += [gdata.vision.transforms.Resize(resize)]
            transformer += [gdata.vision.transforms.ToTensor()]
            transformer = gdata.vision.transforms.Compose(transformer)
            mnist_train = gdata.vision.FashionMNIST(root=root, train=True)
            mnist_test = gdata.vision.FashionMNIST(root=root, train=False)
            num_workers = 0 if sys.platform.startswith('win32') else 4
            train_iter = gdata.DataLoader(
                mnist_train.transform_first(transformer), batch_size, shuffle=True,
                num_workers=num_workers)
            test_iter = gdata.DataLoader(
                mnist_test.transform_first(transformer), batch_size, shuffle=False,
                num_workers=num_workers)
            return train_iter, test_iter

        batch_size = 128
        train_iter, test_iter = load_data_fashion_mnist(batch_size, resize=224)
```

6.7.4 Training

Now, we can start training AlexNet. Compared to LeNet in the previous section, the main change here is the use of a smaller learning rate and much slower training due to the deeper and wider network, the higher image resolution and the more costly convolutions.

```
In [4]: lr, num_epochs, ctx = 0.01, 5, d2l.try_gpu()
        net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
        d2l.train_ch5(net, train_iter, test_iter, batch_size, trainer, ctx,
                      num_epochs)

training on gpu(0)
epoch 1, loss 1.3132, train acc 0.509, test acc 0.739, time 70.3 sec
epoch 2, loss 0.6474, train acc 0.758, test acc 0.811, time 65.8 sec
epoch 3, loss 0.5285, train acc 0.803, test acc 0.826, time 65.8 sec
epoch 4, loss 0.4671, train acc 0.828, test acc 0.861, time 65.9 sec
epoch 5, loss 0.4241, train acc 0.843, test acc 0.862, time 65.8 sec
```

Summary

- AlexNet has a similar structure to that of LeNet, but uses more convolutional layers and a larger parameter space to fit the large-scale data set ImageNet.

- Today AlexNet has been surpassed by much more effective architectures but it is a key step from shallow to deep networks that are used nowadays.

- Although it seems that there are only a few more lines in AlexNet's implementation than in LeNet, it took the academic community many years to embrace this conceptual change and take advantage of its excellent experimental results. This was also due to the lack of efficient computational tools.

- Dropout, ReLu and preprocessing were the other key steps in achieving excellent performance in computer vision tasks.

Exercises

1. Try increasing the number of epochs. Compared with LeNet, how are the results different? Why?

2. AlexNet may be too complex for the Fashion-MNIST data set.

    - Try to simplify the model to make the training faster, while ensuring that the accuracy does not drop significantly.

    - Can you design a better model that works directly on 28×28 images.

3. Modify the batch size, and observe the changes in accuracy and GPU memory.

4. Rooflines

    - What is the dominant part for the memory footprint of AlexNet?

    - What is the dominant part for computation in AlexNet?

    - How about memory bandwidth when computing the results?

5. Apply dropout and ReLu to LeNet5. Does it improve? How about preprocessing?

Scan the QR Code to Discuss

6.8 Networks Using Blocks (VGG)

AlexNet adds three convolutional layers to LeNet. Beyond that, the authors of AlexNet made significant adjustments to the convolution windows, the number of output channels, nonlinear activation, and regularization. Although AlexNet proved that deep convolutional neural networks can achieve good results,

it does not provide simple rules to guide subsequent researchers in the design of new networks. In the following sections, we will introduce several different concepts used in deep network design.

Progress in this field mirrors that in chip design where engineers went from placing transistors (neurons) to logical elements (layers) to logic blocks (the topic of the current section). The idea of using blocks was first proposed by the Visual Geometry Group (VGG) at Oxford University. This led to the VGG network, which we will be discussing below. When using a modern deep learning framework repeated structures can be expressed as *code* with for loops and subroutines. Just like we would use a for loop to count from 1 to 10, we'll use code to combine layers.

6.8.1 VGG Blocks

The basic building block of a ConvNet is the combination of a convolutional layer (with padding to keep the resolution unchanged), followed by a nonlinearity such as a ReLu. A VGG block is given by a sequence of such layers, followed by maximum pooling. Throughout their design Simonyan and Ziserman, 2014 used convolution windows of size 3 and maximum poolin with stride and window width 2, effectively halving the resolution after each block. We use the vgg_block function to implement this basic VGG block. This function takes the number of convolutional layers num_convs and the number of output channels num_channels as input.

```
In [1]: import sys
        sys.path.insert(0, '..')

        import d2l
        from mxnet import gluon, init, nd
        from mxnet.gluon import nn

        def vgg_block(num_convs, num_channels):
            blk = nn.Sequential()
            for _ in range(num_convs):
                blk.add(nn.Conv2D(num_channels, kernel_size=3,
                                  padding=1, activation='relu'))
            blk.add(nn.MaxPool2D(pool_size=2, strides=2))
            return blk
```

6.8.2 VGG Network

Like AlexNet and LeNet, the VGG Network is composed of convolutional layer modules attached to fully connected layers. Several vgg_block modules are connected in series in the convolutional layer module, the hyper-parameter of which is defined by the variable conv_arch. This variable specifies the numbers of convolutional layers and output channels in each VGG block. The fully connected module is the same as that of AlexNet.
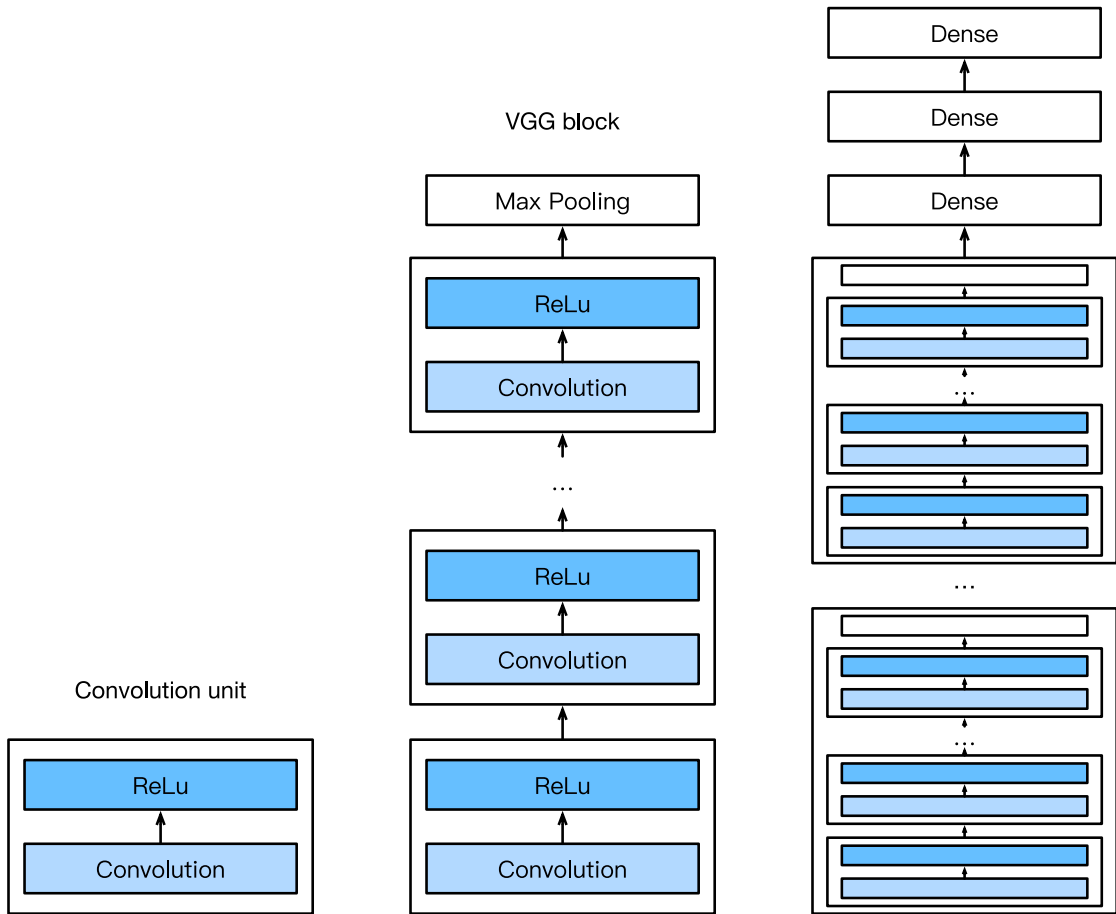
Fig. 6.11: Designing a network from building blocks

The VGG network proposed by Simonyan and Ziserman has 5 convolutional blocks, among which the former two use a single convolutional layer, while the latter three use a double convolutional layer. The first block has 64 output channels, and the latter blocks double the number of output channels, until that number reaches 512. Since this network uses 8 convolutional layers and 3 fully connected layers, it is often called VGG-11.

```
In [2]: conv_arch = ((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))
```

Now, we will implement VGG-11. This is a simple matter of executing a for loop over `conv_arch`.

```
In [3]: def vgg(conv_arch):
            net = nn.Sequential()
            # The convolutional layer part
            for (num_convs, num_channels) in conv_arch:
                net.add(vgg_block(num_convs, num_channels))
```

```
        # The fully connected layer part
        net.add(nn.Dense(4096, activation='relu'), nn.Dropout(0.5),
                nn.Dense(4096, activation='relu'), nn.Dropout(0.5),
                nn.Dense(10))
        return net

    net = vgg(conv_arch)
```

Next, we will construct a single-channel data example with a height and width of 224 to observe the output shape of each layer.

```
In [4]: net.initialize()
        X = nd.random.uniform(shape=(1, 1, 224, 224))
        for blk in net:
            X = blk(X)
            print(blk.name, 'output shape:\t', X.shape)

sequential1 output shape:        (1, 64, 112, 112)
sequential2 output shape:        (1, 128, 56, 56)
sequential3 output shape:        (1, 256, 28, 28)
sequential4 output shape:        (1, 512, 14, 14)
sequential5 output shape:        (1, 512, 7, 7)
dense0 output shape:     (1, 4096)
dropout0 output shape:   (1, 4096)
dense1 output shape:     (1, 4096)
dropout1 output shape:   (1, 4096)
dense2 output shape:     (1, 10)
```

As we can see, we halve the entered value of the height and width each time, until the final values of height and width change to 7 before we pass it to the fully connected layer. Meanwhile, the number of output channels doubles until it becomes 512. Since the windows of each convolutional layer are of the same size, the model parameter size of each layer and the computational complexity is proportional to the product of height, width, number of input channels, and number of output channels. By halving the height and width while doubling the number of channels, VGG allows most convolutional layers to have the same model activation size and computational complexity.

6.8.3 Model Training

Since VGG-11 is more complicated than AlexNet in terms of computation, we construct a network with a smaller number of channels. This is more than sufficient for training on Fashion-MNIST.

```
In [5]: ratio = 4
        small_conv_arch = [(pair[0], pair[1] // ratio) for pair in conv_arch]
        net = vgg(small_conv_arch)
```

Apart from using a slightly larger learning rate, the model training process is similar to that of AlexNet in the last section.

```
In [6]: lr, num_epochs, batch_size, ctx = 0.05, 5, 128, d2l.try_gpu()
        net.initialize(ctx=ctx, init=init.Xavier())
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
        train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
        d2l.train_ch5(net, train_iter, test_iter, batch_size, trainer, ctx,
                      num_epochs)
```

```
training on gpu(0)
epoch 1, loss 0.9409, train acc 0.656, test acc 0.838, time 141.1 sec
epoch 2, loss 0.4147, train acc 0.848, test acc 0.877, time 135.1 sec
epoch 3, loss 0.3376, train acc 0.876, test acc 0.897, time 135.2 sec
epoch 4, loss 0.2984, train acc 0.891, test acc 0.906, time 135.1 sec
epoch 5, loss 0.2651, train acc 0.902, test acc 0.905, time 135.1 sec
```

Summary

- VGG-11 constructs a network using reusable convolutional blocks. Different VGG models can be defined by the differences in the number of convolutional layers and output channels in each block.

- The use of blocks leads to very compact representations of the network definition. It allows for efficient design of complex networks.

- In their work Simonyan and Ziserman experimented with various architectures. In particular, they found that several layers of deep and narrow convolutions (i.e. 3×3) were more effective than fewer layers of wider convolutions.

Exercises

1. When printing out the dimensions of the layers we only saw 8 results rather than 11. Where did the remaining 3 layer informations go?

2. Compared with AlexNet, VGG is much slower in terms of computation, and it also needs more GPU memory. Try to analyze the reasons for this.

3. Try to change the height and width of the images in Fashion-MNIST from 224 to 96. What influence does this have on the experiments?

4. Refer to Table 1 in the original VGG Paper to construct other common models, such as VGG-16 or VGG-19.

Scan the QR Code to Discuss

6.9 Network in Network (NiN)

LeNet, AlexNet, and VGG all share a common design pattern: extract the spatial features through a sequence of convolutions and pooling layers and then post-process the representations via fully connected layers. The improvements upon LeNet by AlexNet and VGG mainly lie in how these later networks widen and deepen these two modules. An alternative is to use fully connected layers much earlier in the process. However, a careless use of a dense layer would destroy the spatial structure of the data entirely, since fully connected layers mangle all inputs. Network in Network (NiN) blocks offer an alternative. They were proposed by Lin, Chen and Yan, 2013 based on a very simple insight - to use an MLP on the channels for each pixel separately.

6.9.1 NiN Blocks

We know that the inputs and outputs of convolutional layers are usually four-dimensional arrays (example, channel, height, width), while the inputs and outputs of fully connected layers are usually two-dimensional arrays (example, feature). This means that once we process data by a fully connected layer it's virtually impossible to recover the spatial structure of the representation. But we could apply a fully connected layer at a pixel level: Recall the 1×1 convolutional layer described in the section discussing *channels*. This somewhat unusual convolution can be thought of as a fully connected layer processing channel activations on a per pixel level. Another way to view this is to think of each element in the spatial dimension (height and width) as equivalent to an example, and the channel as equivalent to a feature. NiNs use the 1×1 convolutional layer instead of a fully connected layer. The spatial information can then be naturally passed to the subsequent layers. The figure below illustrates the main structural differences between NiN and AlexNet, VGG, and other networks.
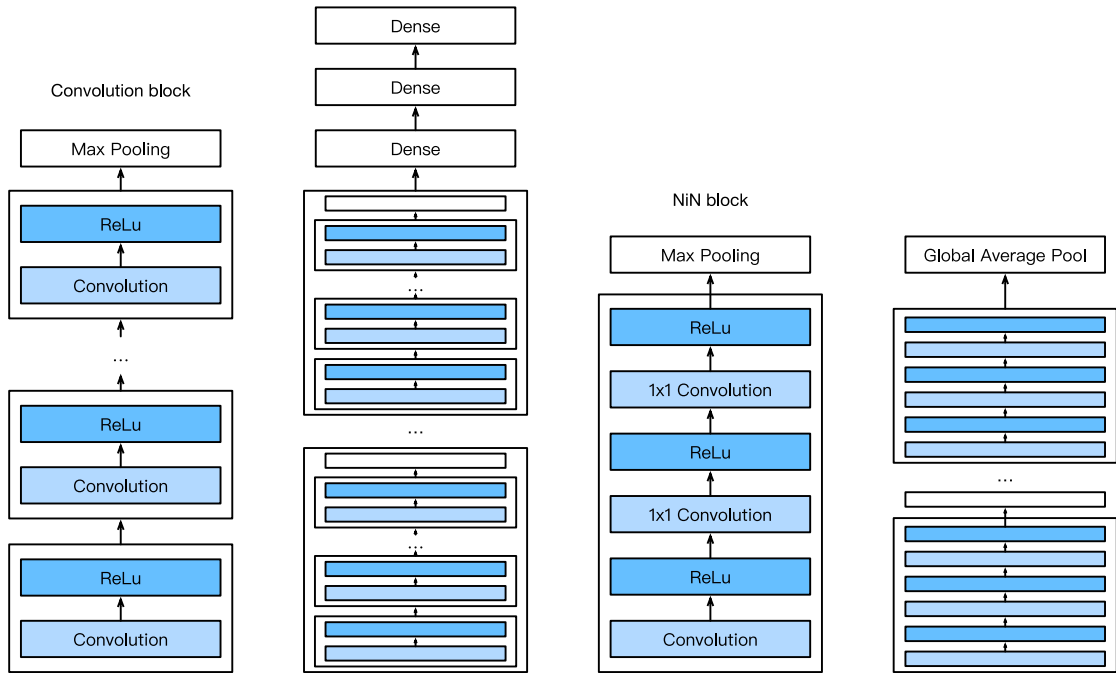
Fig. 6.12: The figure on the left shows the network structure of AlexNet and VGG, and the figure on the right shows the network structure of NiN.

The NiN block is the basic block in NiN. It concatenates a convolutional layer and two 1×1 convolutional layers that act as fully connected layers (with ReLu in between). The convolution width of the first layer is typically set by the user. The subsequent widths are fixed to 1×1.

```
In [1]: import sys
        sys.path.insert(0, '..')

        import d2l
        from mxnet import gluon, init, nd
        from mxnet.gluon import nn

        def nin_block(num_channels, kernel_size, strides, padding):
            blk = nn.Sequential()
            blk.add(nn.Conv2D(num_channels, kernel_size, strides, padding,
↪   activation='relu'),
                    nn.Conv2D(num_channels, kernel_size=1, activation='relu'),
                    nn.Conv2D(num_channels, kernel_size=1, activation='relu'))
            return blk
```

6.9.2 NiN Model

NiN was proposed shortly after the release of AlexNet. Their convolutional layer settings share some similarities. NiN uses convolutional layers with convolution window shapes of 11×11, 5×5, and 3×3, and the corresponding numbers of output channels are the same as in AlexNet. Each NiN block is followed by a maximum pooling layer with a stride of 2 and a window shape of 3×3.

In addition to using NiN blocks, NiN's design is significantly different from AlexNet by avoiding dense connections entirely: Instead, NiN uses a NiN block with a number of output channels equal to the number of label classes, and then uses a global average pooling layer to average all elements in each channel for direct use in classification. Here, the global average pooling layer, i.e. the window shape, is equal to the average pooling layer of the input spatial dimension shape. The advantage of NiN's design is that it can significantly reduce the size of model parameters, thus mitigating overfitting. In other words, short of the average pooling all operations are convolutions. However, this design sometimes results in an increase in model training time.

```
In [2]: net = nn.Sequential()
        net.add(nin_block(96, kernel_size=11, strides=4, padding=0),
                nn.MaxPool2D(pool_size=3, strides=2),
                nin_block(256, kernel_size=5, strides=1, padding=2),
                nn.MaxPool2D(pool_size=3, strides=2),
                nin_block(384, kernel_size=3, strides=1, padding=1),
                nn.MaxPool2D(pool_size=3, strides=2),
                nn.Dropout(0.5),
                # There are 10 label classes
                nin_block(10, kernel_size=3, strides=1, padding=1),
                # The global average pooling layer automatically sets the window shape
                # to the height and width of the input
                nn.GlobalAvgPool2D(),
                # Transform the four-dimensional output into two-dimensional output
                # with a shape of (batch size, 10)
                nn.Flatten())
```

We create a data example to see the output shape of each block.

```
In [3]: X = nd.random.uniform(shape=(1, 1, 224, 224))
        net.initialize()
        for layer in net:
            X = layer(X)
            print(layer.name, 'output shape:\t', X.shape)

sequential1 output shape:        (1, 96, 54, 54)
pool0 output shape:       (1, 96, 26, 26)
sequential2 output shape:        (1, 256, 26, 26)
pool1 output shape:       (1, 256, 12, 12)
sequential3 output shape:        (1, 384, 12, 12)
pool2 output shape:       (1, 384, 5, 5)
dropout0 output shape:    (1, 384, 5, 5)
sequential4 output shape:        (1, 10, 5, 5)
pool3 output shape:       (1, 10, 1, 1)
flatten0 output shape:    (1, 10)
```

6.9.3 Data Acquisition and Training

As before we use Fashion-MNIST to train the model. NiN's training is similar to that for AlexNet and VGG, but it often uses a larger learning rate.

```
In [4]: lr, num_epochs, batch_size, ctx = 0.1, 5, 128, d2l.try_gpu()
        net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
        train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=224)
        d2l.train_ch5(net, train_iter, test_iter, batch_size, trainer, ctx,
                       num_epochs)

training on gpu(0)
epoch 1, loss 2.2583, train acc 0.181, test acc 0.328, time 100.9 sec
epoch 2, loss 1.5810, train acc 0.441, test acc 0.702, time 96.4 sec
epoch 3, loss 0.7845, train acc 0.724, test acc 0.782, time 96.4 sec
epoch 4, loss 0.6211, train acc 0.781, test acc 0.808, time 96.3 sec
epoch 5, loss 0.5242, train acc 0.813, test acc 0.807, time 96.3 sec
```

Summary

- NiN uses blocks consisting of a convolutional layer and multiple 1×1 convolutional layer. This can be used within the convolutional stack to allow for more per-pixel nonlinearity.

- NiN removes the fully connected layers and replaces them with global average pooling (i.e. summing over all locations) after reducing the number of channels to the desired number of outputs (e.g. 10 for Fashion-MNIST).

- Removing the dense layers reduces overfitting. NiN has dramatically fewer parameters.

- The NiN design influenced many subsequent convolutional neural networks designs.

Exercises

1. Tune the hyper-parameters to improve the classification accuracy.

2. Why are there two 1×1 convolutional layers in the NiN block? Remove one of them, and then observe and analyze the experimental phenomena.

3. Calculate the resource usage for NiN

   - What is the number of parameters?

   - What is the amount of computation?

   - What is the amount of memory needed during training?

   - What is the amount of memory needed during inference?

4. What are possible problems with reducing the $384 \times 5 \times 5$ representation to a $10 \times 5 \times 5$ representation in one step?

6.10 Networks with Parallel Concatenations (GoogLeNet)

During the ImageNet Challenge in 2014, a new architecture emerged that outperformed the rest. Szegedy et al., 2014 proposed a structure that combined the strengths of the NiN and repeated blocks paradigms. At its heart was the rather pragmatic answer to the question as to which size of convolution is ideal for processing. After all, we have a smorgasbord of choices, 1×1 or 3×3, 5×5 or even larger. And it isn't always clear which one is the best. As it turns out, the answer is that a combination of all the above works best. Over the next few years, researchers made several improvements to GoogLeNet. In this section, we will introduce the first version of this model series in a slightly simplified form - we omit the peculiarities that were added to stabilize training, due to the availability of better training algorithms.

6.10.1 Inception Blocks

The basic convolutional block in GoogLeNet is called an Inception block, named after the movie of the same name. This basic block is more complex in structure than the NiN block described in the previous section.
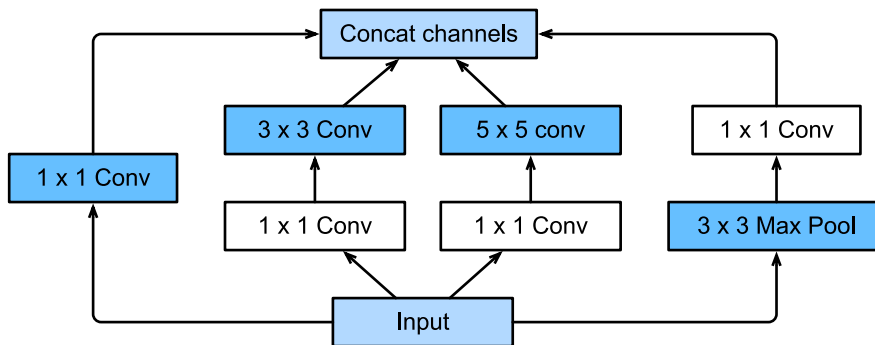
Fig. 6.13: Structure of the Inception block.

As can be seen in the figure above, there are four parallel paths in the Inception block. The first three paths use convolutional layers with window sizes of 1×1, 3×3, and 5×5 to extract information from different

spatial sizes. The middle two paths will perform a 1×1 convolution on the input to reduce the number of input channels, so as to reduce the model's complexity. The fourth path uses the 3×3 maximum pooling layer, followed by the 1×1 convolutional layer, to change the number of channels. The four paths all use appropriate padding to give the input and output the same height and width. Finally, we concatenate the output of each path on the channel dimension and input it to the next layer. The customizable parameters of the Inception block are the number of output channels per layer, which can be used to control the model complexity.

```
In [1]: import sys
        sys.path.insert(0, '..')

        import d2l
        from mxnet import gluon, init, nd
        from mxnet.gluon import nn

        class Inception(nn.Block):
            # c1 - c4 are the number of output channels for each layer in the path
            def __init__(self, c1, c2, c3, c4, **kwargs):
                super(Inception, self).__init__(**kwargs)
                # Path 1 is a single 1 x 1 convolutional layer
                self.p1_1 = nn.Conv2D(c1, kernel_size=1, activation='relu')
                # Path 2 is a 1 x 1 convolutional layer followed by a 3 x 3
                # convolutional layer
                self.p2_1 = nn.Conv2D(c2[0], kernel_size=1, activation='relu')
                self.p2_2 = nn.Conv2D(c2[1], kernel_size=3, padding=1,
                                      activation='relu')
                # Path 3 is a 1 x 1 convolutional layer followed by a 5 x 5
                # convolutional layer
                self.p3_1 = nn.Conv2D(c3[0], kernel_size=1, activation='relu')
                self.p3_2 = nn.Conv2D(c3[1], kernel_size=5, padding=2,
                                      activation='relu')
                # Path 4 is a 3 x 3 maximum pooling layer followed by a 1 x 1
                # convolutional layer
                self.p4_1 = nn.MaxPool2D(pool_size=3, strides=1, padding=1)
                self.p4_2 = nn.Conv2D(c4, kernel_size=1, activation='relu')

            def forward(self, x):
                p1 = self.p1_1(x)
                p2 = self.p2_2(self.p2_1(x))
                p3 = self.p3_2(self.p3_1(x))
                p4 = self.p4_2(self.p4_1(x))
                # Concatenate the outputs on the channel dimension
                return nd.concat(p1, p2, p3, p4, dim=1)
```

To understand why this works as well as it does, consider the combination of the filters. They explore the image in varying ranges. This means that details at different extents can be recognized efficiently by different filters. At the same time, we can allocate different amounts of parameters for different ranges (e.g. more for short range but not ignore the long range entirely).

6.10.2 GoogLeNet Model

GoogLeNet uses an initial long range feature convolution, a stack of a total of 9 inception blocks and global average pooling to generate its estimates. Maximum pooling between inception blocks reduced the dimensionality. The first part is identical to AlexNet and LeNet, the stack of blocks is inherited from VGG and the global average pooling that avoids a stack of fully connected layers at the end. The architecture is depicted below.
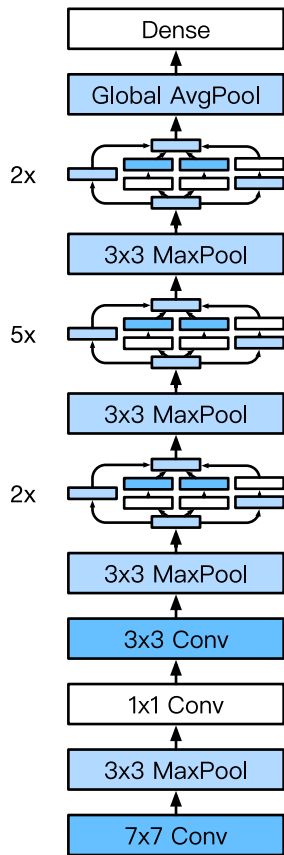
Fig. 6.14: Full GoogLeNet Model

Let's build the network piece by piece. The first block uses a 64-channel 7Œ7 convolutional layer.

```
In [2]: b1 = nn.Sequential()
        b1.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3, activation='relu'),
               nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

The second block uses two convolutional layers: first, a 64-channel 1×1 convolutional layer, then a 3×3 convolutional layer that triples the number of channels. This corresponds to the second path in the

Inception block.

```
In [3]: b2 = nn.Sequential()
        b2.add(nn.Conv2D(64, kernel_size=1),
               nn.Conv2D(192, kernel_size=3, padding=1),
               nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

The third block connects two complete Inception blocks in series. The number of output channels of the first Inception block is $64 + 128 + 32 + 32 = 256$, and the ratio to the output channels of the four paths is $64 : 128 : 32 : 32 = 2 : 4 : 1 : 1$. The second and third paths first reduce the number of input channels to $96/192 = 1/2$ and $16/192 = 1/12$, respectively, and then connect the second convolutional layer. The number of output channels of the second Inception block is increased to $128 + 192 + 96 + 64 = 480$, and the ratio to the number of output channels per path is $128 : 192 : 96 : 64 = 4 : 6 : 3 : 2$. The second and third paths first reduce the number of input channels to $128/256 = 1/2$ and $32/256 = 1/8$, respectively.

```
In [4]: b3 = nn.Sequential()
        b3.add(Inception(64, (96, 128), (16, 32), 32),
               Inception(128, (128, 192), (32, 96), 64),
               nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

The fourth block is more complicated. It connects five Inception blocks in series, and they have $192 + 208 + 48 + 64 = 512$, $160 + 224 + 64 + 64 = 512$, $128 + 256 + 64 + 64 = 512$, $112 + 288 + 64 + 64 = 528$, and $256 + 320 + 128 + 128 = 832$ output channels, respectively. The number of channels assigned to these paths is similar to that in the third module: the second path with the 3×3 convolutional layer outputs the largest number of channels, followed by the first path with only the 1×1 convolutional layer, the third path with the 5×5 convolutional layer, and the fourth path with the 3×3 maximum pooling layer. The second and third paths will first reduce the number of channels according the ratio. These ratios are slightly different in different Inception blocks.

```
In [5]: b4 = nn.Sequential()
        b4.add(Inception(192, (96, 208), (16, 48), 64),
               Inception(160, (112, 224), (24, 64), 64),
               Inception(128, (128, 256), (24, 64), 64),
               Inception(112, (144, 288), (32, 64), 64),
               Inception(256, (160, 320), (32, 128), 128),
               nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

The fifth block has two Inception blocks with $256 + 320 + 128 + 128 = 832$ and $384 + 384 + 128 + 128 = 1024$ output channels. The number of channels assigned to each path is the same as that in the third and fourth modules, but differs in specific values. It should be noted that the fifth block is followed by the output layer. This block uses the global average pooling layer to change the height and width of each channel to 1, just as in NiN. Finally, we turn the output into a two-dimensional array followed by a fully connected layer whose number of outputs is the number of label classes.

```
In [6]: b5 = nn.Sequential()
        b5.add(Inception(256, (160, 320), (32, 128), 128),
               Inception(384, (192, 384), (48, 128), 128),
               nn.GlobalAvgPool2D())

        net = nn.Sequential()
        net.add(b1, b2, b3, b4, b5, nn.Dense(10))
```

The GoogLeNet model is computationally complex, so it is not as easy to modify the number of channels

as in VGG. To have a reasonable training time on Fashion-MNIST we reduce the input height and width from 224 to 96. This simplifies the computation. The changes in the shape of the output between the various modules is demonstrated below.

```
In [7]: X = nd.random.uniform(shape=(1, 1, 96, 96))
        net.initialize()
        for layer in net:
            X = layer(X)
            print(layer.name, 'output shape:\t', X.shape)

sequential0 output shape:        (1, 64, 24, 24)
sequential1 output shape:        (1, 192, 12, 12)
sequential2 output shape:        (1, 480, 6, 6)
sequential3 output shape:        (1, 832, 3, 3)
sequential4 output shape:        (1, 1024, 1, 1)
dense0 output shape:     (1, 10)
```

6.10.3 Data Acquisition and Training

As before, we train our model using the Fashion-MNIST dataset. We transform it to 96×96 pixel resolution before invoking the training procedure.

```
In [8]: lr, num_epochs, batch_size, ctx = 0.1, 5, 128, d2l.try_gpu()
        net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
        train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
        d2l.train_ch5(net, train_iter, test_iter, batch_size, trainer, ctx,
                      num_epochs)

training on gpu(0)
epoch 1, loss 1.6887, train acc 0.379, test acc 0.726, time 83.6 sec
epoch 2, loss 0.6435, train acc 0.763, test acc 0.812, time 73.9 sec
epoch 3, loss 0.4320, train acc 0.837, test acc 0.866, time 73.8 sec
epoch 4, loss 0.3921, train acc 0.853, test acc 0.861, time 73.9 sec
epoch 5, loss 0.3341, train acc 0.874, test acc 0.880, time 74.0 sec
```

Summary

- The Inception block is equivalent to a subnetwork with four paths. It extracts information in parallel through convolutional layers of different window shapes and maximum pooling layers. 1×1 convolutions reduce channel dimensionality on a per-pixel level. Max-pooling reduces the resolution.
- GoogLeNet connects multiple well-designed Inception blocks with other layers in series. The ratio of the number of channels assigned in the Inception block is obtained through a large number of experiments on the ImageNet data set.
- GoogLeNet, as well as its succeeding versions, was one of the most efficient models on ImageNet, providing similar test accuracy with lower computational complexity.

Exercises

1. There are several iterations of GoogLeNet. Try to implement and run them. Some of them include the following:

   - Add a batch normalization layer, as described later in this chapter [2].

   - Make adjustments to the Inception block [3].

   - Use label smoothing for model regularization [3].

   - Include it in the residual connection, as described later in this chapter [4].

2. What is the minimum image size for GoogLeNet to work?

3. Compare the model parameter sizes of AlexNet, VGG, and NiN with GoogLeNet. How do the latter two network architectures significantly reduce the model parameter size?

4. Why do we need a large range convolution initially?

References

[1] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., & Anguelov, D. & Rabinovich, A. (2015). Going deeper with convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 1-9).

[2] Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167.

[3] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 2818-2826).

[4] Szegedy, C., Ioffe, S., Vanhoucke, V., & Alemi, A. A. (2017, February). Inception-v4, inception-resnet and the impact of residual connections on learning. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 4, p. 12).

Scan the QR Code to Discuss

6.11 Batch Normalization

Training very deep models is difficult and it can be tricky to get the models to converge (or converge within a reasonable amount of time) when training. It can be equally challenging to ensure that they do not overfit. This is one of the reasons why it took a long time for very deep networks with over 100 layers to gain popularity.

6.11.1 Training Deep Networks

Let's review some of the practical challenges when training deep networks.

1. Data preprocessing is a key aspect of effective statistical modeling. Recall our discussion when we applied deep networks to Predicting House Prices. There we standardized input data to zero mean and unit variance. Standardizing input data makes the distribution of features similar, which generally makes it easier to train effective models since parameters are a-priori at a similar scale.

2. As we train the model, the activations in intermediate layers of the network will assume rather different orders of magnitude. This can lead to issues with the convergence of the network due to scale of activations - if one layer has activation values that are 100x that of another layer, we need to adjust learning rates adaptively per layer (or even per parameter group per layer).

3. Deeper networks are fairly complex and they are more prone to overfitting. This means that regularization becomes more critical. That said dropout is nontrivial to use in convolutional layers and does not perform as well, hence we need a more appropriate type of regularization.

4. When training deep networks the last layers will converge first, at which point the layers below start converging. Unfortunately, once this happens, the weights for the last layers are no longer optimal and they need to converge again. As training progresses, this gets worse.

Batch normalization (BN), as proposed by Ioffe and Szegedy, 2015, can be used to cope with the challenges of deep model training. During training, BN continuously adjusts the intermediate output of the neural network by utilizing the mean and standard deviation of the mini-batch. In effect that causes the optimization landscape of the model to be smoother, hence allowing the model to reach a local minimum and to be trained faster. That being said, one has to be careful in oder to avoid the already troubling trends in machine learning (Lipton et al, 2018). Batch normalization has been shown (Santukar et al., 2018) to have no relation at all with internal covariate shift, as a matter in fact it has been shown that it actually causes the opposite result from what it was originally intended, pointed by Lipton et al., 2018 as well. In a nutshell, the idea in Batch Normalization is to transform the activation at a given layer from $\mathbf{x}$ to

$$\text{BN}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \hat{\mu}}{\hat{\sigma}} + \beta$$

Here $\hat{\mu}$ is the estimate of the mean and $\hat{\sigma}$ is the estimate of the variance. The result is that the activations are approximately rescaled to zero mean and unit variance. Since this may not be quite what we want, we allow for a coordinate-wise scaling coefficient γ and an offset β. Consequently the activations for intermediate layers cannot diverge any longer: we are actively rescaling it back to a given order of magnitude

via μ and σ. Consequently we can be more aggressive in picking large learning rates on the data. To address the fact that in some cases the activations actually *need* to differ from standardized data, we need to introduce scaling coefficients γ and an offset β.

We use training data to estimate mean and variance. Unfortunately, the statistics change as we train our model. To address this, we use the current minibatch also for estimating $\hat{\mu}$ and $\hat{\sigma}$. This is fairly straightforward. All we need to do is aggregate over a small set of activations, such as a minibatch of data. Hence the name *Batch* Normalization. To indicate which minibatch $\mathcal{B}$ we draw this from, we denote the quantities with $\hat{\mu}_{\mathcal{B}}$ and $\hat{\sigma}_{\mathcal{B}}$.

$$\hat{\mu}_{\mathcal{B}} \leftarrow \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} \mathbf{x} \text{ and } \hat{\sigma}_{\mathcal{B}}^2 \leftarrow \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} (\mathbf{x} - \mu_{\mathcal{B}})^2 + \epsilon$$

Note that we add a small constant $\epsilon > 0$ to the variance estimate to ensure that we never end up dividing by zero, even in cases where the empirical variance estimate might vanish by accident. The estimates $\hat{\mu}_{\mathcal{B}}$ and $\hat{\sigma}_{\mathcal{B}}$ counteract the scaling issue by using unbiased but potentially very noisy estimates of mean and variance. Normally we would consider this a problem. After all, each minibatch has different data, different labels and with it, different activations, predictions and errors. As it turns out, this is actually beneficial. This natural variation acts as *regularization* which prevents models from overfitting too badly. There is some preliminary work by Teye, Azizpour and Smith, 2018 and by Luo et al, 2018 which relate the properties of Batch Normalization (BN) to Bayesian Priors and penalties respectively. In particular, this resolves the puzzle why BN works best for moderate sizes of minibatches, i.e. of size 50-100.

Lastly, let us briefly review the original motivation of BN, namely covariate shift correction due to training. Obviously, rescaling activations to zero mean and unit variance does not entirely remove covariate shift (in fact, recent work suggests that it actually increases it). In fact, if it did, it would render deep networks entirely useless. After all, we want the activations become more meaningful for solving estimation problems. However, at least, it prevents mean and variance from diverging and thus decouples one of the more problematic aspects from training and inference.

After a lot of theory, let's look at how BN works in practice. Empirically it appears to stabilize the gradient (less exploding or vanishing values) and batch-normalized models appear to overfit less. In fact, batch-normalized models seldom even use dropout.

6.11.2 Batch Normalization Layers

The batch normalization methods for fully connected layers and convolutional layers are slightly different. This is due to the dimensionality of the data generated by convolutional layers. We discuss both cases below. Note that one of the key differences between BN and other layers is that BN operates on a a full minibatch at a time (otherwise it cannot compute the mean and variance parameters per batch).

Fully Connected Layers

Usually we apply the batch normalization layer between the affine transformation and the activation function in a fully connected layer. In the following we denote by $\mathbf{u}$ the input and by $\mathbf{x} = \mathbf{W}\mathbf{u} + \mathbf{b}$ the

output of the linear transform. This yields the following variant of the batch norm:

$$\mathbf{y} = \phi(\mathrm{BN}(\mathbf{x})) = \phi(\mathrm{BN}(\mathbf{W}\mathbf{u} + \mathbf{b}))$$

Recall that mean and variance are computed on the *same* minibatch $\mathcal{B}$ on which this transformation is applied to. Also recall that the scaling coefficient γ and the offset β are parameters that need to be learned. They ensure that the effect of batch normalization can be neutralized as needed.

Convolutional Layers

For convolutional layers, batch normalization occurs after the convolution computation and before the application of the activation function. If the convolution computation outputs multiple channels, we need to carry out batch normalization for *each* of the outputs of these channels, and each channel has an independent scale parameter and shift parameter, both of which are scalars. Assume that there are m examples in the mini-batch. On a single channel, we assume that the height and width of the convolution computation output are p and q, respectively. We need to carry out batch normalization for $m \times p \times q$ elements in this channel simultaneously. While carrying out the standardization computation for these elements, we use the same mean and variance. In other words, we use the means and variances of the $m \times p \times q$ elements in this channel rather than one per pixel.

Batch Normalization During Prediction

At prediction time we might not have the luxury of computing offsets per batch - we might be required to make one prediction at a time. Secondly, the uncertainty in μ and σ, as arising from a minibatch are highly undesirable once we've trained the model. One way to mitigate this is to compute more stable estimates on a larger set for once (e.g. via a moving average) and then fix them at prediction time. Consequently, Batch Normalization behaves differently during training and test time (just like we already saw in the case of Dropout).

6.11.3 Implementation from Scratch

Next, we will implement the batch normalization layer via the NDArray from scratch.

```
In [1]: import sys
        sys.path.insert(0, '..')

        import d2l
        from mxnet import autograd, gluon, init, nd
        from mxnet.gluon import nn

        def batch_norm(X, gamma, beta, moving_mean, moving_var, eps, momentum):
            # Use autograd to determine whether the current mode is training mode or
            # prediction mode
            if not autograd.is_training():
                # If it is the prediction mode, directly use the mean and variance
                # obtained from the incoming moving average
```

```
            X_hat = (X - moving_mean) / nd.sqrt(moving_var + eps)
        else:
            assert len(X.shape) in (2, 4)
            if len(X.shape) == 2:
                # When using a fully connected layer, calculate the mean and
                # variance on the feature dimension
                mean = X.mean(axis=0)
                var = ((X - mean) ** 2).mean(axis=0)
            else:
                # When using a two-dimensional convolutional layer, calculate the
                # mean and variance on the channel dimension (axis=1). Here we
                # need to maintain the shape of X, so that the broadcast operation
                # can be carried out later
                mean = X.mean(axis=(0, 2, 3), keepdims=True)
                var = ((X - mean) ** 2).mean(axis=(0, 2, 3), keepdims=True)
            # In training mode, the current mean and variance are used for the
            # standardization
            X_hat = (X - mean) / nd.sqrt(var + eps)
            # Update the mean and variance of the moving average
            moving_mean = momentum * moving_mean + (1.0 - momentum) * mean
            moving_var = momentum * moving_var + (1.0 - momentum) * var
        Y = gamma * X_hat + beta  # Scale and shift
        return Y, moving_mean, moving_var
```

Next, we will customize a `BatchNorm` layer. This retains the scale parameter `gamma` and the shift parameter `beta` involved in gradient finding and iteration, and it also maintains the mean and variance obtained from the moving average, so that they can be used during model prediction. The `num_features` parameter required by the `BatchNorm` instance is the number of outputs for a fully connected layer and the number of output channels for a convolutional layer. The `num_dims` parameter also required by this instance is 2 for a fully connected layer and 4 for a convolutional layer.

Besides the algorithm per se, also note the design pattern in implementing layers. Typically one defines the math in a separate function, say `batch_norm`. This is then integrated into a custom layer that mostly focuses on bookkeeping, such as moving data to the right device context, ensuring that variables are properly initialized, keeping track of the running averages for mean and variance, etc.; That way we achieve a clean separation of math and boilerplate code. Also note that for the sake of convenience we did not add automagic size inference here, hence we will need to specify the number of features throughout (the Gluon version takes care of this).

```
In [2]: class BatchNorm(nn.Block):
            def __init__(self, num_features, num_dims, **kwargs):
                super(BatchNorm, self).__init__(**kwargs)
                if num_dims == 2:
                    shape = (1, num_features)
                else:
                    shape = (1, num_features, 1, 1)
                # The scale parameter and the shift parameter involved in gradient
                # finding and iteration are initialized to 0 and 1 respectively
                self.gamma = self.params.get('gamma', shape=shape, init=init.One())
                self.beta = self.params.get('beta', shape=shape, init=init.Zero())
                # All the variables not involved in gradient finding and iteration are
                # initialized to 0 on the CPU
                self.moving_mean = nd.zeros(shape)
                self.moving_var = nd.zeros(shape)
```

```python
    def forward(self, X):
        # If X is not on the CPU, copy moving_mean and moving_var to the
        # device where X is located
        if self.moving_mean.context != X.context:
            self.moving_mean = self.moving_mean.copyto(X.context)
            self.moving_var = self.moving_var.copyto(X.context)
        # Save the updated moving_mean and moving_var
        Y, self.moving_mean, self.moving_var = batch_norm(
            X, self.gamma.data(), self.beta.data(), self.moving_mean,
            self.moving_var, eps=1e-5, momentum=0.9)
        return Y
```

6.11.4 Use a Batch Normalization LeNet

Next, we will modify the *LeNet model* in order to apply the batch normalization layer. We add the batch normalization layer after all the convolutional layers and after all fully connected layers. As discussed, we add it before the activation layer.

```python
In [3]: net = nn.Sequential()
        net.add(nn.Conv2D(6, kernel_size=5),
                BatchNorm(6, num_dims=4),
                nn.Activation('sigmoid'),
                nn.MaxPool2D(pool_size=2, strides=2),
                nn.Conv2D(16, kernel_size=5),
                BatchNorm(16, num_dims=4),
                nn.Activation('sigmoid'),
                nn.MaxPool2D(pool_size=2, strides=2),
                nn.Dense(120),
                BatchNorm(120, num_dims=2),
                nn.Activation('sigmoid'),
                nn.Dense(84),
                BatchNorm(84, num_dims=2),
                nn.Activation('sigmoid'),
                nn.Dense(10))
```

Next we train the modified model, again on Fashion-MNIST. The code is virtually identical to that in previous steps. The main difference is the considerably larger learning rate.

```python
In [4]: lr, num_epochs, batch_size, ctx = 1.0, 5, 256, d2l.try_gpu()
        net.initialize(ctx=ctx, init=init.Xavier())
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
        train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
        d2l.train_ch5(net, train_iter, test_iter, batch_size, trainer, ctx,
                      num_epochs)

training on gpu(0)
epoch 1, loss 0.6505, train acc 0.767, test acc 0.844, time 4.4 sec
epoch 2, loss 0.3915, train acc 0.858, test acc 0.863, time 3.9 sec
epoch 3, loss 0.3447, train acc 0.876, test acc 0.816, time 4.0 sec
epoch 4, loss 0.3155, train acc 0.885, test acc 0.795, time 4.0 sec
epoch 5, loss 0.2975, train acc 0.892, test acc 0.869, time 3.9 sec
```

Let's have a look at the scale parameter `gamma` and the shift parameter `beta` learned from the first batch

normalization layer.

```
In [5]: net[1].gamma.data().reshape((-1,)), net[1].beta.data().reshape((-1,))
Out[5]: (
        [2.022849  1.4888155 1.7267715 1.5379683 1.0357325 1.8945812]
        <NDArray 6 @gpu(0)>,
        [ 1.1867723   0.00592438  0.00485388  0.40120235 -0.7290907  -1.8788795 ]
        <NDArray 6 @gpu(0)>)
```

6.11.5 Concise Implementation

Compared with the `BatchNorm` class, which we just defined ourselves, the `BatchNorm` class defined by the `nn` model in Gluon is easier to use. In Gluon, we do not have to define the `num_features` and `num_dims` parameter values required in the `BatchNorm` class. Instead, these parameter values will be obtained automatically by delayed initialization. The code looks virtually identical (save for the lack of an explicit specification of the dimensionality of the features for the Batch Normalization layers).

```
In [6]: net = nn.Sequential()
        net.add(nn.Conv2D(6, kernel_size=5),
                nn.BatchNorm(),
                nn.Activation('sigmoid'),
                nn.MaxPool2D(pool_size=2, strides=2),
                nn.Conv2D(16, kernel_size=5),
                nn.BatchNorm(),
                nn.Activation('sigmoid'),
                nn.MaxPool2D(pool_size=2, strides=2),
                nn.Dense(120),
                nn.BatchNorm(),
                nn.Activation('sigmoid'),
                nn.Dense(84),
                nn.BatchNorm(),
                nn.Activation('sigmoid'),
                nn.Dense(10))
```

Use the same hyper-parameter to carry out the training. Note that as always the Gluon variant runs a lot faster since the code that is being executed is compiled C++/CUDA rather than interpreted Python.

```
In [7]: net.initialize(ctx=ctx, init=init.Xavier())
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
        d2l.train_ch5(net, train_iter, test_iter, batch_size, trainer, ctx,
                      num_epochs)

training on gpu(0)
epoch 1, loss 0.6521, train acc 0.770, test acc 0.852, time 2.1 sec
epoch 2, loss 0.3986, train acc 0.856, test acc 0.862, time 2.1 sec
epoch 3, loss 0.3488, train acc 0.873, test acc 0.878, time 2.1 sec
epoch 4, loss 0.3228, train acc 0.882, test acc 0.868, time 2.2 sec
epoch 5, loss 0.3027, train acc 0.890, test acc 0.850, time 2.1 sec
```

Summary

- During model training, batch normalization continuously adjusts the intermediate output of the neural network by utilizing the mean and standard deviation of the mini-batch, so that the values of the intermediate output in each layer throughout the neural network are more stable.
- The batch normalization methods for fully connected layers and convolutional layers are slightly different.
- Like a dropout layer, batch normalization layers have different computation results in training mode and prediction mode.
- Batch Normalization has many beneficial side effects, primarily that of regularization. On the other hand, the original motivation of reducing covariate shift seems not to be a valid explanation.

Exercises

1. Can we remove the fully connected affine transformation before the batch normalization or the bias parameter in convolution computation?

    - Find an equivalent transformation that applies prior to the fully connected layer.
    - Is this reformulation effective. Why (not)?

2. Compare the learning rates for LeNet with and without batch normalization.

    - Plot the decrease in training and test error.
    - What about the region of convergence? How large can you make the learning rate?

3. Do we need Batch Normalization in every layer? Experiment with it?

4. Can you replace Dropout by Batch Normalization? How does the behavior change?

5. Fix the coefficients `beta` and `gamma` (add the parameter `grad_req='null'` at the time of construction to avoid calculating the gradient), and observe and analyze the results.

6. Review the Gluon documentation for `BatchNorm` to see the other applications for Batch Normalization.

7. Research ideas - think of other normalization transforms that you can apply? Can you apply the probability integral transform? How about a full rank covariance estimate?

References

[1] Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167.

6.12 Residual Networks (ResNet)

As we design increasingly deeper networks it becomes imperative to understand how adding layers can increase the complexity and expressiveness of the network. Even more important is the ability to design networks where adding layers makes networks strictly more expressive rather than just different. To make some progress we need a bit of theory.

6.12.1 Function Classes

Consider $\mathcal{F}$, the class of functions that a specific network architecture (together with learning rates and other hyperparameter settings) can reach. That is, for all $f \in \mathcal{F}$ there exists some set of parameters W that can be obtained through training on a suitable dataset. Let's assume that f^* is the function that we really would like to find. If it's in $\mathcal{F}$, we're in good shape but typically we won't be quite so lucky. Instead, we will try to find some $f^*_{\mathcal{F}}$ which is our best bet within $\mathcal{F}$. For instance, we might try finding it by solving the following optimization problem:

$$f^*_{\mathcal{F}} := \underset{f}{\mathrm{argmin}}\, L(X, Y, f) \text{ subject to } f \in \mathcal{F}$$

It is only reasonable to assume that if we design a different and more powerful architecture $\mathcal{F}'$ we should arrive at a better outcome. In other words, we would expect that $f^*_{\mathcal{F}'}$ is better' than $f^*_{\mathcal{F}}$. However, if $\mathcal{F} \not\subseteq \mathcal{F}'$ there is no guarantee that this should even happen. In fact, $f^*_{\mathcal{F}'}$ might well be worse. This is a situation that we often encounter in practice - adding layers doesn't only make the network more expressive, it also changes it in sometimes not quite so predictable ways. The picture below illustrates this in slightly abstract terms.

generic function classes   nested function classes
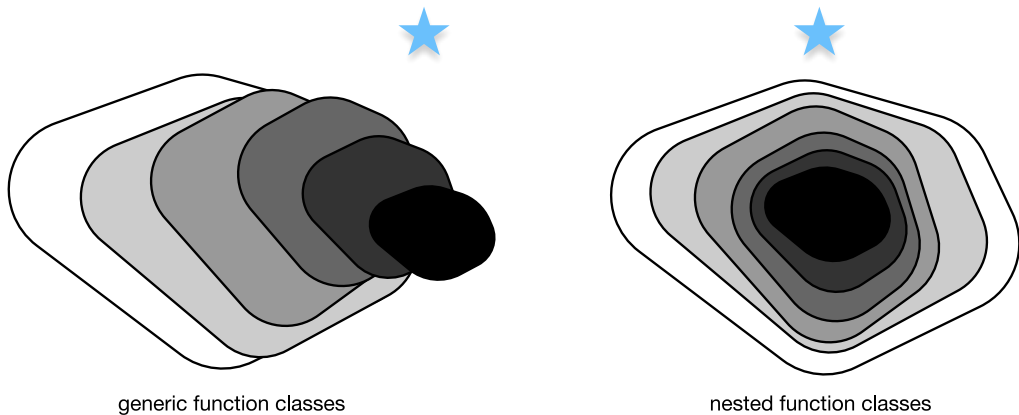
Fig. 6.15: Left: non-nested function classes. The distance may in fact increase as the complexity increases. Right: with nested function classes this does not happen.

Only if larger function classes contain the smaller ones are we guaranteed that increasing them strictly increases the expressive power of the network. This is the question that He et al, 2016 considered when working on very deep computer vision models. At the heart of ResNet is the idea that every additional layer should contain the identity function as one of its elements. This means that if we can train the newly-added layer into an identity mapping $f(\mathbf{x}) = \mathbf{x}$, the new model will be as effective as the original model. As the new model may get a better solution to fit the training data set, the added layer might make it easier to reduce training errors. Even better, the identity function rather than the null $f(\mathbf{x}) = 0$ should be the the simplest function within a layer.

These considerations are rather profound but they led to a surprisingly simple solution, a residual block. With it, He et al, 2015 won the ImageNet Visual Recognition Challenge in 2015. The design had a profound influence on how to build deep neural networks.

6.12.2 Residual Blocks

Let us focus on a local neural network, as depicted below. Denote the input by $\mathbf{x}$. We assume that the ideal mapping we want to obtain by learning is $f(\mathbf{x})$, to be used as the input to the activation function. The portion within the dotted-line box in the left image must directly fit the mapping $f(\mathbf{x})$. This can be tricky if we don't need that particular layer and we would much rather retain the input $\mathbf{x}$. The portion within the dotted-line box in the right image now only needs to parametrize the *deviation* from the identity, since we return $\mathbf{x} + f(\mathbf{x})$. In practice, the residual mapping is often easier to optimize. We only need to set $f(\mathbf{x}) = 0$. The right image in the figure below illustrates the basic Residual Block of ResNet. Similar architectures were later proposed for sequence models which we will study later.
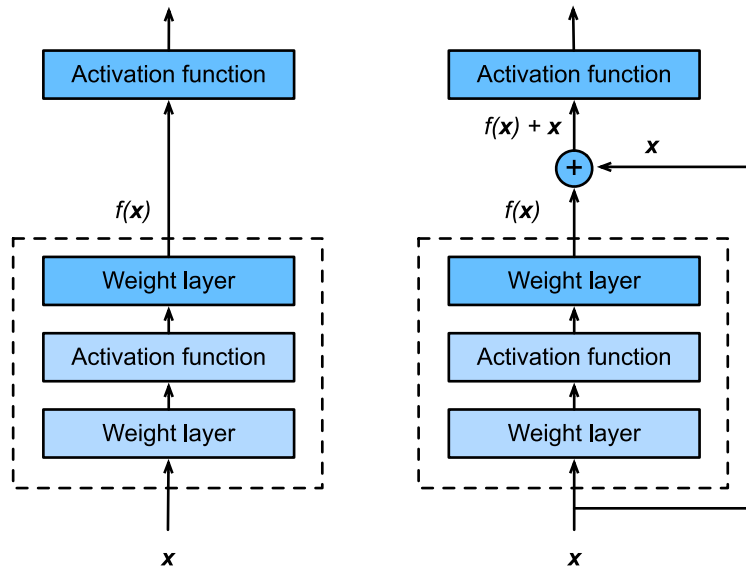
Fig. 6.16: The difference between a regular block (left) and a residual block (right). In the latter case, we can short-circuit the convolutions.

ResNet follows VGG's full 3×3 convolutional layer design. The residual block has two 3×3 convolutional layers with the same number of output channels. Each convolutional layer is followed by a batch normalization layer and a ReLU activation function. Then, we skip these two convolution operations and add the input directly before the final ReLU activation function. This kind of design requires that the output of the two convolutional layers be of the same shape as the input, so that they can be added together. If we want to change the number of channels or the the stride, we need to introduce an additional 1×1 convolutional layer to transform the input into the desired shape for the addition operation. Let's have a look at the code below.

```
In [1]: import sys
        sys.path.insert(0, '..')

        import d2l
        from mxnet import gluon, init, nd
        from mxnet.gluon import nn

        # This class has been saved in the d2l package for future use
        class Residual(nn.Block):
            def __init__(self, num_channels, use_1x1conv=False, strides=1, **kwargs):
                super(Residual, self).__init__(**kwargs)
                self.conv1 = nn.Conv2D(num_channels, kernel_size=3, padding=1,
                                       strides=strides)
                self.conv2 = nn.Conv2D(num_channels, kernel_size=3, padding=1)
                if use_1x1conv:
                    self.conv3 = nn.Conv2D(num_channels, kernel_size=1,
                                           strides=strides)
```

6. Convolutional Neural Networks

```
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm()
        self.bn2 = nn.BatchNorm()

    def forward(self, X):
        Y = nd.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        return nd.relu(Y + X)
```

This code generates two types of networks: one where we add the input to the output before applying the ReLu nonlinearity, and whenever `use_1x1conv=True`, one where we adjust channels and resolution by means of a 1×1 convolution before adding. The diagram below illustrates this:
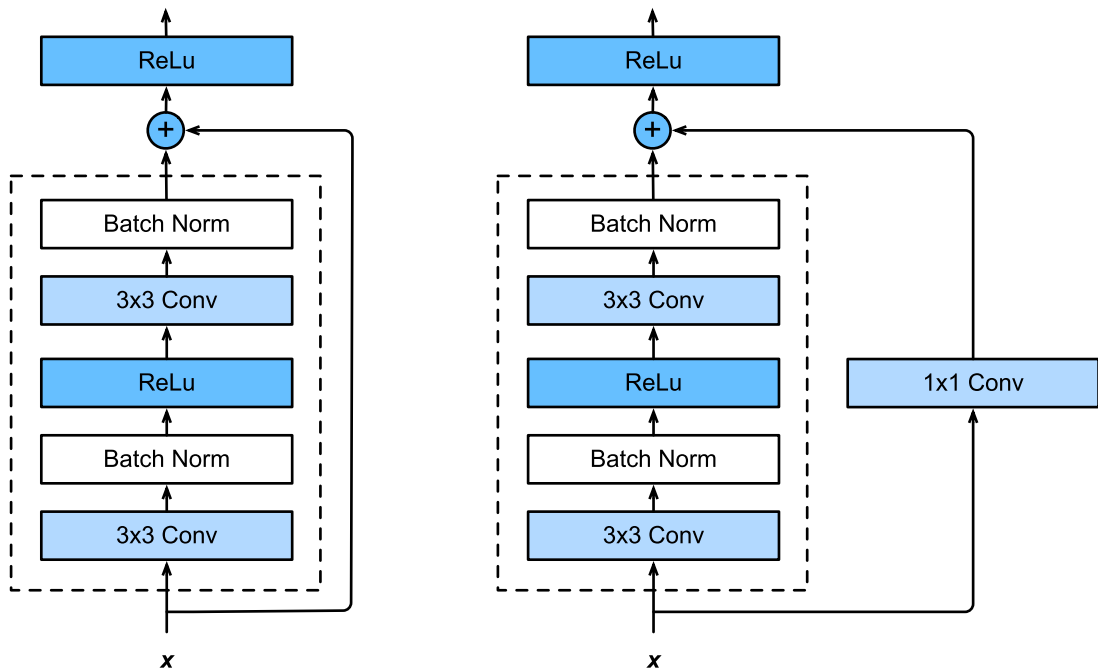
Now let us look at a situation where the input and output are of the same shape.

```
In [2]: blk = Residual(3)
        blk.initialize()
        X = nd.random.uniform(shape=(4, 3, 6, 6))
        blk(X).shape

Out[2]: (4, 3, 6, 6)
```

We also have the option to halve the output height and width while increasing the number of output channels.

```
In [3]: blk = Residual(6, use_1x1conv=True, strides=2)
```

```
        blk.initialize()
        blk(X).shape
```
`Out[3]:` (4, 6, 3, 3)

6.12.3 ResNet Model

The first two layers of ResNet are the same as those of the GoogLeNet we described before: the 7×7 convolutional layer with 64 output channels and a stride of 2 is followed by the 3×3 maximum pooling layer with a stride of 2. The difference is the batch normalization layer added after each convolutional layer in ResNet.

```
In [4]: net = nn.Sequential()
        net.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3),
                nn.BatchNorm(), nn.Activation('relu'),
                nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

GoogLeNet uses four blocks made up of Inception blocks. However, ResNet uses four modules made up of residual blocks, each of which uses several residual blocks with the same number of output channels. The number of channels in the first module is the same as the number of input channels. Since a maximum pooling layer with a stride of 2 has already been used, it is not necessary to reduce the height and width. In the first residual block for each of the subsequent modules, the number of channels is doubled compared with that of the previous module, and the height and width are halved.

Now, we implement this module. Note that special processing has been performed on the first module.

```
In [5]: def resnet_block(num_channels, num_residuals, first_block=False):
            blk = nn.Sequential()
            for i in range(num_residuals):
                if i == 0 and not first_block:
                    blk.add(Residual(num_channels, use_1x1conv=True, strides=2))
                else:
                    blk.add(Residual(num_channels))
            return blk
```

Then, we add all the residual blocks to ResNet. Here, two residual blocks are used for each module.

```
In [6]: net.add(resnet_block(64, 2, first_block=True),
                resnet_block(128, 2),
                resnet_block(256, 2),
                resnet_block(512, 2))
```

Finally, just like GoogLeNet, we add a global average pooling layer, followed by the fully connected layer output.

```
In [7]: net.add(nn.GlobalAvgPool2D(), nn.Dense(10))
```

There are 4 convolutional layers in each module (excluding the 1×1 convolutional layer). Together with the first convolutional layer and the final fully connected layer, there are 18 layers in total. Therefore, this model is commonly known as ResNet-18. By configuring different numbers of channels and residual blocks in the module, we can create different ResNet models, such as the deeper 152-layer ResNet-152. Although the main architecture of ResNet is similar to that of GoogLeNet, ResNet's structure is simpler

and easier to modify. All these factors have resulted in the rapid and widespread use of ResNet. Below is a diagram of the full ResNet-18.
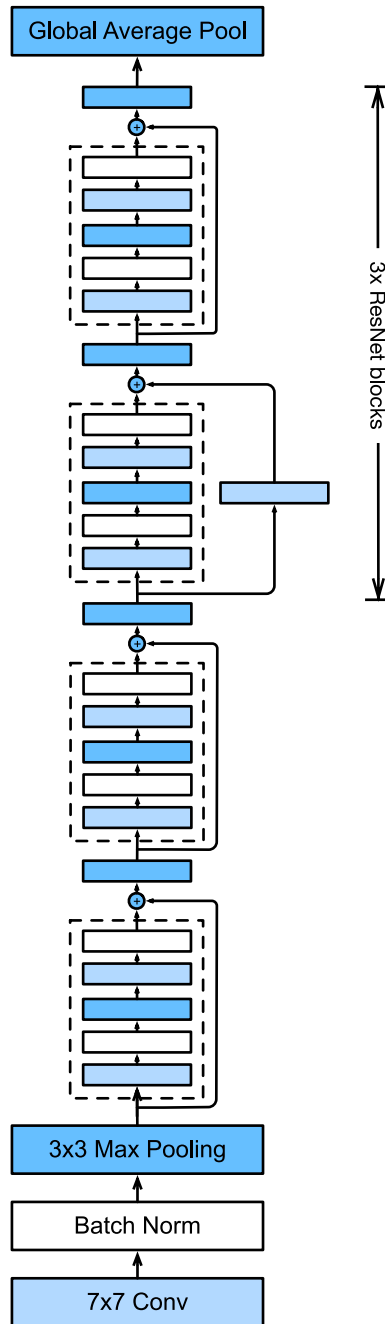
Fig. 6.17: ResNet 18

Before training ResNet, let us observe how the input shape changes between different modules in ResNet. As in all previous architectures, the resolution decreases while the number of channels increases up until the point where a global average pooling layer aggregates all features.

```
In [8]: X = nd.random.uniform(shape=(1, 1, 224, 224))
        net.initialize()
        for layer in net:
            X = layer(X)
            print(layer.name, 'output shape:\t', X.shape)

conv5 output shape:        (1, 64, 112, 112)
batchnorm4 output shape:         (1, 64, 112, 112)
relu0 output shape:        (1, 64, 112, 112)
pool0 output shape:        (1, 64, 56, 56)
sequential1 output shape:        (1, 64, 56, 56)
sequential2 output shape:        (1, 128, 28, 28)
sequential3 output shape:        (1, 256, 14, 14)
sequential4 output shape:        (1, 512, 7, 7)
pool1 output shape:        (1, 512, 1, 1)
dense0 output shape:       (1, 10)
```

6.12.4 Data Acquisition and Training

We train ResNet on the Fashion-MNIST data set, just like before. The only thing that has changed is the learning rate that decreased again, due to the more complex architecture.

```
In [9]: lr, num_epochs, batch_size, ctx = 0.05, 5, 256, d2l.try_gpu()
        net.initialize(force_reinit=True, ctx=ctx, init=init.Xavier())
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
        train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
        d2l.train_ch5(net, train_iter, test_iter, batch_size, trainer, ctx,
                      num_epochs)

training on gpu(0)
epoch 1, loss 0.4846, train acc 0.830, test acc 0.897, time 61.3 sec
epoch 2, loss 0.2485, train acc 0.908, test acc 0.905, time 57.0 sec
epoch 3, loss 0.1905, train acc 0.930, test acc 0.858, time 57.0 sec
epoch 4, loss 0.1434, train acc 0.948, test acc 0.917, time 57.1 sec
epoch 5, loss 0.1067, train acc 0.962, test acc 0.919, time 57.0 sec
```

Summary

- Residual blocks allow for a parametrization relative to the identity function $f(\mathbf{x}) = \mathbf{x}$.
- Adding residual blocks increases the function complexity in a well-defined manner.
- We can train an effective deep neural network by having residual blocks pass through cross-layer data channels.
- ResNet had a major influence on the design of subsequent deep neural networks, both for convolutional and sequential nature.

Exercises

1. Refer to Table 1 in the ResNet paper to implement different variants.

2. For deeper networks, ResNet introduces a bottleneck architecture to reduce model complexity. Try to implement it.

3. In subsequent versions of ResNet, the author changed the convolution, batch normalization, and activation architecture to the batch normalization, activation, and convolution architecture. Make this improvement yourself. See Figure 1 in He et al., 2016 for details.

4. Prove that if $\mathbf{x}$ is generated by a ReLu, the ResNet block does indeed include the identity function.

5. Why can't we just increase the complexity of functions without bound, even if the function classes are nested?

References

[1] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

[2] He, K., Zhang, X., Ren, S., & Sun, J. (2016, October). Identity mappings in deep residual networks. In European Conference on Computer Vision (pp. 630-645). Springer, Cham.

Scan the QR Code to Discuss

6.13 Densely Connected Networks (DenseNet)

ResNet significantly changed the view of how to parametrize the functions in deep networks. DenseNet is to some extent the logical extension of this. To understand how to arrive at it, let's take a small detour to theory. Recall the Taylor expansion for functions. For scalars it can be written as

$$f(x) = f(0) + f'(x)x + \frac{1}{2}f''(x)x^2 + \frac{1}{6}f'''(x)x^3 + o(x^3)$$

6.13.1 Function Decomposition

The key point is that it decomposes the function into increasingly higher order terms. In a similar vein, ResNet decomposes functions into

$$f(\mathbf{x}) = \mathbf{x} + g(\mathbf{x})$$

That is, ResNet decomposes f into a simple linear term and a more complex nonlinear one. What if we want to go beyond two terms? A solution was proposed by Huang et al, 2016 in the form of DenseNet, an architecture that reported record performance on the ImageNet dataset.
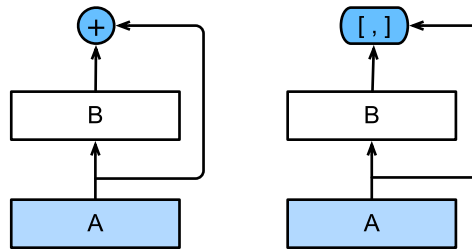
Fig. 6.18: The main difference between ResNet (left) and DenseNet (right) in cross-layer connections: use of addition and use of concatenation.

The key difference between ResNet and DenseNet is that in the latter case outputs are *concatenated* rather than added. As a result we perform a mapping from $\mathbf{x}$ to its values after applying an increasingly complex sequence of functions.

$$\mathbf{x} \to [\mathbf{x}, f_1(\mathbf{x}), f_2(\mathbf{x}, f_1(\mathbf{x})), f_3(\mathbf{x}, f_1(\mathbf{x}), f_2(\mathbf{x}, f_1(\mathbf{x})), \ldots]$$

In the end, all these functions are combined in an MLP to reduce the number of features again. In terms of implementation this is quite simple - rather than adding terms, we concatenate them. The name DenseNet arises from the fact that the dependency graph between variables becomes quite dense. The last layer of such a chain is densely connected to all previous layers. The main components that compose a DenseNet are dense blocks and transition layers. The former defines how the inputs and outputs are concatenated, while the latter controls the number of channels so that it is not too large.
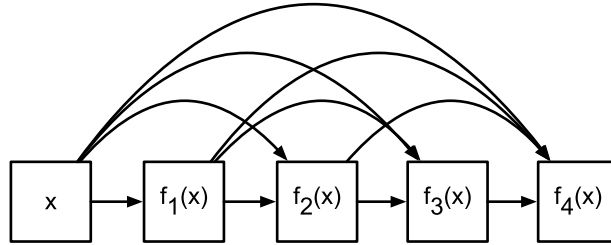
Fig. 6.19: Dense connections in DenseNet

6.13.2 Dense Blocks

DenseNet uses the modified batch normalization, activation, and convolution architecture of ResNet (see the exercise in the *previous section*). First, we implement this architecture in the `conv_block` function.

```
In [1]: import sys
        sys.path.insert(0, '..')

        import d2l
        from mxnet import gluon, init, nd
        from mxnet.gluon import nn

        def conv_block(num_channels):
            blk = nn.Sequential()
            blk.add(nn.BatchNorm(),
                    nn.Activation('relu'),
                    nn.Conv2D(num_channels, kernel_size=3, padding=1))
            return blk
```

A dense block consists of multiple `conv_block` units, each using the same number of output channels. In the forward computation, however, we concatenate the input and output of each block on the channel dimension.

```
In [2]: class DenseBlock(nn.Block):
            def __init__(self, num_convs, num_channels, **kwargs):
                super(DenseBlock, self).__init__(**kwargs)
                self.net = nn.Sequential()
                for _ in range(num_convs):
                    self.net.add(conv_block(num_channels))

            def forward(self, X):
                for blk in self.net:
                    Y = blk(X)
                    # Concatenate the input and output of each block on the channel
                    # dimension
                    X = nd.concat(X, Y, dim=1)
                return X
```

In the following example, we define a convolution block with two blocks of 10 output channels. When using an input with 3 channels, we will get an output with the $3 + 2 \times 10 = 23$ channels. The number of

convolution block channels controls the increase in the number of output channels relative to the number of input channels. This is also referred to as the growth rate.

```
In [3]: blk = DenseBlock(2, 10)
        blk.initialize()
        X = nd.random.uniform(shape=(4, 3, 8, 8))
        Y = blk(X)
        Y.shape

Out[3]: (4, 23, 8, 8)
```

6.13.3 Transition Layers

Since each dense block will increase the number of channels, adding too many of them will lead to an excessively complex model. A transition layer is used to control the complexity of the model. It reduces the number of channels by using the 1×1 convolutional layer and halves the height and width of the average pooling layer with a stride of 2, further reducing the complexity of the model.

```
In [4]: def transition_block(num_channels):
            blk = nn.Sequential()
            blk.add(nn.BatchNorm(), nn.Activation('relu'),
                    nn.Conv2D(num_channels, kernel_size=1),
                    nn.AvgPool2D(pool_size=2, strides=2))
            return blk
```

Apply a transition layer with 10 channels to the output of the dense block in the previous example. This reduces the number of output channels to 10, and halves the height and width.

```
In [5]: blk = transition_block(10)
        blk.initialize()
        blk(Y).shape

Out[5]: (4, 10, 4, 4)
```

6.13.4 DenseNet Model

Next, we will construct a DenseNet model. DenseNet first uses the same single convolutional layer and maximum pooling layer as ResNet.

```
In [6]: net = nn.Sequential()
        net.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3),
                nn.BatchNorm(), nn.Activation('relu'),
                nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

Then, similar to the four residual blocks that ResNet uses, DenseNet uses four dense blocks. Similar to ResNet, we can set the number of convolutional layers used in each dense block. Here, we set it to 4, consistent with the ResNet-18 in the previous section. Furthermore, we set the number of channels (i.e. growth rate) for the convolutional layers in the dense block to 32, so 128 channels will be added to each dense block.

In ResNet, the height and width are reduced between each module by a residual block with a stride of 2. Here, we use the transition layer to halve the height and width and halve the number of channels.

```
In [7]: # Num_channels: the current number of channels
        num_channels, growth_rate = 64, 32
        num_convs_in_dense_blocks = [4, 4, 4, 4]

        for i, num_convs in enumerate(num_convs_in_dense_blocks):
            net.add(DenseBlock(num_convs, growth_rate))
            # This is the number of output channels in the previous dense block
            num_channels += num_convs * growth_rate
            # A transition layer that haves the number of channels is added between
            # the dense blocks
            if i != len(num_convs_in_dense_blocks) - 1:
                net.add(transition_block(num_channels // 2))
```

Similar to ResNet, a global pooling layer and fully connected layer are connected at the end to produce the output.

```
In [8]: net.add(nn.BatchNorm(),
                nn.Activation('relu'),
                nn.GlobalAvgPool2D(),
                nn.Dense(10))
```

6.13.5 Data Acquisition and Training

Since we are using a deeper network here, in this section, we will reduce the input height and width from 224 to 96 to simplify the computation.

```
In [9]: lr, num_epochs, batch_size, ctx = 0.1, 5, 256, d2l.try_gpu()
        net.initialize(ctx=ctx, init=init.Xavier())
        trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
        train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size, resize=96)
        d2l.train_ch5(net, train_iter, test_iter, batch_size, trainer, ctx,
                      num_epochs)

training on gpu(0)
epoch 1, loss 0.5414, train acc 0.806, test acc 0.804, time 64.5 sec
epoch 2, loss 0.3151, train acc 0.885, test acc 0.890, time 59.6 sec
epoch 3, loss 0.2643, train acc 0.904, test acc 0.887, time 59.6 sec
epoch 4, loss 0.2317, train acc 0.915, test acc 0.906, time 59.6 sec
epoch 5, loss 0.2095, train acc 0.924, test acc 0.882, time 59.6 sec
```

Summary

- In terms of cross-layer connections, unlike ResNet, where inputs and outputs are added together, DenseNet concatenates inputs and outputs on the channel dimension.

- The main units that compose DenseNet are dense blocks and transition layers.

- We need to keep the dimensionality under control when composing the network by adding transition layers that shrink the number of channels again.

Exercises

1. Why do we use average pooling rather than max-pooling in the transition layer?

2. One of the advantages mentioned in the DenseNet paper is that its model parameters are smaller than those of ResNet. Why is this the case?

3. One problem for which DenseNet has been criticized is its high memory consumption.

   - Is this really the case? Try to change the input shape to 224×224 to see the actual (GPU) memory consumption.

   - Can you think of an alternative means of reducing the memory consumption? How would you need to change the framework?

4. Implement the various DenseNet versions presented in Table 1 of the DenseNet paper.

5. Why do we not need to concatenate terms if we are just interested in $\mathbf{x}$ and $f(\mathbf{x})$ for ResNet? Why do we need this for more than two layers in DenseNet?

6. Design a DenseNet for fully connected networks and apply it to the Housing Price prediction task.

References

[1] Huang, G., Liu, Z., Weinberger, K. Q., & van der Maaten, L. (2017). Densely connected convolutional networks. In Proceedings of the IEEE conference on computer vision and pattern recognition (Vol. 1, No. 2).

Scan the QR Code to Discuss

7

Recurrent Neural Networks

So far we encountered two types of data: generic vectors and images. For the latter we designed specialized layers to take advantage of the regularity properties in them. In other words, if we were to permute the pixels in an image, it would be much more difficult to reason about its content of something that would look much like the background of a test pattern in the times of Analog TV.

Most importantly, so far we tacitly assumed that our data is generated iid, i.e. independently and identically distributed, all drawn from some distribution. Unfortunately, this isn't true for most data. For instance, the words in this paragraph are written in sequence, and it would be quite difficult to decipher its meaning if they were permuted randomly. Likewise, image frames in a video, the audio signal in a conversation, or the browsing behavior on a website, all follow sequential order. It is thus only reasonable to assume that specialized models for such data will do better at describing it and at solving estimation problems.

Another issue arises from the fact that we might not only receive a sequence as an input but rather might be expected to continue the sequence. For instance, the task could be to continue the series 2, 4, 6, 8, 10, This is quite common in time series analysis, to predict the stock market, the fever curve of a patient or the acceleration needed for a race car. Again we want to have models that can handle such data.

In short, while convolutional neural networks can efficiently process spatial information, recurrent neural networks are designed to better handle sequential information. These networks introduces state variables to store past information and, together with the current input, determine the current output.

Many of the examples for using recurrent networks are based on text data. Hence, we will emphasize language models in this chapter. After a more formal review of sequence data we discuss basic concepts of a language model and use this discussion as the inspiration for the design of recurrent neural networks.

Next, we describe the gradient calculation method in recurrent neural networks to explore problems that may be encountered in recurrent neural network training. For some of these problems, we can use gated recurrent neural networks, such as LSTMs and GRUs, described later in this chapter.

7.1 Sequence Models

Imagine that you're watching movies on Netflix. As a good Netflix user you decide to rate each of the movies religiously. After all, a good movie is a good movie, and you want to watch more of them, right? As it turns out, things are not quite so simple. People's opinions on movies can change quite significantly over time. In fact, psychologists even have names for some of the effects:

- There's anchoring, based on someone else's opinion. For instance after the Oscar awards, ratings for the corresponding movie go up, even though it's still the same movie. This effect persists for a few months until the award is forgotten. Wu et al., 2017 showed that the effect lifts rating by over half a point.

- There's the Hedonic adaptation, where humans quickly adapt to accept an improved (or a bad) situation as the new normal. For instance, after watching many good movies, the expectations that the next movie be equally good or better are high, and even an average movie might be considered a bad movie after many great ones.

- There's seasonality. Very few viewers like to watch a Santa Claus movie in August.

- In some cases movies become unpopular due to the misbehaviors of directors or actors in the production.

- Some movies become cult movies, because they were almost comically bad. *Plan 9 from Outer Space* and *Troll 2* achieved a high degree of notoriety for this reason.

In short, ratings are anything but stationary. Using temporal dynamics helped Yehuda Koren, 2009 to recommend movies more accurately. But it isn't just about movies.

- Many users have highly particular behavior when it comes to the time when they open apps. For instance, social media apps are much more popular after school with students. Stock market trading apps are more commonly used when the markets are open.

- It is much harder to predict tomorrow's stock prices than to fill in the blanks for a stock price we missed yesterday, even though both are just a matter of estimating one number. After all, hindsight is so much easier than foresight. In statistics the former is called *prediction* whereas the latter is called *filtering*.

- Music, speech, text, movies, steps, etc. are all sequential in nature. If we were to permute them they would make little sense. The headline *dog bites man* is much less surprising than *man bites dog*, even though the words are identical.

- Earthquakes are strongly correlated, i.e. after a massive earthquake there are very likely several smaller aftershocks, much more so than without the strong quake. In fact, earthquakes are spa-

tiotemporally correlated, i.e. the aftershocks typically occur within a short time span and in close proximity.

- Humans interact with each other in a sequential nature, as can be seen in Twitter fights, dance patterns and debates.

7.1.1 Statistical Tools

In short, we need statistical tools and new deep networks architectures to deal with sequence data. To keep things simple, we use the stock price as an example.
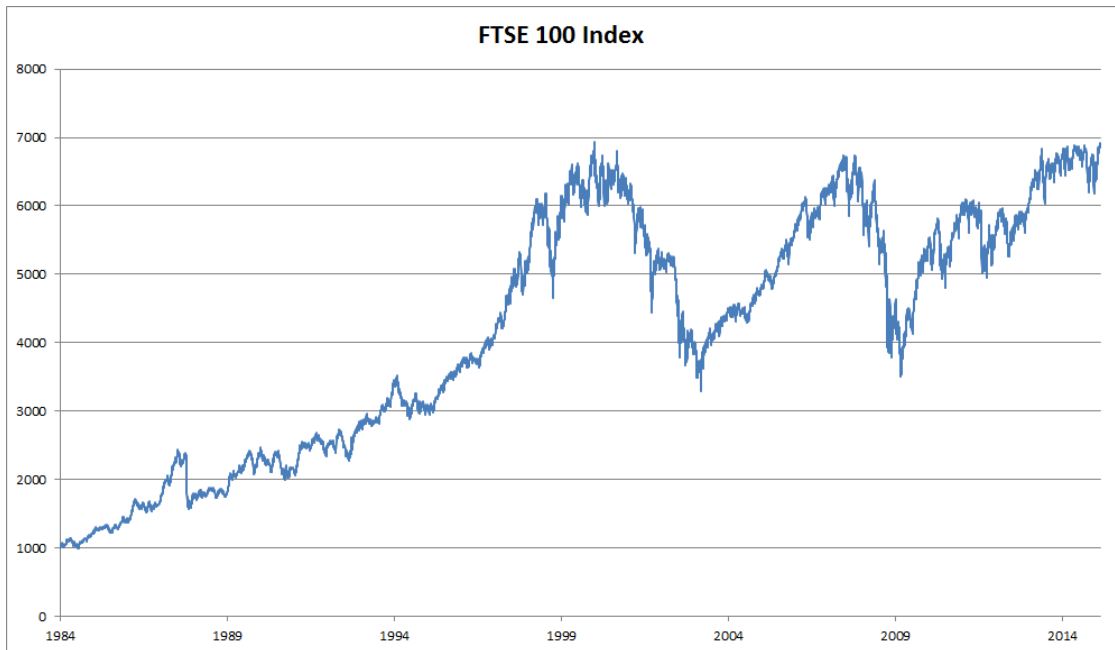
Fig. 7.1: FTSE 100 index over 30 years

Let's denote the prices by $x_t \geq 0$, i.e. at time $t \in \mathbb{N}$ we observe some price x_t. For a trader to do well in the stock market on day t he should want to predict x_t via

$$x_t \sim p(x_t | x_{t-1}, \ldots x_1).$$

Autoregressive Models

In order to achieve this, our trader could use a regressor such as the one we trained in the section on regression. There's just a major problem - the number of inputs, $x_{t-1}, \ldots x_1$ varies, depending on t. That

is, the number increases with the amount of data that we encounter, and we will need an approximation to make this computationally tractable. Much of what follows in this chapter will revolve around how to estimate $p(x_t|x_{t-1}, \ldots x_1)$ efficiently. In a nutshell it boils down to two strategies:

1. Assume that the potentially rather long sequence $x_{t-1}, \ldots x_1$ isn't really necessary. In this case we might content ourselves with some timespan τ and only use $x_{t-1}, \ldots x_{t-\tau}$ observations. The immediate benefit is that now the number of arguments is always the same, at least for $t > \tau$. This allows us to train a deep network as indicated above. Such models will be called *autoregressive* models, as they quite literally perform regression on themselves.

2. Another strategy is to try and keep some summary h_t of the past observations around and update that in addition to the actual prediction. This leads to models that estimate $x_t|x_{t-1}, h_{t-1}$ and moreover updates of the form $h_t = g(h_t, x_t)$. Since h_t is never observed, these models are also called *latent autoregressive models*. LSTMs and GRUs are examples of this.

Both cases raise the obvious question how to generate training data. One typically uses historical observations to predict the next observation given the ones up to right now. Obviously we do not expect time to stand still. However, a common assumption is that while the specific values of x_t might change, at least the dynamics of the time series itself won't. This is reasonable, since novel dynamics are just that, novel and thus not predictable using data we have so far. Statisticians call dynamics that don't change *stationary*. Regardless of what we do, we will thus get an estimate of the entire time series via

$$p(x_1, \ldots x_T) = \prod_{t=1}^{T} p(x_t|x_{t-1}, \ldots x_1).$$

Note that the above considerations still hold if we deal with discrete objects, such as words, rather than numbers. The only difference is that in such a situation we need to use a classifier rather than a regressor to estimate $p(x_t|x_{t-1}, \ldots x_1)$.

Markov Model

Recall the approximation that in an autoregressive model we use only $(x_{t-1}, \ldots x_{t-\tau})$ instead of $(x_{t-1}, \ldots x_1)$ to estimate x_t. Whenever this approximation is accurate we say that the sequence satisfies a Markov condition. In particular, if $\tau = 1$, we have a *first order* Markov model and $p(x)$ is given by

$$p(x_1, \ldots x_T) = \prod_{t=1}^{T} p(x_t|x_{t-1}).$$

Such models are particularly nice whenever x_t assumes only discrete values, since in this case dynamic programming can be used to compute values along the chain exactly. For instance, we can compute $x_{t+1}|x_{t-1}$ efficiently using the fact that we only need to take into account a very short history of past observations.

$$p(x_{t+1}|x_{t-1}) = \sum_{x_t} p(x_{t+1}|x_t)p(x_t|x_{t-1})$$

Going into details of dynamic programming is beyond the scope of this section. Control and reinforcement learning algorithms use such tools extensively.

Causality

In principle, there's nothing wrong with unfolding $p(x_1, \ldots x_T)$ in reverse order. After all, by conditioning we can always write it via

$$p(x_1, \ldots x_T) = \prod_{t=T}^{1} p(x_t | x_{t+1}, \ldots x_T).$$

In fact, if we have a Markov model we can obtain a reverse conditional probability distribution, too. In many cases, however, there exists a natural direction for the data, namely going forward in time. It is clear that future events cannot influence the past. Hence, if we change x_t, we may be able to influence what happens for x_{t+1} going forward but not the converse. That is, if we change x_t, the distribution over past events will not change. Consequently, it ought to be easier to explain $x_{t+1} | x_t$ rather than $x_t | x_{t+1}$. For instance, Hoyer et al., 2008 show that in some cases we can find $x_{t+1} = f(x_t) + \epsilon$ for some additive noise, whereas the converse is not true. This is great news, since it is typically the forward direction that we're interested in estimating. For more on this topic see e.g. the book by Peters, Janzing and Schölkopf, 2015. We are barely scratching the surface of it.

7.1.2 Toy Example

After so much theory, let's try this out in practice. Since much of the modeling is identical to when we built regression estimators in Gluon, we will not delve into much detail regarding the choice of architecture besides the fact that we will use several layers of a fully connected network. Let's begin by generating some data. To keep things simple we generate our time series' by using a sine function with some additive noise.

```
In [1]: %matplotlib inline
        from IPython import display
        from matplotlib import pyplot as plt
        from mxnet import autograd, nd, gluon, init
        display.set_matplotlib_formats('svg')

        embedding = 4  # Embedding dimension for autoregressive model
        T = 1000  # Generate a total of 1000 points
        time = nd.arange(0,T)
        x = nd.sin(0.01 * time) + 0.2 * nd.random.normal(shape=(T))

        plt.plot(time.asnumpy(), x.asnumpy());
```
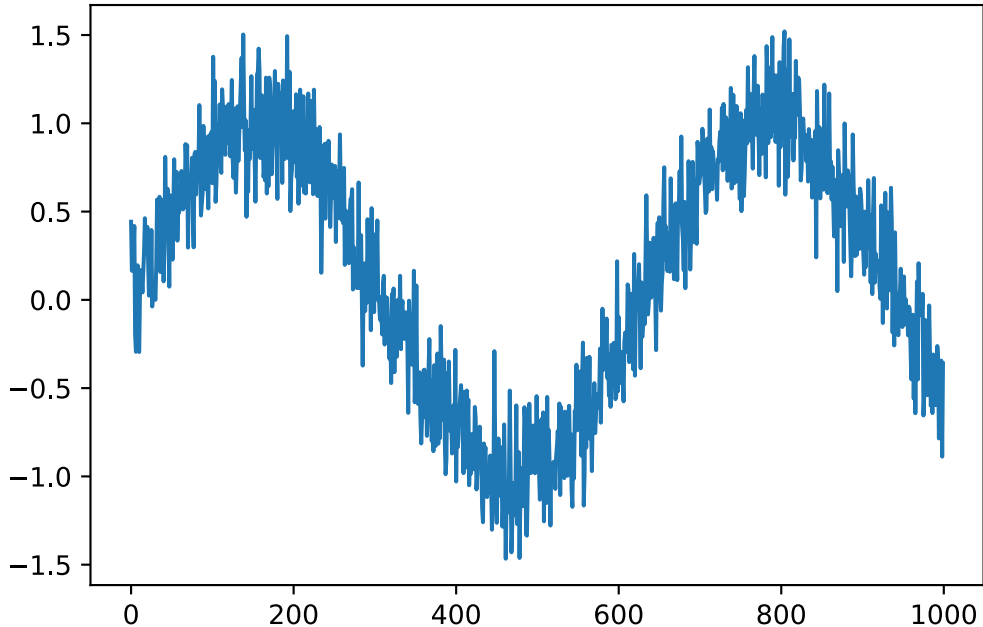
Next we need to turn this time series' into data the network can train on. Based on the embedding dimension τ we map the data into pairs $y_t = x_t$ and $\mathbf{z}_t = (x_{t-1}, \ldots x_{t-\tau})$. The astute reader might have noticed that this gives us τ fewer datapoints, since we don't have sufficient history for the first τ of them. A simple fix, in particular if the time series is long is to discard those few terms. Alternatively we could pad the time series with zeros. The code below is essentially identical to the training code in previous sections.

```
In [2]: features = nd.zeros((T-embedding, embedding))
        for i in range(embedding):
            features[:,i] = x[i:T-embedding+i]
        labels = x[embedding:]

        ntrain = 600
        train_data = gluon.data.ArrayDataset(features[:ntrain,:], labels[:ntrain])
        test_data  = gluon.data.ArrayDataset(features[ntrain:,:], labels[ntrain:])

        # Vanilla MLP architecture
        def get_net():
            net = gluon.nn.Sequential()
            net.add(gluon.nn.Dense(10, activation='relu'))
            net.add(gluon.nn.Dense(10, activation='relu'))
            net.add(gluon.nn.Dense(1))
            net.initialize(init.Xavier())
            return net

        # Least mean squares loss
        loss = gluon.loss.L2Loss()
```

We kept the architecture fairly simple. A few layers of a fully connected network, ReLu activation and ℓ_2 loss. Now we are ready to train.

```
In [3]: # Simple optimizer using adam, random shuffle and minibatch size 16
        def train_net(net, data, loss, epochs, learningrate):
            batch_size = 16
            trainer = gluon.Trainer(net.collect_params(), 'adam',
                                    {'learning_rate': learningrate})
            data_iter = gluon.data.DataLoader(data, batch_size, shuffle=True)
            for epoch in range(1, epochs + 1):
                for X, y in data_iter:
                    with autograd.record():
                        l = loss(net(X), y)
                    l.backward()
                    trainer.step(batch_size)
                l = loss(net(data[:][0]), nd.array(data[:][1]))
                print('epoch %d, loss: %f' % (epoch, l.mean().asnumpy()))
            return net

        net = get_net()
        net = train_net(net, train_data, loss, 10, 0.01)

        l = loss(net(test_data[:][0]), nd.array(test_data[:][1]))
        print('test loss: %f' % l.mean().asnumpy())

epoch 1, loss: 0.034594
epoch 2, loss: 0.029675
epoch 3, loss: 0.029356
epoch 4, loss: 0.028222
epoch 5, loss: 0.028895
epoch 6, loss: 0.027512
epoch 7, loss: 0.027489
epoch 8, loss: 0.027454
epoch 9, loss: 0.026997
epoch 10, loss: 0.027068
test loss: 0.028095
```

The both training and test loss are small and we would expect our model to work well. Let's see what this means in practice. The first thing to check is how well the model is able to predict what happens in the next timestep.

```
In [4]: estimates = net(features)
        plt.plot(time.asnumpy(), x.asnumpy(), label='data');
        plt.plot(time[embedding:].asnumpy(), estimates.asnumpy(), label='estimate');
        plt.legend();
```
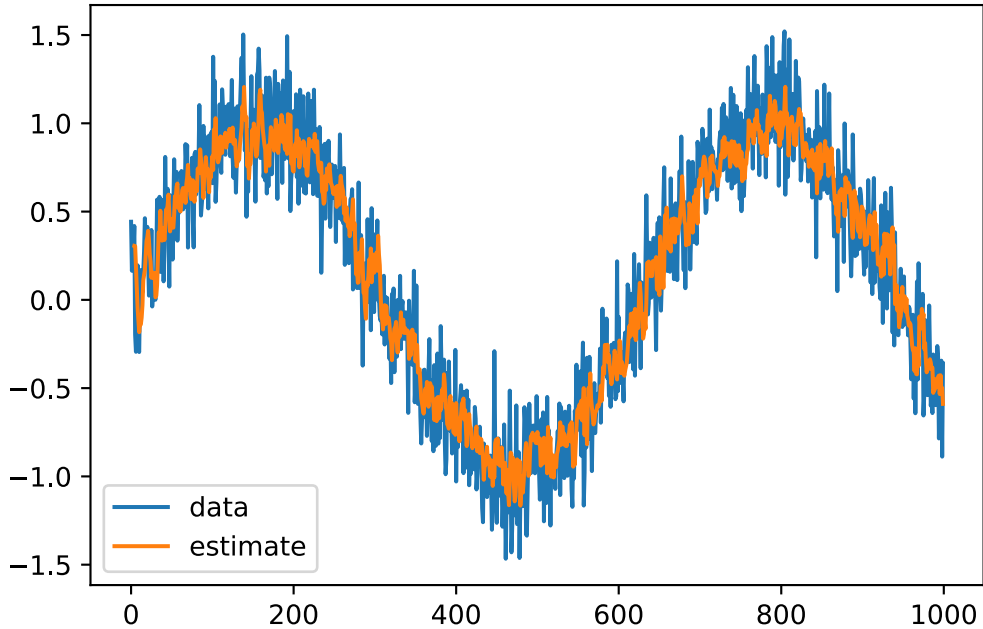
7.1.3 Predictions

This looks nice, just as we expected it. Even beyond 600 observations the estimates still look rather trustworthy. There's just one little problem to this - if we observe data only until time step 600, we cannot hope to receive the ground truth for all future predictions. Instead, we need to work our way forward one step at a time:

$$x_{601} = f(x_{600}, \ldots, x_{597})$$
$$x_{602} = f(x_{601}, \ldots, x_{598})$$
$$x_{603} = f(x_{602}, \ldots, x_{599})$$

In other words, very quickly will we have to use our own predictions to make future predictions. Let's see how well this goes.

```
In [5]: predictions = nd.zeros_like(estimates)
        predictions[:(ntrain-embedding)] = estimates[:(ntrain-embedding)]
        for i in range(ntrain-embedding, T-embedding):
            predictions[i] = net(
                predictions[(i-embedding):i].reshape(1,-1)).reshape(1)

        plt.plot(time.asnumpy(), x.asnumpy(), label='data');
        plt.plot(time[embedding:].asnumpy(), estimates.asnumpy(), label='estimate');
        plt.plot(time[embedding:].asnumpy(), predictions.asnumpy(),
                 label='multistep');
```

```
plt.legend();
```
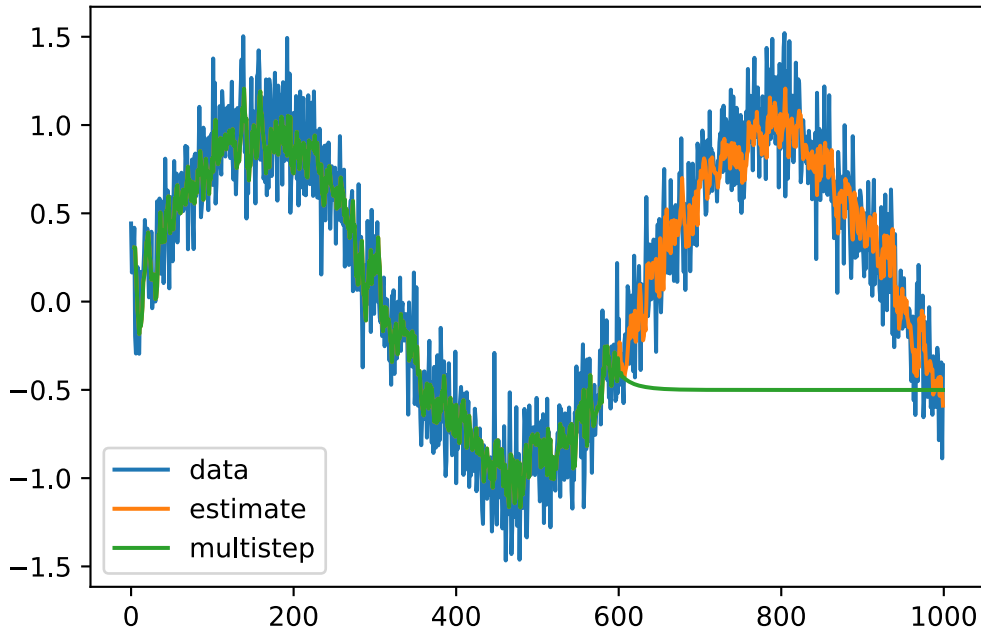

As the above example shows, this is a spectacular failure. The estimates decay to 0 pretty quickly after a few prediction steps. Why did the algorithm work so poorly? This is ultimately due to the fact that errors build up. Let's say that after step 1 we have some error $\epsilon_1 = \bar{\epsilon}$. Now the *input* for step 2 is perturbed by ϵ_1, hence we suffer some error in the order of $\epsilon_2 = \bar{\epsilon} + L\epsilon_1$, and so on. The error can diverge rather rapidly from the true observations. This is a common phenomenon - for instance weather forecasts for the next 24 hours tend to be pretty accurate but beyond that their accuracy declines rapidly. We will discuss methods for improvig this throughout this chapter and beyond.

Let's verify this observation by computing the k-step predictions on the entire sequence.

```
In [6]: k = 33   # Look up to k - embedding steps ahead

        features = nd.zeros((T-k, k))
        for i in range(embedding):
            features[:,i] = x[i:T-k+i]

        for i in range(embedding, k):
            features[:,i] = net(features[:,(i-embedding):i]).reshape((-1))

        for i in (4, 8, 16, 32):
            plt.plot(time[i:T-k+i].asnumpy(), features[:,i].asnumpy(),
                     label=('step ' + str(i)))
        plt.legend();
```
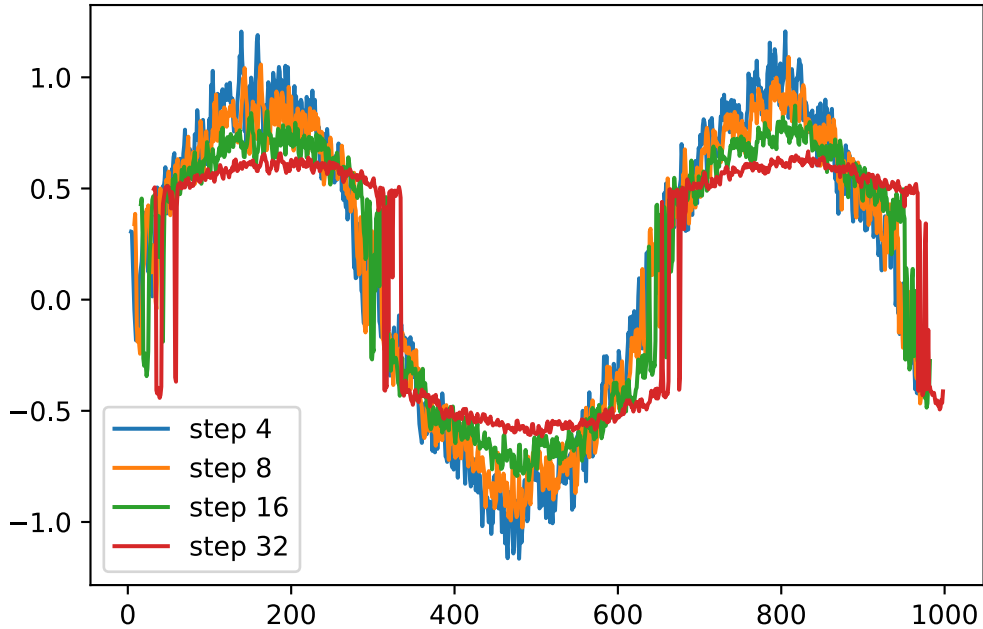
This clearly illustrates how the quality of the estimates changes as we try to predict further into the future. While the 8-step predictions are still pretty good, anything beyond that is pretty useless.

Summary

- Sequence models require specialized statistical tools for estimation. Two popular choices are autoregressive models and latent-variable autoregressive models.

- As we predict further in time, the errors accumulate and the quality of the estimates degrades, often dramatically.

- There's quite a difference in difficulty between filling in the blanks in a sequence (smoothing) and forecasting. Consequently, if you have a time series, always respect the temporal order of the data when training, i.e. never train on future data.

- For causal models (e.g. time going forward), estimating the forward direction is typically a lot easier than the reverse direction, i.e. we can get by with simpler networks.

Exercises

1. Improve the above model.
    - Incorporate more than the past 4 observations? How many do you really need?

- How many would you need if there were no noise? Hint - you can write $\sin$ and $\cos$ as a differential equation.

- Can you incorporate older features while keeping the total number of features constant? Does this improve accuracy? Why?

- Change the architecture and see what happens.

2. An investor wants to find a good security to buy. She looks at past returns to decide which one is likely to do well. What could possibly go wrong with this strategy?

3. Does causality also apply to text? To which extent?

4. Give an example for when a latent variable autoregressive model might be needed to capture the dynamic of the data.

Scan the QR Code to Discuss

7.2 Language Models

Text is an important example of sequence data. In fact, we will use natural language models as the basis for many of the examples in this chapter. Given that, it's worth while discussing some things in a bit more detail. In the following we will view words (or sequences of characters) as a time series of discrete observations. Assuming the words in a text of length T are in turn $w_1, w_2, \ldots, w_T$, then, in the discrete time series, $w_t(1 \le t \le T)$ can be considered as the output or label of time step t. Given such a sequence, the goal of a language model is to estimate the probability

$$p(w_1, w_2, \ldots, w_T).$$

Language models are incredibly useful. For instance, an ideal language model would be able to generate natural text just on its own, simply by drawing one word at a time $w_t \sim p(w_t | w_{t-1}, \ldots w_1)$. Quite unlike the monkey using a typewriter, all text emerging from such a model would pass as natural language, e.g. English text. Furthermore, it would be sufficient for generating a meaningful dialog, simply by conditioning the text on previous dialog fragments. Clearly we are still very far from designing such a system, since it would need to *understand* the text rather than just generate grammatically sensible content.

Nonetheless language models are of great service even in their limited form. For instance, the phrases *to recognize speech'* and *to wreck a nice beach'* sound very similar. This can cause ambiguity in speech

recognition, ambiguity that is easily resolved through a language model which rejects the second translation as outlandish. Likewise, in a document summarization algorithm it's worth while knowing that *dog bites man'* is much more frequent than *man bites dog'*, or that *let's eat grandma'* is a rather disturbing statement, whereas *let's eat, grandma'* is much more benign.

7.2.1 Estimating a language model

The obvious question is how we should model a document, or even a sequence of words. We can take recourse to the analysis we applied to sequence models in the previous section. Let's start by applying basic probability rules:

$$p(w_1, w_2, \ldots, w_T) = \prod_{t=1}^{T} p(w_t | w_1, \ldots, w_{t-1}).$$

For example, the probability of a text sequence containing four tokens consisting of words and punctuation would be given as:

$$p(\text{Statistics}, \text{is}, \text{fun}, .) = p(\text{Statistics})p(\text{is}|\text{Statistics})p(\text{fun}|\text{Statistics}, \text{is})p(.|\text{Statistics}, \text{is}, \text{fun}).$$

In order to compute the language model, we need to calculate the probability of words and the conditional probability of a word given the previous few words, i.e. language model parameters. Here, we assume that the training data set is a large text corpus, such as all Wikipedia entries, Project Gutenberg, or all text posted online on the web. The probability of words can be calculated from the relative word frequency of a given word in the training data set.

For example, $p(\text{Statistics})$ can be calculated as the probability of any sentence starting with the word statistics'. A slightly less accurate approach would be to count all occurrences of the word statistics' and divide it by the total number of words in the corpus. This works fairly well, particularly for frequent words. Moving on, we could attempt to estimate

$$\hat{p}(\text{is}|\text{Statistics}) = \frac{n(\text{Statistics is})}{n(\text{Statistics})}.$$

Here $n(w)$ and $n(w, w')$ are the number of occurrences of singletons and pairs of words respectively. Unfortunately, estimating the probability of a word pair is somewhat more difficult, since the occurrences of *Statistics is'* are a lot less frequent. In particular, for some unusual word combinations it may be tricky to find enough occurrences to get accurate estimates. Things take a turn for the worse for 3 word combinations and beyond. There will be many plausible 3-word combinations that we likely won't see in our dataset. Unless we provide some solution to give such word combinations nonzero weight we will not be able to use these as a language model. If the dataset is small or if the words are very rare, we might not find even a single one of them.

A common strategy is to perform some form of Laplace smoothing. We already encountered this in our discussion of *Naive Bayes* where the solution was to add a small constant to all counts. This helps with

singletons, e.g. via

$$\hat{p}(w) = \frac{n(w) + \epsilon_1/m}{n + \epsilon_1}$$

$$\hat{p}(w'|w) = \frac{n(w, w') + \epsilon_2 \hat{p}(w')}{n(w) + \epsilon_2}$$

$$\hat{p}(w''|w', w) = \frac{n(w, w', w'') + \epsilon_3 \hat{p}(w', w'')}{n(w, w') + \epsilon_3}$$

Here the coefficients $\epsilon_i > 0$ determine how much we use the estimate for a shorter sequence as a fill-in for longer ones. Moreover, m is the total number of words we encounter. The above is a rather primitive variant of what is Kneser-Ney smoothing and Bayesian Nonparametrics can accomplish. See e.g. the Sequence Memoizer of Wood et al., 2012 for more details of how to accomplish this. Unfortunately models like this get unwieldy rather quickly: first off, we need to store all counts and secondly, this entirely ignores the meaning of the words. For instance, *cat'* and *feline'* should occur in related contexts. Deep learning based language models are well suited to take this into account. This, it is quite difficult to adjust such models to additional context. Lastly, long word sequences are almost certain to be novel, hence a model that simply counts the frequency of previously seen word sequences is bound to perform poorly there.

7.2.2 Markov Models and n-grams

Before we discuss solutions involving deep learning we need some more terminology and concepts. Recall our discussion of Markov Models in the previous section. Let's apply this to language modeling. A distribution over sequences satisfies the Markov property of first order if $p(w_{t+1}|w_t, \ldots w_1) = p(w_{t+1}|w_t)$. Higher orders correspond to longer dependencies. This leads to a number of approximations that we could apply to model a sequence:

$$p(w_1, w_2, w_3, w_4) = p(w_1)p(w_2)p(w_3)p(w_4)$$

$$p(w_1, w_2, w_3, w_4) = p(w_1)p(w_2|w_1)p(w_3|w_2)p(w_4|w_3)$$

$$p(w_1, w_2, w_3, w_4) = p(w_1)p(w_2|w_1)p(w_3|w_1, w_2)p(w_4|w_2, w_3)$$

Since they involve one, two or three terms, these are typically referred to as unigram, bigram and trigram models. In the following we will learn how to design better models.

7.2.3 Natural Language Statistics

Let's see how this works on real data. To get started we load text from H.G. Wells' Time Machine. This is a fairly small corpus of just over 30,000 words but for the purpose of what we want to illustrate this is just fine. More realistic document collections contain many billions of words. To begin, we split the document up into words and ignore punctuation and capitalization. While this discards some relevant information, it is useful for computing count statistics in general. Let's see what the first few lines look like.

```
In [1]: import sys
        sys.path.insert(0, '..')

        import collections
        import re
        with open('../data/timemachine.txt', 'r') as f:
            lines = f.readlines()
            raw_dataset = [re.sub('[^A-Za-z]+', ' ', st).lower().split()
                           for st in lines]

        # Let's read the first 10 lines of the text
        for st in raw_dataset[8:10]:
            print('# tokens:', len(st), st)

# tokens: 13 ['the', 'time', 'traveller', 'for', 'so', 'it', 'will', 'be',
↪  'convenient', 'to', 'speak', 'of', 'him']
# tokens: 12 ['was', 'expounding', 'a', 'recondite', 'matter', 'to', 'us', 'his',
↪  'grey', 'eyes', 'shone', 'and']
```

Now we need to insert this into a word counter. This is where the `collections` datastructure comes in handy. It takes care of all the accounting for us.

```
In [2]: counter = collections.Counter([tk for st in raw_dataset for tk in st])
        print("frequency of 'traveller':", counter['traveller'])
        # Print the 10 most frequent words with word frequency count
        print(counter.most_common(10))

frequency of 'traveller': 61
[('the', 2261), ('i', 1267), ('and', 1245), ('of', 1155), ('a', 816), ('to', 695),
↪  ('was', 552), ('in', 541), ('that', 443), ('my', 440)]
```

As we can see, the most popular words are actually quite boring to look at. In traditional NLP they're often referred to as stopwords and thus filtered out. That said, they still carry meaning and we will use them nonetheless. However, one thing that is quite clear is that the word frequency decays rather rapidly. The 10th word is less than $\frac{1}{5}$ as common as the most popular one. To get a better idea we plot the graph of word frequencies.

```
In [3]: %matplotlib inline
        from matplotlib import pyplot as plt
        from IPython import display
        display.set_matplotlib_formats('svg')

        wordcounts = [count for _,count in counter.most_common()]
        plt.loglog(wordcounts);
```

7. Recurrent Neural Networks
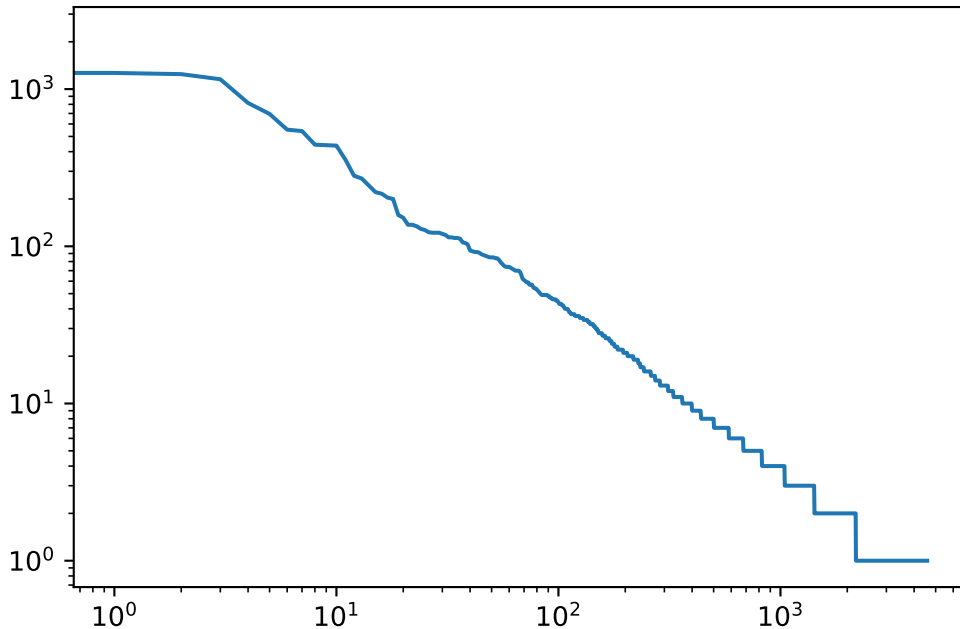
We're on to something quite fundamental here - the word frequencies decay rapidly in a well defined way. After dealing with the first four words as exceptions (the', i', and', of'), all remaining words follow a straight line on a log-log plot. This means that words satisfy [Zipf's law](link) which states that the item frequency is given by

$$n(x) \propto (x + c)^{-\alpha} \text{ and hence } \log n(x) = -\alpha \log(x + c) + \text{const.}$$

This should already give us pause if we want to model words by count statistics and smoothing. After all, we will significantly overestimate the frequency of the tail, aka the infrequent words. But what about word pairs (and trigrams and beyond)? Let's see.

```
In [4]: wseq = [tk for st in raw_dataset for tk in st]
        word_pairs = [pair for pair in zip(wseq[:-1], wseq[1:])]
        print('Beginning of the book\n', word_pairs[:10])
        counter_pairs = collections.Counter(word_pairs)
        print('Most common word pairs\n', counter_pairs.most_common(10))

Beginning of the book
 [('the', 'time'), ('time', 'machine'), ('machine', 'by'), ('by', 'h'), ('h', 'g'),
 ↪  ('g', 'wells'), ('wells', 'i'), ('i', 'the'), ('the', 'time'), ('time',
 ↪  'traveller')]
Most common word pairs
 [(('of', 'the'), 309), (('in', 'the'), 169), (('i', 'had'), 130), (('i', 'was'),
 ↪  112), (('and', 'the'), 109), (('the', 'time'), 102), (('it', 'was'), 99), (('to',
 ↪  'the'), 85), (('as', 'i'), 78), (('of', 'a'), 73)]
```

Two things are notable. Out of the 10 most frequent word pairs, 9 are composed of stop words and only

---

one is relevant to the actual book - the time'. Let's see whether the bigram frequencies behave in the same manner as the unigram frequencies.

```
In [5]: word_triples = [triple for triple in zip(wseq[:-2], wseq[1:-1], wseq[2:])]
        counter_triples = collections.Counter(word_triples)

        bigramcounts = [count for _,count in counter_pairs.most_common()]
        triplecounts = [count for _,count in counter_triples.most_common()]
        plt.loglog(wordcounts, label='word counts');
        plt.loglog(bigramcounts, label='bigram counts');
        plt.loglog(triplecounts, label='triple counts');
        plt.legend();
```
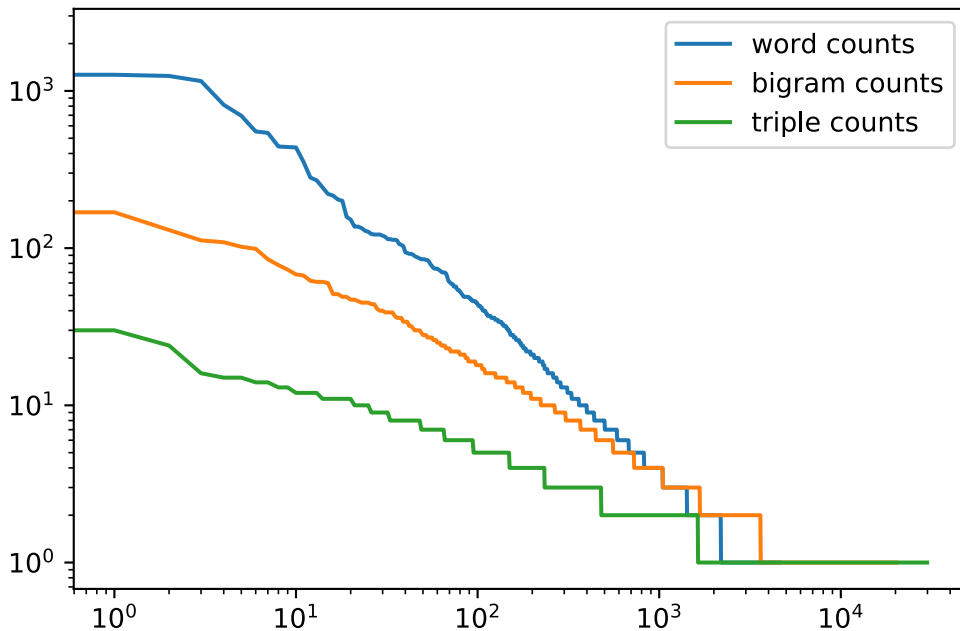

The graph is quite exciting for a number of reasons. Firstly, beyond words, also sequences of words appear to be following Zipf's law, albeit with a lower exponent, depending on sequence length. Secondly, the number of distinct n-grams is not that large. This gives us hope that there is quite a lot of structure in language. Third, *many* n-grams occur very rarely, which makes Laplace smoothing rather unsuitable for language modeling. Instead, we will use deep learning based models.

Summary

- Language models are an important technology for natural language processing.
- n-grams provide a convenient model for dealing with long sequences by truncating the dependence.

- Long sequences suffer from the problem that they occur very rarely or never. This requires smoothing, e.g. via Bayesian Nonparametrics or alternatively via deep learning.

- Zipf's law governs the word distribution for both unigrams and n-grams.

- There's a lot of structure but not enough frequency to deal with infrequent word combinations efficiently via smoothing.

Exercises

1. Suppose there are 100,000 words in the training data set. How many word frequencies and multiword adjacent frequencies does a four-gram need to store?

2. Review the smoothed probability estimates. Why are they not accurate? Hint - we are dealing with a contiguous sequence rather than singletons.

3. How would you model a dialogue?

4. Estimate the exponent of Zipf's law for unigrams, bigrams and trigrams.

Scan the QR Code to Discuss

7.3 Recurrent Neural Networks

In the previous section we introduced n-gram models, where the conditional probability of word w_t at position t only depends on the $n-1$ previous words. If we want to check the possible effect of words earlier than $t-(n-1)$ on w_t, we need to increase n. However, the number of model parameters would also increase exponentially with it, as we need to store $|V|^n$ numbers for a vocabulary V. Hence, rather than modeling $p(w_t|w_{t-1}, \ldots w_{t-n+1})$ it is preferable to use a latent variable model in which we have

$$p(w_t|w_{t-1}, \ldots w_1) \approx p(w_t|h_t(w_{t-1}, h_{t-1})).$$

In theory, for a sufficiently powerful function h_t this is not an approximation. After all, it could simply store all the data it observed so far. We discussed this previously in the *introduction* to the current chapter. Let's see why building such models is a bit more tricky than simple autoregressive models where

$$p(w_t|w_{t-1}, \ldots w_1) \approx p(w_t|f(w_{t-1}, \ldots w_{t-n+1})).$$

As a warmup we will review the latter for discrete outputs and $n = 2$, i.e. for Markov model of first order. To simplify things further we use a single layer in the design of the RNN. Later on we will see how to add more expressivity efficiently across items.

7.3.1 Recurrent Networks Without Hidden States

Let us take a look at a multilayer perceptron with a single hidden layer. Given a mini-batch of instances $\mathbf{X} \in \mathbb{R}^{n \times d}$ with sample size n and d inputs (features or feature vector dimensions). Let the hidden layer's activation function be ϕ. Hence the hidden layer's output $\mathbf{H} \in \mathbb{R}^{n \times h}$ is calculated as

$$\mathbf{H} = \phi(\mathbf{X}\mathbf{W}_{xh} + \mathbf{b}_h),$$

Here, we have the weight parameter $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}$, bias parameter $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$, and the number of hidden units h, for the hidden layer. Recall that $\mathbf{b}_h$ is just a vector - its values are replicated using the broadcasting mechanism) to match those of the matrix-matrix product.

Also note that hidden *state* and hidden *layer* refer to two very different concepts. Hidden layers are, as explained, layers that are hidden from view on the path from input to output. Hidden states are technically speaking *inputs* to whatever we do at a given step. Instead, they can only be computed by looking at data at previous iterations. In this sense they have much in common with latent variable models in statistics, such as clustering or topic models where e.g. the cluster ID affects the output but cannot be directly observed.

The hidden variable $\mathbf{H}$ is used as the input of the output layer. For classification purposes, such as predicting the next character, the output dimensionality q might e.g. match the number of categories in the classification problem. Lastly the the output layer is given by

$$\mathbf{O} = \mathbf{H}\mathbf{W}_{hq} + \mathbf{b}_q.$$

Here, $\mathbf{O} \in \mathbb{R}^{n \times q}$ is the output variable, $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$ is the weight parameter, and $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$ is the bias parameter of the output layer. If it is a classification problem, we can use softmax$(\mathbf{O})$ to compute the probability distribution of the output category. This is entirely analogous to the regression problem we solved *previously*, hence we omit details. Suffice it to say that we can pick (w_t, w_{t-1}) pairs at random and estimate the parameters $\mathbf{W}$ and $\mathbf{b}$ of our network via autograd and stochastic gradient descent.

7.3.2 Recurrent Networks with Hidden States

Matters are entirely different when we have hidden states. Let's look at the structure in some more detail. Assume that $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ is the mini-batch input and $\mathbf{H}_t \in \mathbb{R}^{n \times h}$ is the hidden layer variable of time step t from the sequence. Unlike the multilayer perceptron, here we save the hidden variable $\mathbf{H}_{t-1}$ from the previous time step and introduce a new weight parameter $\mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$, to describe how to use the hidden variable of the previous time step in the current time step. Specifically, the calculation of the hidden variable of the current time step is determined by the input of the current time step together with

the hidden variable of the previous time step:

$$\mathbf{H}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h).$$

Compared with the multilayer perceptron, we added one more $\mathbf{H}_{t-1}\mathbf{W}_{hh}$ here. From the relationship between hidden variables $\mathbf{H}_t$ and $\mathbf{H}_{t-1}$ of adjacent time steps, we know that those variables captured and retained the sequence's historical information up to the current time step, just like the state or memory of the neural network's current time step. Therefore, such a hidden variable is also called a hidden state. Since the hidden state uses the same definition of the previous time step in the current time step, the computation of the equation above is recurrent, hence the name recurrent neural network (RNN).

There are many different RNN construction methods. RNNs with a hidden state defined by the equation above are very common. For time step t, the output of the output layer is similar to the computation in the multilayer perceptron:

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q$$

RNN parameters include the weight $\mathbf{W}_{xh} \in \mathbb{R}^{d \times h}, \mathbf{W}_{hh} \in \mathbb{R}^{h \times h}$ of the hidden layer with the bias $\mathbf{b}_h \in \mathbb{R}^{1 \times h}$, and the weight $\mathbf{W}_{hq} \in \mathbb{R}^{h \times q}$ of the output layer with the bias $\mathbf{b}_q \in \mathbb{R}^{1 \times q}$. It is worth mentioning that RNNs always use these model parameters, even for different time steps. Therefore, the number of RNN model parameters does not grow as the number of time steps increases.

The figure below shows the computational logic of an RNN at three adjacent time steps. In time step t, the computation of the hidden state can be treated as an entry of a fully connected layer with the activation function ϕ after concatenating the input $\mathbf{X}_t$ with the hidden state $\mathbf{H}_{t-1}$ of the previous time step. The output of the fully connected layer is the hidden state of the current time step $\mathbf{H}_t$. Its model parameter is the concatenation of $\mathbf{W}_{xh}$ and $\mathbf{W}_{hh}$, with a bias of $\mathbf{b}_h$. The hidden state of the current time step t $\mathbf{H}_t$ will participate in computing the hidden state $\mathbf{H}_{t+1}$ of the next time step $t+1$, the result of which will become the input for the fully connected output layer of the current time step.
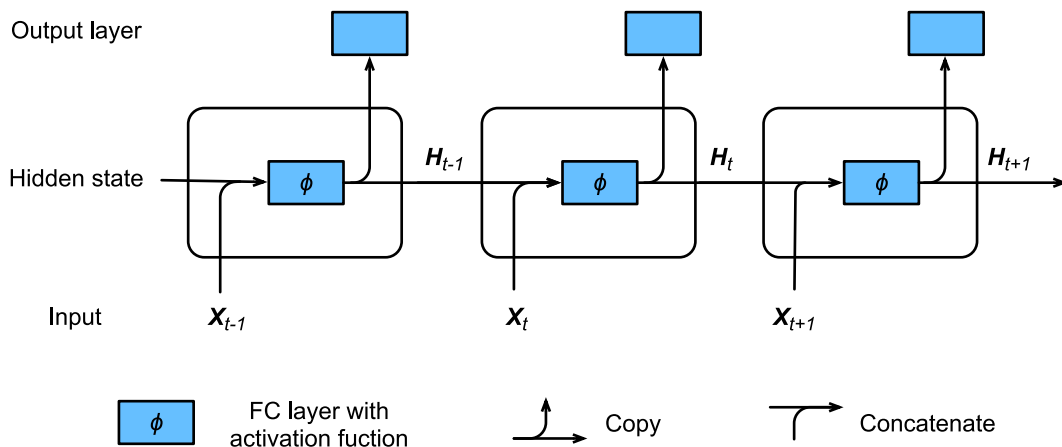
Fig. 7.2: An RNN with a hidden state.

As discussed, the computation in the hidden state uses $\mathbf{H}_t = \mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh}$ to generate an object matching $\mathbf{H}_{t-1}$ in dimensionality. Moreover, we use $\mathbf{H}_t$ to generate the output $\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq}$.

```
In [1]: from mxnet import nd

        # Data X and hidden state H
        X = nd.random.normal(shape=(3, 1))
        H = nd.random.normal(shape=(3, 2))

        # Weights
        W_xh = nd.random.normal(shape=(1, 2))
        W_hh = nd.random.normal(shape=(2, 2))
        W_hq = nd.random.normal(shape=(2, 3))

        def net(X, H):
            H = nd.relu(nd.dot(X, W_xh) + nd.dot(H, W_hh))
            O = nd.relu(nd.dot(H, W_hq))
            return H, O
```

The recurrent network defined above takes observations X and a hidden state H as arguments and uses them to update the hidden state and emit an output O. Since this chain could go on for a very long time, training the model with backprop is out of the question (at least without some approximation). After all, this leads to a very long chain of dependencies that would be prohibitive to solve exactly: books typically have more than 100,000 characters and it is unreasonable to assume that the later text relies indiscriminately on all occurrences that happened, say, 10,000 characters in the past. Truncation methods such as *BPTT* and *Long Short Term Memory* are useful to address this in a more principled manner. For now, let's see how a state update works.

```
In [2]: (H, O) = net(X,H)
        print(H, O)
```

```
[[0. 0.]
 [0. 0.]
 [0. 0.]]
<NDArray 3x2 @cpu(0)>
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
<NDArray 3x3 @cpu(0)>
```

7.3.3 Steps in a Language Model

We conclude this section by illustrating how RNNs can be used to build a language model. For simplicity of illustration we use words rather than characters, since the former are easier to comprehend. Let the number of mini-batch examples be 1, and the sequence of the text be the beginning of our dataset, i.e. the time machine by h. g. wells. The figure below illustrates how to estimate the next character based on the present and previous characters. During the training process, we run a softmax operation on the output from the output layer for each time step, and then use the cross-entropy loss function to compute the error between the result and the label. Due to the recurrent computation of the hidden state in the hidden layer, the output of time step 3 O_3 is determined by the text sequence the, time, machine. Since the next word of the sequence in the training data is by, the loss of time step 3 will depend on the probability distribution of the next word generated based on the sequence the, time, machine and the label by of this time step.
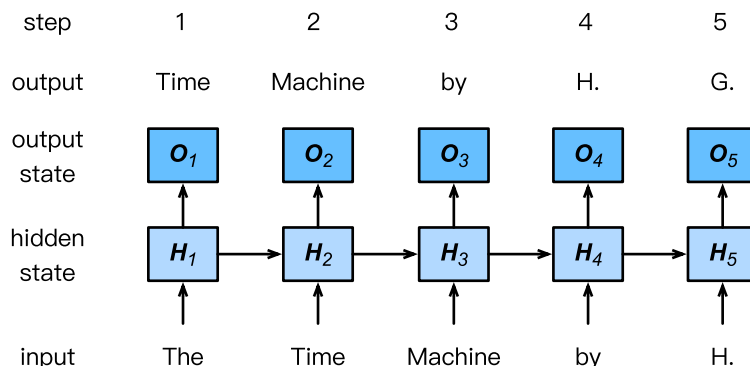
Fig. 7.3: Word-level RNN language model. The input and label sequences are `The Time Machine by H.` and `Time Machine by H. G.` respectively.

The number of words is huge compared to the number of characters. This is why quite often (such as in the subsequent sections) we will use a character-level RNN instead. In the next few sections, we will introduce its implementation.

Summary

- A network that uses recurrent computation is called a recurrent neural network (RNN).
- The hidden state of the RNN can capture historical information of the sequence up to the current time step.
- The number of RNN model parameters does not grow as the number of time steps increases.
- We can create language models using a character-level RNN.

Exercises

1. If we use an RNN to predict the next character in a text sequence, how many output dimensions do we need?
2. Can you design a mapping for which an RNN with hidden states is exact? Hint - what about a finite number of words?
3. What happens to the gradient if you backpropagate through a long sequence?
4. What are some of the problems associated with the simple sequence model described above?

Scan the QR Code to Discuss

7.4 Text Preprocessing

In the previous section we discussed some properties that make language unique. The key is that the number of tokens (aka words) is large and very unevenly distributed. Hence, a naive multiclass classification approach to predict the next symbol doesn't always work very well. Moreover, we need to turn text into a format that we can optimize over, i.e. we need to map it to vectors. At its extreme we have two alternatives. One is to treat each word as a unique entity, as proposed e.g. by Salton, Wong and Yang, 1975. The problem with this strategy is that we might well have to deal with 100,000 to 1,000,000 vectors for very large and diverse corpora.

At the other extreme lies the strategy to predict one character at a time, as suggested e.g. by Ling et al., 2015. A good balance in between both strategies is byte-pair encoding, as described by Sennrich, Haddow and Birch, 2015 for the purpose of neural machine translation. It decomposes text into

syllable-like fragments that occur frequently. This allows for models that are able to generate words like `heteroscedastic` or `pentagram` based on previously viewed words, e.g. `heterogeneous`, `homoscedastic`, `diagram`, and `pentagon`. Going into details of these models is beyond the scope of the current chapter. We will address this later when discussing Natural Language Processing in much more detail. Suffice it to say that it can contribute significantly to the accuracy of natural language processing models.

For the sake of simplicity we will limit ourselves to pure character sequences. We use H.G. Wells' *The Timemachine* as before. We begin by filtering the text and convert it into a a sequence of character IDs.

7.4.1 Data Loading

We begin, as before, by loading the data and by mapping it into a sequence of whitespaces, punctuation signs and regular characters. Preprocessing is minimal and we limit ourselves to removing multiple whitespaces.

```
In [1]: import sys
        sys.path.insert(0, '..')

        from mxnet import nd
        import random

        with open('../data/timemachine.txt', 'r') as f:
            lines = f.readlines()
            raw_dataset = ' '.join(' '.join(lines).lower().split())

        print('number of characters: ', len(raw_dataset))
        print(raw_dataset[0:70])

number of characters:  178605
the time machine, by h. g. wells [1898] i the time traveller (for so i
```

7.4.2 Character Index

This data set has more than 178,000 characters. Next we need to identify all the characters occurring in the document. That is, we want to map each character to continuous integers starting from 0, also known as an index, to facilitate subsequent data processing. To get the index, we extract all the different characters in the data set and then map them to the index one by one to construct the dictionary. Then, print `vocab_size`, which is the number of different characters in the dictionary, i.e. the dictionary size.

```
In [2]: idx_to_char = list(set(raw_dataset))
        char_to_idx = dict([(char, i) for i, char in enumerate(idx_to_char)])
        vocab_size = len(char_to_idx)
        print(char_to_idx)

{'1': 0, ',': 1, 'u': 2, '?': 3, 'p': 4, 't': 5, 'm': 6, 'x': 7, '!': 8, 'w': 9, 'i':
↪  10, 'r': 11, 'c': 12, 'v': 13, 'e': 14, ')': 15, ' ': 16, '9': 17, '.': 18, 'o':
↪  19, '8': 20, ']': 21, "'": 22, '[': 23, 'g': 24, 'k': 25, '-': 26, 'a': 27, '(':
↪  28, 'y': 29, 'f': 30, 'd': 31, '_': 32, 's': 33, 'n': 34, 'q': 35, 'h': 36, 'z':
↪  37, ';': 38, '"': 39, ':': 40, 'j': 41, 'l': 42, 'b': 43}
```

After that, each character in the training data set is converted into an index ID. To illustrate things we print the first 20 characters and their corresponding indices.

```
In [3]: corpus_indices = [char_to_idx[char] for char in raw_dataset]
        sample = corpus_indices[:20]
        print('chars:', ''.join([idx_to_char[idx] for idx in sample]))
        print('indices:', sample)

chars: the time machine, by
indices: [5, 36, 14, 16, 5, 10, 6, 14, 16, 6, 27, 12, 36, 10, 34, 14, 1, 16, 43, 29]
```

We packaged the above code in the `(corpus_indices, char_to_idx, idx_to_char, vocab_size) = load_data_timemachine()` function of the `d2l` package to make it easier to call it in later chapters.

7.4.3 Training Data Preparation

During training, we need to read mini-batches of examples and labels at random. Since sequence data is by its very nature sequential, we need to address the issue of processing it. We did so in a rather ad-hoc manner when we introduced *Sequence Models*. Let's formalize this a bit. Consider the beginning of the book we just processed. If we want to split it up into sequences of 5 symbols each, we have quite some freedom since we could pick an arbitrary offset.

Fig. 7.4: Different offsets lead to different subsequences when splitting up text.

In fact, any one of these offsets is fine. Hence, which one should we pick? In fact, all of them are equally good. But if we pick all offsets we end up with rather redundant data due to overlap, particularly if the sequences are long. Picking just a random set of initial positions is no good either since it does not guarantee uniform coverage of the array. For instance, if we pick n elements at random out of a set of n with random replacement, the probability for a particular element not being picked is $(1 - 1/n)^n \to e^{-1}$. This means that we cannot expect uniform coverage this way. Even randomly permuting a set of all offsets does not offer good guarantees. Instead we can use a simple trick to get both *coverage* and *randomness*: use a random offset, after which one uses the terms sequentially. We describe how to accomplish this for both random sampling and sequential partitioning strategies below.

7. Recurrent Neural Networks

Random sampling

The following code randomly generates a minibatch from the data each time. Here, the batch size `batch_size` indicates to the number of examples in each mini-batch and `num_steps` is the length of the sequence (or time steps if we have a time series) included in each example. In random sampling, each example is a sequence arbitrarily captured on the original sequence. The positions of two adjacent random mini-batches on the original sequence are not necessarily adjacent. The target is to predict the next character based on what we've seen so far, hence the labels are the original sequence, shifted by one character. Note that this is not recommended for latent variable models, since we do not have access to the hidden state *prior* to seeing the sequence. We packaged the above code in the `load_data_timemachine` function of the d2l package to make it easier to call it in later chapters. It returns four variables: `corpus_indices`, `char_to_idx`, `idx_to_char`, and `vocab_size`.

```
In [4]: # This function is saved in the d2l package for future use
        def data_iter_random(corpus_indices, batch_size, num_steps, ctx=None):
            # Offset for the iterator over the data for uniform starts
            offset = int(random.uniform(0,num_steps))
            corpus_indices = corpus_indices[offset:]
            # Subtract 1 extra since we need to account for the sequence length
            num_examples = ((len(corpus_indices) - 1) // num_steps) - 1
            # Discard half empty batches
            num_batches = num_examples // batch_size
            example_indices = list(range(0, num_examples * num_steps, num_steps))
            random.shuffle(example_indices)

            # This returns a sequence of the length num_steps starting from pos
            def _data(pos):
                return corpus_indices[pos: pos + num_steps]

            for i in range(0, batch_size * num_batches, batch_size):
                # Batch_size indicates the random examples read each time
                batch_indices = example_indices[i:(i+batch_size)]
                X = [_data(j) for j in batch_indices]
                Y = [_data(j + 1) for j in batch_indices]
                yield nd.array(X, ctx), nd.array(Y, ctx)
```

Let us generate an artificial sequence from 0 to 30. We assume that the batch size and numbers of time steps are 2 and 5 respectively. This means that depending on the offset we can generate between 4 and 5 (x, y) pairs. With a minibatch size of 2 we only get 2 minibatches.

```
In [5]: my_seq = list(range(30))
        for X, Y in data_iter_random(my_seq, batch_size=2, num_steps=5):
            print('X: ', X, '\nY:', Y)

X:
[[15. 16. 17. 18. 19.]
 [ 0.  1.  2.  3.  4.]]
<NDArray 2x5 @cpu(0)>
Y:
[[16. 17. 18. 19. 20.]
 [ 1.  2.  3.  4.  5.]]
<NDArray 2x5 @cpu(0)>
X:
[[10. 11. 12. 13. 14.]
```

```
 [ 5.  6.  7.  8.  9.]]
<NDArray 2x5 @cpu(0)>
Y:
[[11. 12. 13. 14. 15.]
 [ 6.  7.  8.  9. 10.]]
<NDArray 2x5 @cpu(0)>
```

Sequential partitioning

In addition to random sampling of the original sequence, we can also make the positions of two adjacent random mini-batches adjacent in the original sequence. Now, we can use a hidden state of the last time step of a mini-batch to initialize the hidden state of the next mini-batch, so that the output of the next mini-batch is also dependent on the input of the mini-batch, with this pattern continuing in subsequent mini-batches. This has two effects on the implementation of a recurrent neural network. On the one hand, when training the model, we only need to initialize the hidden state at the beginning of each epoch. On the other hand, when multiple adjacent mini-batches are concatenated by passing hidden states, the gradient calculation of the model parameters will depend on all the mini-batch sequences that are concatenated. In the same epoch as the number of iterations increases, the costs of gradient calculation rise. So that the model parameter gradient calculations only depend on the mini-batch sequence read by one iteration, we can separate the hidden state from the computational graph before reading the mini-batch (this can be done by detaching the graph). We will gain a deeper understand this approach in the following sections.

```
In [6]: # This function is saved in the d2l package for future use
        def data_iter_consecutive(corpus_indices, batch_size, num_steps, ctx=None):
            # Offset for the iterator over the data for uniform starts
            offset = int(random.uniform(0,num_steps))
            # Slice out data - ignore num_steps and just wrap around
            num_indices = ((len(corpus_indices) - offset) // batch_size) * batch_size
            indices = nd.array(corpus_indices[offset:(offset + num_indices)], ctx=ctx)
            indices = indices.reshape((batch_size,-1))
            # Need to leave one last token since targets are shifted by 1
            num_epochs = ((num_indices // batch_size) - 1) // num_steps

            for i in range(0, num_epochs * num_steps, num_steps):
                X = indices[:,i:(i+num_steps)]
                Y = indices[:,(i+1):(i+1+num_steps)]
                yield X, Y
```

Using the same settings, print input `X` and label `Y` for each mini-batch of examples read by random sampling. The positions of two adjacent random mini-batches on the original sequence are adjacent.

```
In [7]: for X, Y in data_iter_consecutive(my_seq, batch_size=2, num_steps=6):
            print('X: ', X, '\nY:', Y)

X:
[[ 2.  3.  4.  5.  6.  7.]
 [16. 17. 18. 19. 20. 21.]]
<NDArray 2x6 @cpu(0)>
Y:
[[ 3.  4.  5.  6.  7.  8.]
 [17. 18. 19. 20. 21. 22.]]
<NDArray 2x6 @cpu(0)>
```

```
X:
[[ 8.  9. 10. 11. 12. 13.]
 [22. 23. 24. 25. 26. 27.]]
<NDArray 2x6 @cpu(0)>
Y:
[[ 9. 10. 11. 12. 13. 14.]
 [23. 24. 25. 26. 27. 28.]]
<NDArray 2x6 @cpu(0)>
```

Sequential partitioning decomposes the sequence into `batch_size` many strips of data which are traversed as we iterate over minibatches. Note that the i-th element in a minibatch matches with the i-th element of the next minibatch rather than within a minibatch.

Summary

- Documents are preprocessed by tokenizing the words and mapping them into IDs. There are multiple methods:

  - Character encoding which uses individual characters (good e.g. for Chinese)

  - Word encoding (good e.g. for English)

  - Byte-pair encoding (good for languages that have lots of morphology, e.g. German)

- The main choices for sequence partitioning are whether we pick consecutive or random sequences. In particular for recurrent networks the former is critical.

- Given the overall document length, it is usually acceptable to be slightly wasteful with the documents and discard half-empty minibatches.

Exercises

1. Which other other mini-batch data sampling methods can you think of?

2. Why is it a good idea to have a random offset?

   - Does it really lead to a perfectly uniform distribution over the sequences on the document?

   - What would you have to do to make things even more uniform?

3. If we want a sequence example to be a complete sentence, what kinds of problems does this introduce in mini-batch sampling? Why would we want to do this anyway?

7.5 Implementation of Recurrent Neural Networks from Scratch

In this section, we will implement a language model from scratch. It is based on a character-level recurrent neural network that is trained on H. G. Wells' The Time Machine'. As before, we start by reading the dataset first.

```
In [1]: import sys
        sys.path.insert(0, '..')

        import d2l
        import math
        from mxnet import autograd, nd
        from mxnet.gluon import loss as gloss
        import time

        (corpus_indices, char_to_idx, idx_to_char, vocab_size) = \
            d2l.load_data_time_machine()
```

7.5.1 One-hot Encoding

One-hot encoding vectors provide an easy way to express words as vectors in order to process them in a deep network. In a nutshell, we map each word to a different unit vector: assume that the number of different characters in the dictionary is N (the `vocab_size`) and each character has a one-to-one correspondence with a single value in the index of successive integers from 0 to $N-1$. If the index of a character is the integer i, then we create a vector $\mathbf{e}_i$ of all 0s with a length of N and set the element at position i to 1. This vector is the one-hot vector of the original character. The one-hot vectors with indices 0 and 2 are shown below (the length of the vector is equal to the dictionary size).

```
In [2]: nd.one_hot(nd.array([0, 2]), vocab_size)

Out[2]:
        [[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
          0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
         [0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
          0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
        <NDArray 2x43 @cpu(0)>
```

The shape of the mini-batch we sample each time is (batch size, time step). The following function transforms such mini-batches into a number of matrices with the shape of (batch size, dictionary size) that can be entered into the network. The total number of vectors is equal to the number of time steps. That is, the input of time step t is $\boldsymbol{X}_t \in \mathbb{R}^{n \times d}$, where n is the batch size and d is the number of inputs. That is the one-hot vector length (the dictionary size).

```
In [3]: # This function is saved in the d2l package for future use
        def to_onehot(X, size):
            return [nd.one_hot(x, size) for x in X.T]

        X = nd.arange(10).reshape((2, 5))
        inputs = to_onehot(X, vocab_size)
        len(inputs), inputs[0].shape
Out[3]: (5, (2, 43))
```

The code above generates 5 minibatches containing 2 vectors each. Since we have a total of 43 distinct symbols in The Time Machine we get 43-dimensional vectors.

7.5.2 Initializing the Model Parameters

Next, we initialize the model parameters. The number of hidden units `num_hiddens` is a tunable parameter.

```
In [4]: num_inputs, num_hiddens, num_outputs = vocab_size, 512, vocab_size
        ctx = d2l.try_gpu()
        print('Using', ctx)

        # Create the parameters of the model, initialize them and attach gradients
        def get_params():
            def _one(shape):
                return nd.random.normal(scale=0.01, shape=shape, ctx=ctx)

            # Hidden layer parameters
            W_xh = _one((num_inputs, num_hiddens))
            W_hh = _one((num_hiddens, num_hiddens))
            b_h = nd.zeros(num_hiddens, ctx=ctx)
            # Output layer parameters
            W_hq = _one((num_hiddens, num_outputs))
            b_q = nd.zeros(num_outputs, ctx=ctx)
            # Attach a gradient
            params = [W_xh, W_hh, b_h, W_hq, b_q]
            for param in params:
                param.attach_grad()
            return params
Using gpu(0)
```

7.5.3 Sequence Modeling

RNN Model

We implement this model based on the definition of an RNN. First, we need an `init_rnn_state` function to return the hidden state at initialization. It returns a tuple consisting of an NDArray with a value of 0 and a shape of (batch size, number of hidden units). Using tuples makes it easier to handle situations where the hidden state contains multiple NDArrays (e.g. when combining multiple layers in an RNN).

```
In [5]: def init_rnn_state(batch_size, num_hiddens, ctx):
            return (nd.zeros(shape=(batch_size, num_hiddens), ctx=ctx), )
```

The following `rnn` function defines how to compute the hidden state and output in a time step. The activation function here uses the tanh function. As described in the Multilayer Perceptron section, the mean value of tanh function values is 0 when the elements are evenly distributed over the real number field.

```
In [6]: def rnn(inputs, state, params):
            # Both inputs and outputs are composed of num_steps matrices of the shape
            # (batch_size, vocab_size)
            W_xh, W_hh, b_h, W_hq, b_q = params
            H, = state
            outputs = []
            for X in inputs:
                H = nd.tanh(nd.dot(X, W_xh) + nd.dot(H, W_hh) + b_h)
                Y = nd.dot(H, W_hq) + b_q
                outputs.append(Y)
            return outputs, (H,)
```

Let's run a simple test to check whether inputs and outputs are accurate. In particular, we check output dimensions, the number of outputs and ensure that the hidden state hasn't changed.

```
In [7]: state = init_rnn_state(X.shape[0], num_hiddens, ctx)
        inputs = to_onehot(X.as_in_context(ctx), vocab_size)
        params = get_params()
        outputs, state_new = rnn(inputs, state, params)
        len(outputs), outputs[0].shape, state_new[0].shape

Out[7]: (5, (2, 43), (2, 512))
```

Prediction Function

The following function predicts the next `num_chars` characters based on the `prefix` (a string containing several characters). This function is a bit more complicated. In it, we set the recurrent neural unit `rnn` as a function parameter, so that this function can be reused in the other recurrent neural networks described in following sections.

```
In [8]: # This function is saved in the d2l package for future use
        def predict_rnn(prefix, num_chars, rnn, params, init_rnn_state,
                        num_hiddens, vocab_size, ctx, idx_to_char, char_to_idx):
            state = init_rnn_state(1, num_hiddens, ctx)
```

```
output = [char_to_idx[prefix[0]]]
for t in range(num_chars + len(prefix) - 1):
    # The output of the previous time step is taken as the input of the
    # current time step.
    X = to_onehot(nd.array([output[-1]], ctx=ctx), vocab_size)
    # Calculate the output and update the hidden state
    (Y, state) = rnn(X, state, params)
    # The input to the next time step is the character in the prefix or
    # the current best predicted character
    if t < len(prefix) - 1:
        output.append(char_to_idx[prefix[t + 1]])
    else:
        # This is maximum likelihood decoding, not sampling
        output.append(int(Y[0].argmax(axis=1).asscalar()))
return ''.join([idx_to_char[i] for i in output])
```

We test the `predict_rnn` function first. We will create a lyric with a length of 10 characters (regardless of the prefix length) based on the prefix separate. Because the model parameters are random values, the prediction results are also random.

```
In [9]: predict_rnn('traveller', 10, rnn, params, init_rnn_state, num_hiddens,
                    vocab_size, ctx, idx_to_char, char_to_idx)

Out[9]: 'travellernz)me.-!af'
```

7.5.4 Gradient Clipping

When solving an optimization problem we take update steps for the weights $\mathbf{w}$ in the general direction of the negative gradient $\mathbf{g}_t$ on a minibatch, say $\mathbf{w} - \eta \cdot \mathbf{g}_t$. Let's further assume that the objective is well behaved, i.e. it is Lipschitz continuous with constant L, i.e.

$$|l(\mathbf{w}) - l(\mathbf{w}')| \leq L\|\mathbf{w} - \mathbf{w}'\|.$$

In this case we can safely assume that if we update the weight vector by $\eta \cdot \mathbf{g}_t$ we will not observe a change by more than $L\eta\|\mathbf{g}_t\|$. This is both a curse and a blessing. A curse since it limits the speed with which we can make progress, a blessing since it limits the extent to which things can go wrong if we move in the wrong direction.

Sometimes the gradients can be quite large and the optimization algorithm may fail to converge. We could address this by reducing the learning rate η or by some other higher order trick. But what if we only rarely get large gradients? In this case such an approach may appear entirely unwarranted. One alternative is to clip the gradients by projecting them back to a ball of a given radius, say θ via

$$\mathbf{g} \leftarrow \min\left(1, \frac{\theta}{\|\mathbf{g}\|}\right)\mathbf{g}.$$

By doing so we know that the gradient norm never exceeds θ and that the updated gradient is entirely aligned with the original direction $\mathbf{g}$. Back to the case at hand - optimization in RNNs. One of the issues is that the gradients in an RNN may either explode or vanish. Consider the chain of matrix-products involved in backpropagation. If the largest eigenvalue of the matrices is typically larger than 1, then the

product of many such matrices can be much larger than 1. As a result, the aggregate gradient might explode. Gradient clipping provides a quick fix. While it doesn't entire solve the problem, it is one of the many techniques to alleviate it.

```
In [10]: # This function is saved in the d2l package for future use
         def grad_clipping(params, theta, ctx):
             norm = nd.array([0], ctx)
             for param in params:
                 norm += (param.grad ** 2).sum()
             norm = norm.sqrt().asscalar()
             if norm > theta:
                 for param in params:
                     param.grad[:] *= theta / norm
```

7.5.5 Perplexity

One way of measuring how well a sequence model works is to check how surprising the text is. A good language model is able to predict with high accuracy what we will see next. Consider the following continuations of the phrase `It is raining`, as proposed by different language models:

1. `It is raining outside`

2. `It is raining banana tree`

3. `It is raining piouw;kcj pwepoiut`

In terms of quality, example 1 is clearly the best. The words are sensible and logically coherent. While it might not quite so accurately reflect which word follows (`in San Francisco` and `in winter` would have been perfectly reasonable extensions), the model is able to capture which kind of word follows. Example 2 is considerably worse by producing a nonsensical and borderline dysgrammatical extension. Nonetheless, at least the model has learned how to spell words and some degree of correlation between words. Lastly, example 3 indicates a poorly trained model that doesn't fit data.

One way of measuring the quality of the model is to compute $p(w)$, i.e. the likelihood of the sequence. Unfortunately this is a number that is hard to understand and difficult to compare. After all, shorter sequences are *much* more likely than long ones, hence evaluating the model on Tolstoy's magnum opus War and Peace' will inevitably produce a much smaller likelihood than, say, on Saint-Exupery's novella The Little Prince'. What is missing is the equivalent of an average.

Information Theory comes handy here. If we want to compress text we can ask about estimating the next symbol given the current set of symbols. A lower bound on the number of bits is given by $-\log_2 p(w_t|w_{t-1}, \ldots w_1)$. A good language model should allow us to predict the next word quite accurately and thus it should allow us to spend very few bits on compressing the sequence. One way of measuring it is by the average number of bits that we need to spend.

$$\frac{1}{n} \sum_{t=1}^{n} -\log p(w_t|w_{t-1}, \ldots w_1) = \frac{1}{|w|} -\log p(w)$$

This makes the performance on documents of different lengths comparable. For historical reasons scientists in natural language processing prefer to use a quantity called perplexity rather than bitrate. In a nutshell it is the exponential of the above:

$$\text{PPL} := \exp\left(-\frac{1}{n}\sum_{t=1}^{n}\log p(w_t|w_{t-1},\ldots w_1)\right)$$

It can be best understood as the harmonic mean of the number of real choices that we have when deciding which word to pick next. Note that Perplexity naturally generalizes the notion of the cross entropy loss defined when we introduced Softmax Regression. That is, for a single symbol both definitions are identical bar the fact that one is the exponential of the other. Let's look at a number of cases:

- In the best case scenario, the model always estimates the probability of the next symbol as 1. In this case the perplexity of the model is 1.

- In the worst case scenario, the model always predicts the probability of the label category as 0. In this situation, the perplexity is infinite.

- At the baseline, the model predicts a uniform distribution over all tokens. In this case the perplexity equals the size of the dictionary `vocab_size`. In fact, if we were to store the sequence without any compression this would be the best we could do to encode it. Hence this provides a nontrivial upper bound that any model must satisfy.

7.5.6 Training the Model

Training a sequence model proceeds quite different from previous codes. In particular we need to take care of the following changes due to the fact that the tokens appear in order:

1. We use perplexity to evaluate the model. This ensures that different tests are comparable.

2. We clip the gradient before updating the model parameters. This ensures that the model doesn't diverge even when gradients blow up at some point during the training process (effectively it reduces the stepsize automatically).

3. Different sampling methods for sequential data (independent sampling and sequential partitioning) will result in differences in the initialization of hidden states. We discussed these issues in detail when we covered *data processing*.

Optimization Loop

To allow for more flexibility the call signature and the code are slightly longer. This will allow us to replace the various pieces by a Gluon implementation subsequently without the need to change the training logic.

```
In [11]: # This function is saved in the d2l package for future use
         def train_and_predict_rnn(rnn, get_params, init_rnn_state, num_hiddens,
                                   vocab_size, ctx, corpus_indices, idx_to_char,
```

```
                       char_to_idx, is_random_iter, num_epochs, num_steps,
                       lr, clipping_theta, batch_size, pred_period,
                       pred_len, prefixes):
    if is_random_iter:
        data_iter_fn = d2l.data_iter_random
    else:
        data_iter_fn = d2l.data_iter_consecutive
    params = get_params()
    loss = gloss.SoftmaxCrossEntropyLoss()

    for epoch in range(num_epochs):
        if not is_random_iter:
            # If adjacent sampling is used, the hidden state is initialized
            # at the beginning of the epoch
            state = init_rnn_state(batch_size, num_hiddens, ctx)
        l_sum, n, start = 0.0, 0, time.time()
        data_iter = data_iter_fn(corpus_indices, batch_size, num_steps, ctx)
        for X, Y in data_iter:
            if is_random_iter:
                # If random sampling is used, the hidden state is initialized
                # before each mini-batch update
                state = init_rnn_state(batch_size, num_hiddens, ctx)
            else:
                # Otherwise, the detach function needs to be used to separate
                # the hidden state from the computational graph to avoid
                # backpropagation beyond the current sample
                for s in state:
                    s.detach()
            with autograd.record():
                inputs = to_onehot(X, vocab_size)
                # outputs is num_steps terms of shape (batch_size, vocab_size)
                (outputs, state) = rnn(inputs, state, params)
                # After stitching it is (num_steps * batch_size, vocab_size)
                outputs = nd.concat(*outputs, dim=0)
                # The shape of Y is (batch_size, num_steps), and then becomes
                # a vector with a length of batch * num_steps after
                # transposition. This gives it a one-to-one correspondence
                # with output rows
                y = Y.T.reshape((-1,))
                # Average classification error via cross entropy loss
                l = loss(outputs, y).mean()
            l.backward()
            grad_clipping(params, clipping_theta, ctx)  # Clip the gradient
            d2l.sgd(params, lr, 1)
            # Since the error is the mean, no need to average gradients here
            l_sum += l.asscalar() * y.size
            n += y.size

        if (epoch + 1) % pred_period == 0:
            print('epoch %d, perplexity %f, time %.2f sec' % (
                epoch + 1, math.exp(l_sum / n), time.time() - start))
            for prefix in prefixes:
                print(' -', predict_rnn(
                    prefix, pred_len, rnn, params, init_rnn_state,
                    num_hiddens, vocab_size, ctx, idx_to_char, char_to_idx))
```

Experiments with a Sequence Model

Now we can train the model. First, we need to set the model hyper-parameters. To allow for some meaningful amount of context we set the sequence length to 64. To get some intuition of how well the model works, we will have it generate 50 characters every 50 epochs of the training phase. In particular, we will see how training using the separate' and sequential' term generation will affect the performance of the model.

```
In [12]: num_epochs, num_steps, batch_size, lr, clipping_theta = 500, 64, 32, 1e2, 1e-2
         pred_period, pred_len, prefixes = 50, 50, ['traveller', 'time traveller']
```

Let's use random sampling to train the model and produce some text.

```
In [13]: train_and_predict_rnn(rnn, get_params, init_rnn_state, num_hiddens,
                               vocab_size, ctx, corpus_indices, idx_to_char,
                               char_to_idx, True, num_epochs, num_steps, lr,
                               clipping_theta, batch_size, pred_period, pred_len,
                               prefixes)
```

```
epoch 50, perplexity 8.721148, time 0.24 sec
 - traveller the time time time time time time time time time
 - time traveller the time time time time time time time time time
epoch 100, perplexity 7.326673, time 0.24 sec
 - traveller some the the the ghis the the the ghis the the th
 - time traveller some the the the ghis the the the ghis the the th
epoch 150, perplexity 5.526817, time 0.24 sec
 - traveller armeres of the the this this wion it wiscon the p
 - time traveller at wathe the this this wion it wiscon the present
epoch 200, perplexity 3.765694, time 0.24 sec
 - traveller sof course ans of the time traveller procendid th
 - time traveller proceeding sis cont rient--o mor call be the bagt
epoch 250, perplexity 2.451412, time 0.24 sec
 - traveller proweed the psoche of conccent of mesccouth o mim
 - time traveller proweed the psoche of conccent of mesccouth o mim
epoch 300, perplexity 1.789101, time 0.24 sec
 - traveller hard be we hevige ane ram the have nermel the wey
 - time traveller. 'but now you began it reas no. that seas the gar
epoch 350, perplexity 1.546432, time 0.24 sec
 - traveller smiled. 'are you sure we can move freely in two d
 - time traveller smiled. 'are you sure we can move freely in two d
epoch 400, perplexity 1.414555, time 0.24 sec
 - traveller hald bsthe theeground at all that this is so exte
 - time traveller.  'it's against reason,' said filby. 'of course a
epoch 450, perplexity 1.352775, time 0.24 sec
 - traveller, after the pause required for the oropssmal gat,e
 - time traveller, after the pause required for the oropssmal gat,e
epoch 500, perplexity 1.368491, time 0.24 sec
 - traveller smiled round at us. then, still treyoc arof couer
 - time traveller smiled round at us. then, still teele coner, chie
```

Even though our model was rather primitive, it is nonetheless able to produce text that resembles language. In particular it learns some notion of quotations, punctuation and a basic sense of grammar, at least for frequent words. Now let's compare this with sequential partitioning.

```
In [14]: train_and_predict_rnn(rnn, get_params, init_rnn_state, num_hiddens,
                               vocab_size, ctx, corpus_indices, idx_to_char,
```

```
                              char_to_idx, False, num_epochs, num_steps, lr,
                              clipping_theta, batch_size, pred_period, pred_len,
                              prefixes)
epoch 50, perplexity 8.977015, time 0.24 sec
 - travellereat our and the the the the the the the the th
 - time travellereat ang the the the the the the the the the th
epoch 100, perplexity 7.076854, time 0.24 sec
 - traveller.  'ione mone the the foured the payoully anoully
 - time traveller an anour abe to is oflered the payoully the and f
epoch 150, perplexity 4.557047, time 0.24 sec
 - traveller.  'in asmeding s ance,' this fllbot you the prome
 - time traveller.  'the sure the but beat ongthe bous of specingen
epoch 200, perplexity 2.429720, time 0.24 sec
 - traveller.  'y ullint one hiof d ane urmenteance, hed filby
 - time traveller.  'yous of shed ins ingeat asolist.'  'is all pe
epoch 250, perplexity 1.563480, time 0.23 sec
 - traveller, had he reamone some psallo nseed titheal ats if
 - time traveller s meve lerisklint and thine tile plyor some thay
epoch 300, perplexity 1.293313, time 0.24 sec
 - traveller.  'you can show black is white by argumentatian t
 - time traveller.  'you can show black is white by argumentatian t
epoch 350, perplexity 1.171948, time 0.24 sec
 - traveller. 'it sorather race tale geint mo it nis noubre we
 - time traveller came back, and filby's anecdote collapsed.  the t
epoch 400, perplexity 1.145446, time 0.24 sec
 - traveller came back, and filby's anecdote collapsed.  the t
 - time traveller came back, and filby's anecdote collapsed.  the t
epoch 450, perplexity 1.109065, time 0.24 sec
 - traveller (ficenty in lhire is a of cines of spack an y ore
 - time traveller (miles. 'aly yuragh ding sith neas mathel three d
epoch 500, perplexity 1.106765, time 0.24 sec
 - traveller shiled. 'ereny urma that sube bur ann tha lateere
 - time traveller came back, and filby's anecdote collapsed.  the t
```

The perplexity is quite a bit lower. In fact, both models are pretty close to 1. This means that if we were to compress the text using this simple character-based language model we would needs less than 1 bit per character to encode a symbol. In the following we will see how to improve significantly on the current model and how to make it faster and easier to implement.

Summary

- Sequence models need state initialization for training.

- Between sequential models you need to ensure to detach the gradient, to ensure that the automatic differentiation does not propagate effects beyond the current sample.

- A simple RNN language model consists of an encoder, an RNN model and a decoder.

- Gradient clipping prevents gradient explosion (but it cannot fix vanishing gradients).

- Perplexity calibrates model performance across variable sequence length. It is the exponentiated average of the cross-entropy loss.

- Sequential partitioning typically leads to better models.

Exercises

1. Show that one-hot encoding is equivalent to picking a different embedding for each object.

2. Adjust the hyperparameters to improve the perplexity.

   - How low can you go? Adjust embeddings, hidden units, learning rate, etc.
   - How well will it work on other books by H. G. Wells, e.g. The War of the Worlds.

3. Run the code in this section without clipping the gradient. What happens?

4. Set the `pred_period` variable to 1 to observe how the under-trained model (high perplexity) writes lyrics. What can you learn from this?

5. Change adjacent sampling so that it does not separate hidden states from the computational graph. Does the running time change? How about the accuracy?

6. Replace the activation function used in this section with ReLU and repeat the experiments in this section.

7. Prove that the perplexity is the inverse of the harmonic mean of the conditional word probabilities.

Scan the QR Code to Discuss

7.6 Concise Implementation of Recurrent Neural Networks

@TODO(smolix/astonzhang): the data set was just changed from lyrics to time machine, so descriptions/hyperparameters have to change.

This section will implement a language model based on a recurrent neural network more concisely. First, we read the time machine data set.

```
In [1]: import sys
        sys.path.insert(0, '..')

        import d2l
        import math
        from mxnet import autograd, gluon, init, nd
```

```
from mxnet.gluon import loss as gloss, nn, rnn
import time

(corpus_indices, char_to_idx, idx_to_char,
 vocab_size) = d2l.load_data_time_machine()
```

7.6.1 Define the Model

Gluon's `rnn` module provides a recurrent neural network implementation. Next, we construct the recurrent neural network layer `rnn_layer` with a single hidden layer and 256 hidden units, and initialize the weights.

```
In [2]: num_hiddens = 256
        rnn_layer = rnn.RNN(num_hiddens)
        rnn_layer.initialize()
```

Then, we call the `rnn_layer`'s member function `begin_state` to return hidden state list for initialization. It has an element of the shape (number of hidden layers, batch size, number of hidden units).

```
In [3]: batch_size = 2
        state = rnn_layer.begin_state(batch_size=batch_size)
        state[0].shape
```

```
Out[3]: (1, 2, 256)
```

Unlike the recurrent neural network implemented in the previous section, the input shape of `rnn_layer` here is (time step, batch size, number of inputs). Here, the number of inputs is the one-hot vector length (the dictionary size). In addition, as an `rnn.RNN` instance in Gluon, `rnn_layer` returns the output and hidden state after forward computation. The output refers to the hidden states that the hidden layer computes and outputs at various time steps, which are usually used as input for subsequent output layers. We should emphasize that the output itself does not involve the computation of the output layer, and its shape is (time step, batch size, number of hidden units). While the hidden state returned by the `rnn.RNN` instance in the forward computation refers to the hidden state of the hidden layer available at the last time step that can be used to initialize the next time step: when there are multiple layers in the hidden layer, the hidden state of each layer is recorded in this variable. For recurrent neural networks such as long short-term memory networks, the variable also contains other information. We will introduce long short-term memory and deep recurrent neural networks in the later sections of this chapter.

```
In [4]: num_steps = 35
        X = nd.random.uniform(shape=(num_steps, batch_size, vocab_size))
        Y, state_new = rnn_layer(X, state)
        Y.shape, len(state_new), state_new[0].shape
```

```
Out[4]: ((35, 2, 256), 1, (1, 2, 256))
```

Next, we inherit the Block class to define a complete recurrent neural network. It first uses one-hot vector to represent input data and enter it into the `rnn_layer`. This, it uses the fully connected output layer to obtain the output. The number of outputs is equal to the dictionary size `vocab_size`.

```
In [5]: # This class has been saved in the d2l package for future use
        class RNNModel(nn.Block):
            def __init__(self, rnn_layer, vocab_size, **kwargs):
```

```
            super(RNNModel, self).__init__(**kwargs)
            self.rnn = rnn_layer
            self.vocab_size = vocab_size
            self.dense = nn.Dense(vocab_size)

        def forward(self, inputs, state):
            # Get the one-hot vector representation by transposing the input to
            # (num_steps, batch_size)
            X = nd.one_hot(inputs.T, self.vocab_size)
            Y, state = self.rnn(X, state)
            # The fully connected layer will first change the shape of Y to
            # (num_steps * batch_size, num_hiddens)
            # Its output shape is (num_steps * batch_size, vocab_size)
            output = self.dense(Y.reshape((-1, Y.shape[-1])))
            return output, state

        def begin_state(self, *args, **kwargs):
            return self.rnn.begin_state(*args, **kwargs)
```

7.6.2 Model Training

As in the previous section, a prediction function is defined below. The implementation here differs from the previous one in the function interfaces for forward computation and hidden state initialization.

```
In [6]: # This function is saved in the d2l package for future use
        def predict_rnn_gluon(prefix, num_chars, model, vocab_size, ctx, idx_to_char,
                              char_to_idx):
            # Use model's member function to initialize the hidden state
            state = model.begin_state(batch_size=1, ctx=ctx)
            output = [char_to_idx[prefix[0]]]
            for t in range(num_chars + len(prefix) - 1):
                X = nd.array([output[-1]], ctx=ctx).reshape((1, 1))
                # Forward computation does not require incoming model parameters
                (Y, state) = model(X, state)
                if t < len(prefix) - 1:
                    output.append(char_to_idx[prefix[t + 1]])
                else:
                    output.append(int(Y.argmax(axis=1).asscalar()))
            return ''.join([idx_to_char[i] for i in output])
```

Let us make one predication using a model with weights that are random values.

```
In [7]: ctx = d2l.try_gpu()
        model = RNNModel(rnn_layer, vocab_size)
        model.initialize(force_reinit=True, ctx=ctx)
        predict_rnn_gluon('traveller', 10, model, vocab_size, ctx, idx_to_char,
                          char_to_idx)

Out[7]: 'traveller_?xa;_?xa;'
```

Next, implement the training function. Its algorithm is the same as in the previous section, but only random sampling is used here to read the data.

```
In [8]: # This function is saved in the d2l package for future use
        def train_and_predict_rnn_gluon(model, num_hiddens, vocab_size, ctx,
```

```
                                corpus_indices, idx_to_char, char_to_idx,
                                num_epochs, num_steps, lr, clipping_theta,
                                batch_size, pred_period, pred_len, prefixes):
    loss = gloss.SoftmaxCrossEntropyLoss()
    model.initialize(ctx=ctx, force_reinit=True, init=init.Normal(0.01))
    trainer = gluon.Trainer(model.collect_params(), 'sgd',
                            {'learning_rate': lr, 'momentum': 0, 'wd': 0})

    for epoch in range(num_epochs):
        l_sum, n, start = 0.0, 0, time.time()
        data_iter = d2l.data_iter_consecutive(
            corpus_indices, batch_size, num_steps, ctx)
        state = model.begin_state(batch_size=batch_size, ctx=ctx)
        for X, Y in data_iter:
            for s in state:
                s.detach()
            with autograd.record():
                (output, state) = model(X, state)
                y = Y.T.reshape((-1,))
                l = loss(output, y).mean()
            l.backward()
            # Clip the gradient
            params = [p.data() for p in model.collect_params().values()]
            d2l.grad_clipping(params, clipping_theta, ctx)
            # Since the error has already taken the mean, the gradient does
            # not need to be averaged
            trainer.step(1)
            l_sum += l.asscalar() * y.size
            n += y.size

        if (epoch + 1) % pred_period == 0:
            print('epoch %d, perplexity %f, time %.2f sec' % (
                epoch + 1, math.exp(l_sum / n), time.time() - start))
            for prefix in prefixes:
                print(' -', predict_rnn_gluon(
                    prefix, pred_len, model, vocab_size, ctx, idx_to_char,
                    char_to_idx))
```

Train the model using the same hyper-parameters as in the previous experiments.

```
In [9]: num_epochs, batch_size, lr, clipping_theta = 200, 32, 1e2, 1e-2
        pred_period, pred_len, prefixes = 50, 50, ['traveller', 'time traveller']
        train_and_predict_rnn_gluon(model, num_hiddens, vocab_size, ctx,
                                    corpus_indices, idx_to_char, char_to_idx,
                                    num_epochs, num_steps, lr, clipping_theta,
                                    batch_size, pred_period, pred_len, prefixes)

epoch 50, perplexity 4.198542, time 0.04 sec
 - traveller a mothere a mong the time traveller than spall of
 - time traveller than spall of the time traveller than spall of th
epoch 100, perplexity 2.004600, time 0.04 sec
 - traveller has g mathematical really a ceatee, the peopeered
 - time traveller helr it wiml be convenient for the ground for ita
epoch 150, perplexity 1.501630, time 0.04 sec
 - traveller held inthind to the others.  'you callon and time
 - time traveller, and thickness, and trace. but you difections of
epoch 200, perplexity 1.332802, time 0.04 sec
```

```
- traveller held in his hand we mevery oole might get one's g
- time traveller, with a slare really this is what is meant reare
```

Summary

- Gluon's `rnn` module provides an implementation at the recurrent neural network layer.

- Gluon's `nn.RNN` instance returns the output and hidden state after forward computation. This forward computation does not involve output layer computation.

Exercises

- Compare the implementation with the previous section. Does Gluon's implementation run faster? If you observe a significant difference, try to find the reason.

Scan the QR Code to Discuss

7.7 Back-propagation Through Time

If you have done the exercise in the previous section, you will know that the model cannot be trained normally if you do not clip the gradient. To provide a better understanding of this issue, this section will introduce the gradient computation and storage method used in recurrent neural networks, namely, back-propagation through time.

In the Forward Propagation, Back Propagation, and Computational Graphs section, we discussed the general ideas behind gradient computation and storage in neural networks and emphasized the interdependence of forward propagation and back propagation. Forward propagation in a recurrent neural network is relatively straightforward. Back-propagation through time is actually a specific application of back propagation in recurrent neural networks. It requires us to expand the recurrent neural network by time step to obtain the dependencies between model variables and parameters. Then, based on the chain rule, we apply back propagation to compute and store gradients.

7.7.1 Define the Model

To keep things simple, we consider an unbiased recurrent neural network, with the activation function set to identity mapping ($\phi(x) = x$). We set the input of the time step t to a single example $\boldsymbol{x}_t \in \mathbb{R}^d$ and use the label y_t, so the calculation expression for the hidden state $\boldsymbol{h}_t \in \mathbb{R}^h$ is:

$$\boldsymbol{h}_t = \boldsymbol{W}_{hx}\boldsymbol{x}_t + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1},$$

Here, $\boldsymbol{W}_{hx} \in \mathbb{R}^{h \times d}$ and $\boldsymbol{W}_{hh} \in \mathbb{R}^{h \times h}$ are the weight parameters of the hidden layer. Assuming the output layer weight parameter is $\boldsymbol{W}_{qh} \in \mathbb{R}^{q \times h}$, the output layer variable $\boldsymbol{o}_t \in \mathbb{R}^q$ for time step t can be calculated as follows:

$$\boldsymbol{o}_t = \boldsymbol{W}_{qh}\boldsymbol{h}_t.$$

Let the loss at time step t be defined as $\ell(\boldsymbol{o}_t, y_t)$. Thus, the loss function L for T time steps is defined as:

$$L = \frac{1}{T}\sum_{t=1}^{T}\ell(\boldsymbol{o}_t, y_t).$$

In what follows, we will refer to L as the objective function for the data instance of a given time step.

7.7.2 Model Computational Graph

In order to visualize the dependencies between model variables and parameters during computation in a recurrent neural network, we can draw a computational graph for the model, as shown in Figure 6.3. For example, the computation of the hidden states of time step 3 $\boldsymbol{h}_3$ depends on the model parameters $\boldsymbol{W}_{hx} and \boldsymbol{W}_{hh}$, the hidden state of the last time step $\boldsymbol{h}_2$, and the input of the current time step $\boldsymbol{x}_3$.
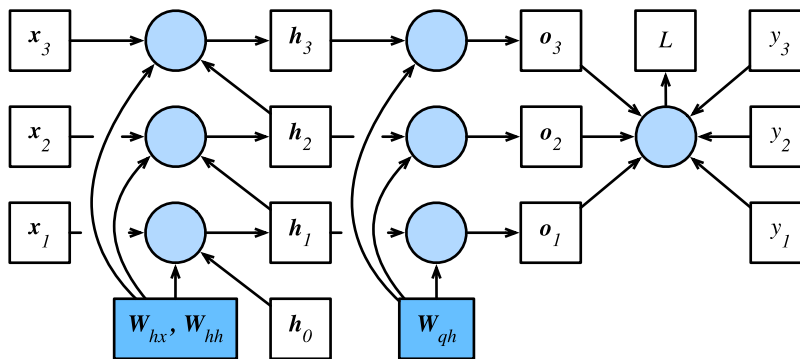

Fig. 7.5: Computational dependencies for a recurrent neural network model with three time steps. Boxes represent variables (not shaded) or parameters (shaded) and circles represent operators.

7.7.3 Back-propagation Through Time

As just mentioned, the model parameters in Figure 6.3 are $\boldsymbol{W}_{hx}$, $\boldsymbol{W}_{hh}$, and $\boldsymbol{W}_{qh}$. Similar to the Forward Propagation, Back Propagation, and Computational Graphs section, model training generally requires the model parameter gradients $\partial L/\partial \boldsymbol{W}_{hx}$, $\partial L/\partial \boldsymbol{W}_{hh}$, and $\partial L/\partial \boldsymbol{W}_{qh}$. According to the dependencies shown in Figure 6.3, we can calculate and store the gradients in turn going in the opposite direction of the arrows in the figure. To simplify the explanation, we continue to use the chain rule operator prod from the Forward Propagation, Back Propagation, and Computational Graphs section.

First, calculate the output layer variable gradient of the objective function with respect to each time step $\partial L/\partial \boldsymbol{o}_t \in \mathbb{R}^q$ using the following formula:

$$\frac{\partial L}{\partial \boldsymbol{o}_t} = \frac{\partial \ell(\boldsymbol{o}_t, y_t)}{T \cdot \partial \boldsymbol{o}_t}.$$

Now, we can calculate the gradient for the objective function model parameter $\boldsymbol{W}_{qh}$: $\partial L/\partial \boldsymbol{W}_{qh} \in \mathbb{R}^{q \times h}$. Based on Figure 6.3, L depends on $\boldsymbol{W}_{qh}$ through $\boldsymbol{o}_1, \ldots, \boldsymbol{o}_T$. Applying the chain rule,

$$\frac{\partial L}{\partial \boldsymbol{W}_{qh}} = \sum_{t=1}^{T} \text{prod}\left(\frac{\partial L}{\partial \boldsymbol{o}_t}, \frac{\partial \boldsymbol{o}_t}{\partial \boldsymbol{W}_{qh}}\right) = \sum_{t=1}^{T} \frac{\partial L}{\partial \boldsymbol{o}_t} \boldsymbol{h}_t^\top.$$

Next, we must note that there are also dependencies between hidden states. In Figure 6.3, L depends on the hidden state $\boldsymbol{h}_T$ of the final time step T only through $\boldsymbol{o}_T$. Therefore, we first calculate the objective function gradient with respect to the hidden state of the final time step: $\partial L/\partial \boldsymbol{h}_T \in \mathbb{R}^h$. According to the chain rule, we get

$$\frac{\partial L}{\partial \boldsymbol{h}_T} = \text{prod}\left(\frac{\partial L}{\partial \boldsymbol{o}_T}, \frac{\partial \boldsymbol{o}_T}{\partial \boldsymbol{h}_T}\right) = \boldsymbol{W}_{qh}^\top \frac{\partial L}{\partial \boldsymbol{o}_T}.$$

Then, for the time steps $t < T$, L depends on $\boldsymbol{h}_t$ through $\boldsymbol{h}_{t+1}$ and $\boldsymbol{o}_t$. Applying the chain rule, the objective function gradient with respect to the hidden states of the time steps $t < T$ ($\partial L/\partial \boldsymbol{h}_t \in \mathbb{R}^h$) must be calculated for each time step in turn from large to small:

$$\frac{\partial L}{\partial \boldsymbol{h}_t} = \text{prod}\left(\frac{\partial L}{\partial \boldsymbol{h}_{t+1}}, \frac{\partial \boldsymbol{h}_{t+1}}{\partial \boldsymbol{h}_t}\right) + \text{prod}\left(\frac{\partial L}{\partial \boldsymbol{o}_t}, \frac{\partial \boldsymbol{o}_t}{\partial \boldsymbol{h}_t}\right) = \boldsymbol{W}_{hh}^\top \frac{\partial L}{\partial \boldsymbol{h}_{t+1}} + \boldsymbol{W}_{qh}^\top \frac{\partial L}{\partial \boldsymbol{o}_t}.$$

Expanding the recursive formula above, we can find a general formula for the objective function hidden state gradient for any time step $1 \le t \le T$.

$$\frac{\partial L}{\partial \boldsymbol{h}_t} = \sum_{i=t}^{T} \left(\boldsymbol{W}_{hh}^\top\right)^{T-i} \boldsymbol{W}_{qh}^\top \frac{\partial L}{\partial \boldsymbol{o}_{T+t-i}}.$$

From the exponential term in the above formula, we can see that, when the number of time steps T is large or the time step t is small, the hidden state gradient of the objective function is prone to vanishing and explosion. This will also affect other gradients that contain the term $\partial L/\partial \boldsymbol{h}_t$, such as the gradients of model parameters in the hidden layer $\partial L/\partial \boldsymbol{W}_{hx} \in \mathbb{R}^{h \times d}$ and $\partial L/\partial \boldsymbol{W}_{hh} \in \mathbb{R}^{h \times h}$. In Figure 6.3, L

depends on these model parameters through $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_T$. According to the chain rule, we get

$$
\frac{\partial L}{\partial \boldsymbol{W}_{hx}} = \sum_{t=1}^{T} \text{prod}\left(\frac{\partial L}{\partial \boldsymbol{h}_t}, \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{W}_{hx}}\right) = \sum_{t=1}^{T} \frac{\partial L}{\partial \boldsymbol{h}_t} \boldsymbol{x}_t^{\top},
$$

$$
\frac{\partial L}{\partial \boldsymbol{W}_{hh}} = \sum_{t=1}^{T} \text{prod}\left(\frac{\partial L}{\partial \boldsymbol{h}_t}, \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{W}_{hh}}\right) = \sum_{t=1}^{T} \frac{\partial L}{\partial \boldsymbol{h}_t} \boldsymbol{h}_{t-1}^{\top}.
$$

As we already explained in the Forward Propagation, Back Propagation, and Computational Graphs section, after we calculate the above gradients in turn for each iteration, we save them to avoid the need for repeat calculation. For example, after calculating and storing the hidden state gradient $\partial L / \partial \boldsymbol{h}_t$, subsequent calculations of the model parameter gradients $\partial L / \partial \boldsymbol{W}_{hx}$ and $\partial L / \partial \boldsymbol{W}_{hh}$ can directly read the value of $\partial L / \partial \boldsymbol{h}_t$, so they do not need to be re-calculated. In addition, gradient calculation in back propagation may depend on the current values of variables. These are calculated using forward propagation. To give an example, the calculation of the parameter gradient $\partial L / \partial \boldsymbol{W}_{hh}$ must depend on the current hidden state value at the time step $t = 0, \ldots, T - 1$: $\boldsymbol{h}_t$ ($\boldsymbol{h}_0$ is obtained during initialization). These values are obtained by calculation and storage via forward propagation from the input layer to the output layer.

Summary

- Back-propagation through time is a specific application of back propagation in recurrent neural networks.

- When the number of time steps is large or the time step is small, the gradients in recurrent neural networks are prone to vanishing or explosion.

Exercises

- Besides gradient clipping, can you think of any other methods to cope with gradient explosion in recurrent neural networks?

Scan the QR Code to Discuss

7.8 Gated Recurrent Unit (GRU)

@TODO(smolix/astonzhang): the data set was just changed from lyrics to time machine, so descriptions/hyperparameters have to change.

In the previous section, we discussed gradient calculation methods in recurrent neural networks. We found that, when the number of time steps is large or the time step is small, the gradients in recurrent neural networks are prone to vanishing or explosion. Although gradient clipping can cope with gradient explosion, it cannot solve the vanishing gradient problem. Therefore, it is generally quite difficult to capture dependencies for time series with large time step distances during the actual use of recurrent neural networks.

Gated recurrent neural networks were proposed as a way to better capture dependencies for time series with large time step distances. Such a network uses learnable gates to control the flow of information. One common type of gated recurrent neural network is a gated recurrent unit (GRU) [1, 2]. Another common type of gated recurrent neural network is discussed in the next section.

7.8.1 Gated Recurrent Units

In what follows, we will discuss the design of GRUs. These networks introduce the reset gate and update gate concepts to change the method used to calculate hidden states in recurrent neural networks.

Reset Gates and Update Gates

As shown in Figure 6.4, the inputs for both reset gates and update gates in GRU are the current time step input X_t and the hidden state of the previous time step H_{t-1}. The output is computed by the fully connected layer with a sigmoid function as its activation function.
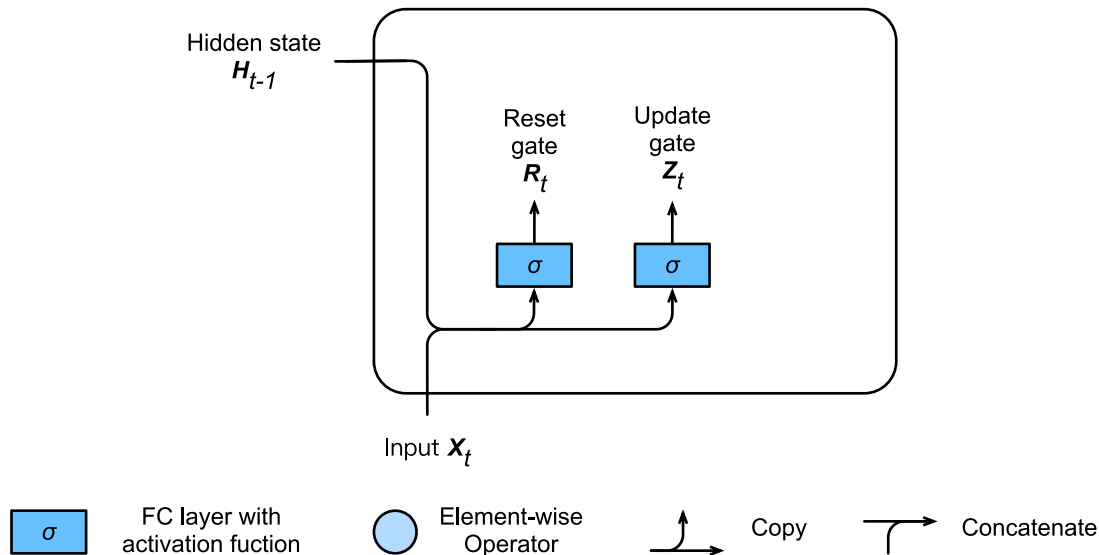
Fig. 7.6: Reset and update gate computation in a GRU.

Here, we assume there are h hidden units and, for a given time step t, the mini-batch input is $\boldsymbol{X}_t \in \mathbb{R}^{n \times d}$ (number of examples: n, number of inputs: d and the hidden state of the last time step is $\boldsymbol{H}_{t-1} \in \mathbb{R}^{n \times h}$. Then, the reset gate $\boldsymbol{R}_t \in \mathbb{R}^{n \times h}$ and update gate $\boldsymbol{Z}_t \in \mathbb{R}^{n \times h}$ computation is as follows:

$$\boldsymbol{R}_t = \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xr} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{hr} + \boldsymbol{b}_r),$$
$$\boldsymbol{Z}_t = \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xz} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{hz} + \boldsymbol{b}_z),$$

Here, $\boldsymbol{W}_{xr}, \boldsymbol{W}_{xz} \in \mathbb{R}^{d \times h}$ and $\boldsymbol{W}_{hr}, \boldsymbol{W}_{hz} \in \mathbb{R}^{h \times h}$ are weight parameters and $\boldsymbol{b}_r, \boldsymbol{b}_z \in \mathbb{R}^{1 \times h}$ is a bias parameter. As described in the Multilayer Perceptron section, a sigmoid function can transform element values between 0 and 1. Therefore, the range of each element in the reset gate $\boldsymbol{R}_t$ and update gate $\boldsymbol{Z}_t$ is $[0, 1]$.

Candidate Hidden States

Next, the GRU computes candidate hidden states to facilitate subsequent hidden state computation. As shown in Figure 6.5, we perform multiplication by element between the current time step reset gate output and previous time step hidden state (symbol: $\odot$). If the element value in the reset gate approaches 0, this means that it resets the value of the corresponding hidden state element to 0, discarding the hidden state from the previous time step. If the element value approaches 1, this indicates that the hidden state from the previous time step is retained. Then, the result of multiplication by element is concatenated with the current time step input to compute candidate hidden states in a fully connected layer with a tanh activation function. The range of all element values is $[-1, 1]$.
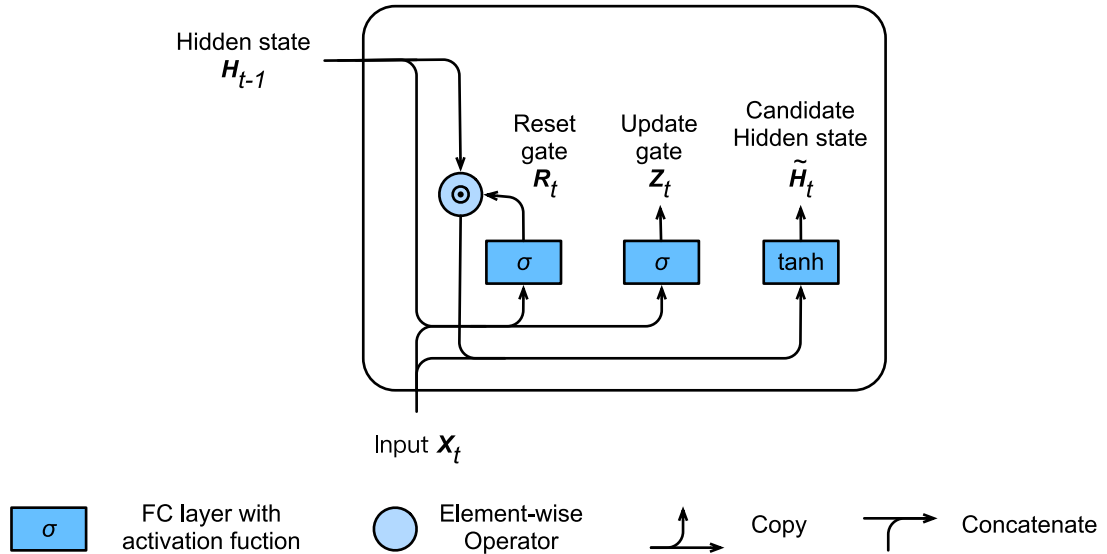
Fig. 7.7: Candidate hidden state computation in a GRU. Here, the multiplication sign indicates multiplication by element.

For time step t, the candidate hidden state $\tilde{\boldsymbol{H}}_t \in \mathbb{R}^{n \times h}$ is computed by the following formula:

$$\tilde{\boldsymbol{H}}_t = \tanh(\boldsymbol{X}_t \boldsymbol{W}_{xh} + (\boldsymbol{R}_t \odot \boldsymbol{H}_{t-1}) \boldsymbol{W}_{hh} + \boldsymbol{b}_h),$$

Here, $\boldsymbol{W}_{xh} \in \mathbb{R}^{d \times h}$ and $\boldsymbol{W}_{hh} \in \mathbb{R}^{h \times h}$ are weight parameters and $\boldsymbol{b}_h \in \mathbb{R}^{1 \times h}$ is a bias parameter. From the formula above, we can see that the reset gate controls how the hidden state of the previous time step enters into the candidate hidden state of the current time step. In addition, the hidden state of the previous time step may contain all historical information of the time series up to the previous time step. Thus, the reset gate can be used to discard historical information that has no bearing on predictions.

Hidden States

Finally, the computation of the hidden state $\boldsymbol{H}_t \in \mathbb{R}^{n \times h}$ for time step t uses the current time step's update gate $\boldsymbol{Z}_t$ to combine the previous time step hidden state $\boldsymbol{H}_{t-1}$ and current time step candidate hidden state $\tilde{\boldsymbol{H}}_t$:

$$\boldsymbol{H}_t = \boldsymbol{Z}_t \odot \boldsymbol{H}_{t-1} + (1 - \boldsymbol{Z}_t) \odot \tilde{\boldsymbol{H}}_t.$$
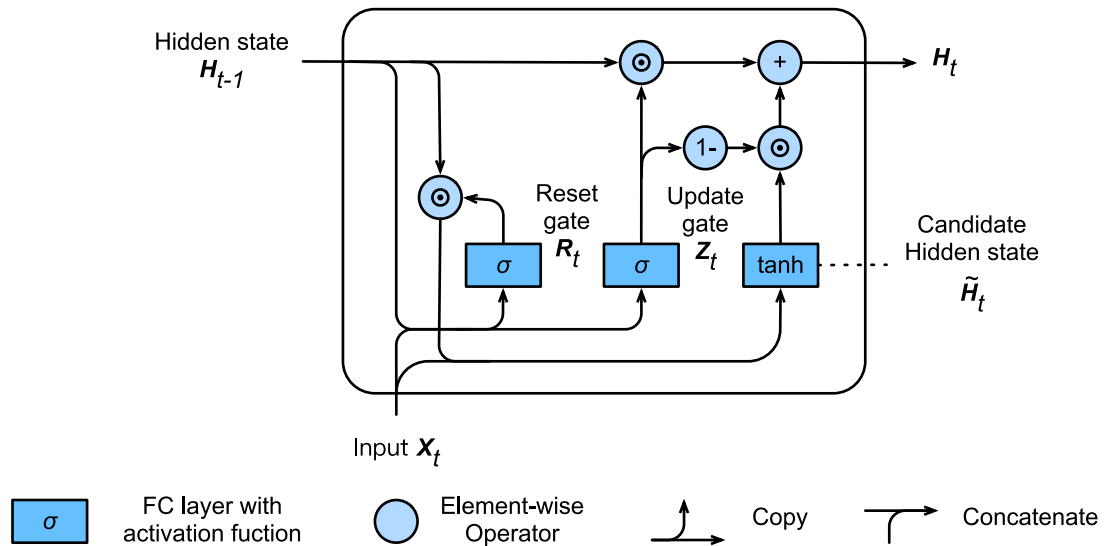
Fig. 7.8: Hidden state computation in a GRU. Here, the multiplication sign indicates multiplication by element.

It should be noted that update gates can control how hidden states should be updated by candidate hidden states containing current time step information, as shown in Figure 6.6. Here, we assume that the update gate is always approximately 1 between the time steps t' and t ($t' < t$). Therefore, the input information between the time steps t' and t almost never enters the hidden state $\boldsymbol{H}_t$ for time step t. In fact, we can think of it like this: The hidden state of an earlier time $\boldsymbol{H}_{t'-1}$ is saved over time and passed to the current time step t. This design can cope with the vanishing gradient problem in recurrent neural networks and better capture dependencies for time series with large time step distances.

We can summarize the design of GRUs as follows:

- Reset gates help capture short-term dependencies in time series.

- Update gates help capture long-term dependencies in time series.

7.8.2 Read the Data Set

To implement and display a GRU, we will again use the Jay Chou lyrics data set to train the model to compose song lyrics. The implementation, except for the GRU, has already been described in the *Recurrent Neural Network* section. The code for reading the data set is given below:

```
In [1]: import sys
        sys.path.insert(0, '..')

        import d2l
        from mxnet import nd
```

```
from mxnet.gluon import rnn

(corpus_indices, char_to_idx, idx_to_char,
 vocab_size) = d2l.load_data_time_machine()
```

7.8.3 Implementation from Scratch

We will start by showing how to implement a GRU from scratch.

Initialize Model Parameters

The code below initializes the model parameters. The hyper-parameter `num_hiddens` defines the number of hidden units.

```
In [2]: num_inputs, num_hiddens, num_outputs = vocab_size, 256, vocab_size
        ctx = d2l.try_gpu()

        def get_params():
            def _one(shape):
                return nd.random.normal(scale=0.01, shape=shape, ctx=ctx)

            def _three():
                return (_one((num_inputs, num_hiddens)),
                        _one((num_hiddens, num_hiddens)),
                        nd.zeros(num_hiddens, ctx=ctx))

            W_xz, W_hz, b_z = _three()  # Update gate parameter
            W_xr, W_hr, b_r = _three()  # Reset gate parameter
            W_xh, W_hh, b_h = _three()  # Candidate hidden state parameter
            # Output layer parameters
            W_hq = _one((num_hiddens, num_outputs))
            b_q = nd.zeros(num_outputs, ctx=ctx)
            # Create gradient
            params = [W_xz, W_hz, b_z, W_xr, W_hr, b_r, W_xh, W_hh, b_h, W_hq, b_q]
            for param in params:
                param.attach_grad()
            return params
```

Define the Model

Now we will define the hidden state initialization function `init_gru_state`. Just like the `init_rnn_state` function defined in the *Implementation of the Recurrent Neural Network from Scratch* section, this function returns a tuple composed of an NDArray with a shape (batch size, number of hidden units) value of 0.

```
In [3]: def init_gru_state(batch_size, num_hiddens, ctx):
            return (nd.zeros(shape=(batch_size, num_hiddens), ctx=ctx), )
```

Below, we define the model based on GRU computing expressions.

```
In [4]: def gru(inputs, state, params):
            W_xz, W_hz, b_z, W_xr, W_hr, b_r, W_xh, W_hh, b_h, W_hq, b_q = params
            H, = state
            outputs = []
            for X in inputs:
                Z = nd.sigmoid(nd.dot(X, W_xz) + nd.dot(H, W_hz) + b_z)
                R = nd.sigmoid(nd.dot(X, W_xr) + nd.dot(H, W_hr) + b_r)
                H_tilda = nd.tanh(nd.dot(X, W_xh) + nd.dot(R * H, W_hh) + b_h)
                H = Z * H + (1 - Z) * H_tilda
                Y = nd.dot(H, W_hq) + b_q
                outputs.append(Y)
            return outputs, (H,)
```

Train the Model and Write Lyrics

During model training, we only use adjacent examples. After setting the hyper-parameters, we train and model and create a 50 character string of lyrics based on the prefixes separate and not separated.

```
In [5]: num_epochs, num_steps, batch_size, lr, clipping_theta = 160, 35, 32, 1e2, 1e-2
        pred_period, pred_len, prefixes = 40, 50, ['traveller', 'time traveller']
```

We create a string of lyrics based on the currently trained model every 40 epochs.

```
In [6]: d2l.train_and_predict_rnn(gru, get_params, init_gru_state, num_hiddens,
                                  vocab_size, ctx, corpus_indices, idx_to_char,
                                  char_to_idx, False, num_epochs, num_steps, lr,
                                  clipping_theta, batch_size, pred_period, pred_len,
                                  prefixes)

epoch 40, perplexity 7.513687, time 0.67 sec
 - traveller the the the the the the the the the the the t
 - time traveller the the the the the the the the the the the the t
epoch 80, perplexity 3.357612, time 0.67 sec
 - traveller the time traveller the move the very young mon th
 - time traveller the move the very young mon the time traveller th
epoch 120, perplexity 1.403060, time 0.67 sec
 - traveller, as we have no travel through a noterestent the f
 - time traveller, we have no means of staying br, for this bust ba
epoch 160, perplexity 1.146948, time 0.67 sec
 - traveller has gone wrong. we are always getting away from t
 - time traveller held in his hand difis along this fourth dimensio
```

7.8.4 Concise Implementation

In Gluon, we can directly call the GRU class in the rnn module.

```
In [7]: gru_layer = rnn.GRU(num_hiddens)
        model = d2l.RNNModel(gru_layer, vocab_size)
        d2l.train_and_predict_rnn_gluon(model, num_hiddens, vocab_size, ctx,
                                        corpus_indices, idx_to_char, char_to_idx,
                                        num_epochs, num_steps, lr, clipping_theta,
                                        batch_size, pred_period, pred_len, prefixes)
```

```
epoch 40, perplexity 7.722676, time 0.05 sec
 - traveller and the the the the the the the the the the t
 - time traveller the the the the the the the the the the the t
epoch 80, perplexity 3.593244, time 0.05 sec
 - traveller the time traveller.  'you can a cunt ment all rea
 - time traveller.  'the time traveller.  'you can a cunt ment all
epoch 120, perplexity 1.547183, time 0.05 sec
 - traveller hasmithere. can a cube than suldist.'  'not is al
 - time traveller.  'you can show bloce that roples this four comen
epoch 160, perplexity 1.159431, time 0.05 sec
 - traveller.  'you can show black is white by argument,' saif
 - time traveller, after the pause required for the proper assimila
```

Summary

- Gated recurrent neural networks can better capture dependencies for time series with large time step distances.

- GRUs introduce the reset gate and update gate concepts to change the method used to calculate hidden states in recurrent neural networks. They include reset gates, update gates, candidate hidden states, and hidden states.

- Reset gates help capture short-term dependencies in time series.

- Update gates help capture long-term dependencies in time series.

Exercises

- Assume that time step $t' < t$. If we only want to use the input for time step t' to predict the output at time step t, what are the best values for the reset and update gates for each time step?

- Adjust the hyper-parameters and observe and analyze the impact on running time, perplexity, and the written lyrics.

- Compare the running times of a GRU and ungated recurrent neural network under the same conditions.

References

[1] Cho, K., Van Merriënboer, B., Bahdanau, D., & Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. arXiv preprint arXiv:1409.1259.

[2] Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. arXiv preprint arXiv:1412.3555.

7.9 Long Short-term Memory (LSTM)

@TODO(smolix/astonzhang): the data set was just changed from lyrics to time machine, so descriptions/hyperparameters have to change.

This section describes another commonly used gated recurrent neural network: long short-term memory (LSTM) [1]. Its structure is slightly more complicated than that of a gated recurrent unit.

7.9.1 Long Short-term Memory

Three gates are introduced in LSTM: the input gate, the forget gate, and the output gate, as well as memory cells in the same shape as the hidden state (some literature treats memory cells as a special kind of hidden state) used to record additional information.

Input Gates, Forget Gates, and Output Gates

Like the reset gate and the update gate in the gated recurrent unit, as shown in Figure 6.7, the input of LSTM gates is the current time step input X_t and the hidden state of the previous time step H_{t-1}. The output is computed by the fully connected layer with a sigmoid function as its activation function. As a result, the three gate elements all have a value range of $[0, 1]$.
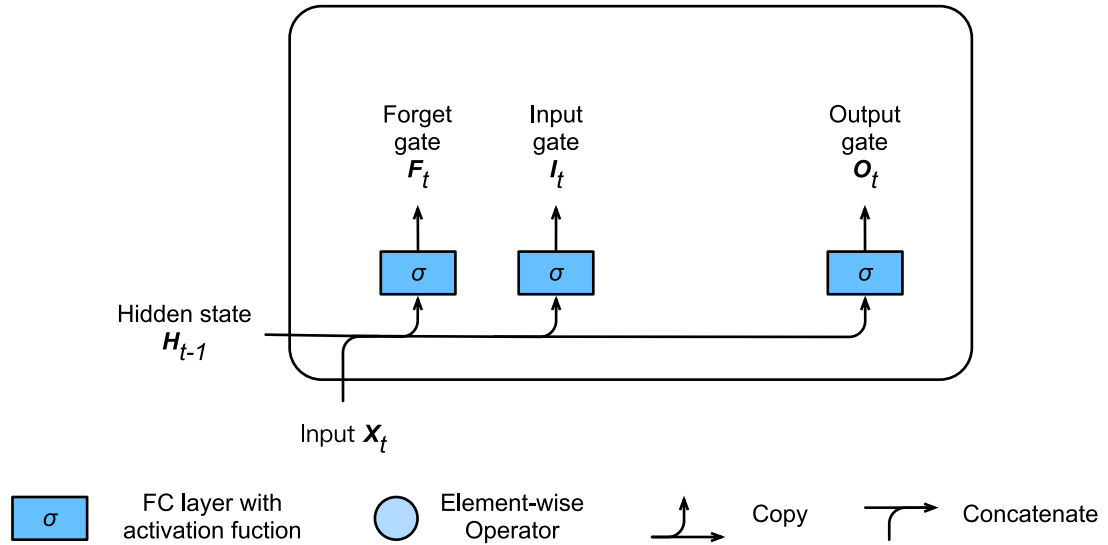
Fig. 7.9: Calculation of input, forget, and output gates in LSTM.

Here, we assume there are h hidden units and, for a given time step t, the mini-batch input is $\boldsymbol{X}_t \in \mathbb{R}^{n \times d}$ (number of examples: n, number of inputs: d and the hidden state of the last time step is $\boldsymbol{H}_{t-1} \in \mathbb{R}^{n \times h}$. For time step t, the input gate $\boldsymbol{I}_t \in \mathbb{R}^{n \times h}$, forget gate $\boldsymbol{F}_t \in \mathbb{R}^{n \times h}$, and output gate $\boldsymbol{O}_t \in \mathbb{R}^{n \times h}$ are calculated as follows:

$$\boldsymbol{I}_t = \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xi} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{hi} + \boldsymbol{b}_i),$$
$$\boldsymbol{F}_t = \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xf} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{hf} + \boldsymbol{b}_f),$$
$$\boldsymbol{O}_t = \sigma(\boldsymbol{X}_t \boldsymbol{W}_{xo} + \boldsymbol{H}_{t-1} \boldsymbol{W}_{ho} + \boldsymbol{b}_o),$$

Here, $\boldsymbol{W}_{xi}, \boldsymbol{W}_{xf}, \boldsymbol{W}_{xo} \in \mathbb{R}^{d \times h}$ and $\boldsymbol{W}_{hi}, \boldsymbol{W}_{hf}, \boldsymbol{W}_{ho} \in \mathbb{R}^{h \times h}$ are weight parameters and $\boldsymbol{b}_i, \boldsymbol{b}_f, \boldsymbol{b}_o \in \mathbb{R}^{1 \times h}$ is a bias parameter.

Candidate Memory Cells

Next, LSTM needs to compute the candidate memory cell $\tilde{\boldsymbol{C}}_t$. Its computation is similar to the three gates described above, but using a tanh function with a value range for $[-1, 1]$ as activation function, as shown in Figure 6.8.
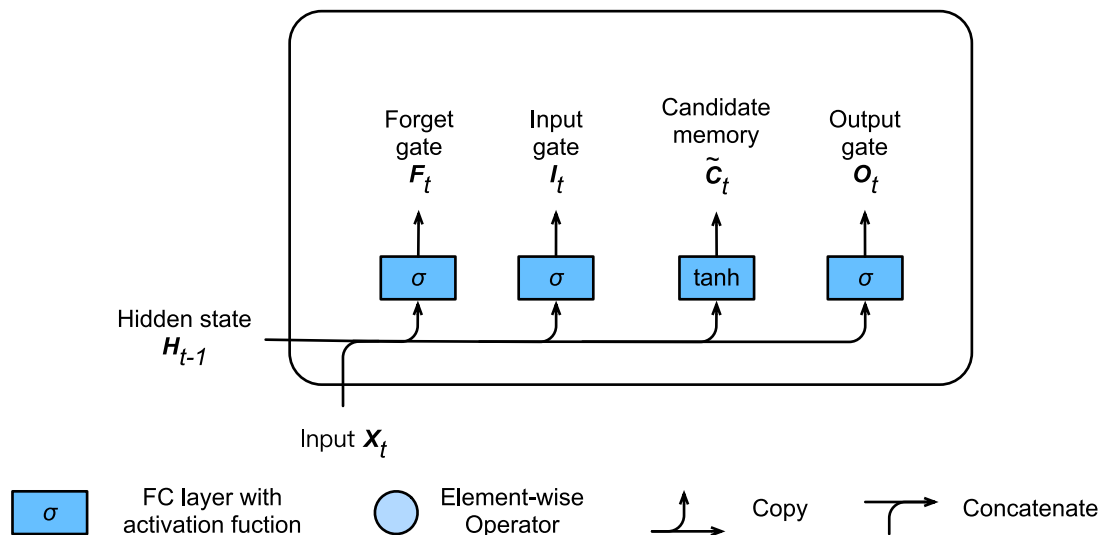
Fig. 7.10: Computation of candidate memory cells in LSTM.

For time step t, the candidate memory cell $\tilde{C}_t \in \mathbb{R}^{n \times h}$ is calculated by the following formula:

$$\tilde{C}_t = \tanh(X_t W_{xc} + H_{t-1} W_{hc} + b_c),$$

Here, $W_{xc} \in \mathbb{R}^{d \times h}$ and $W_{hc} \in \mathbb{R}^{h \times h}$ are weight parameters and $b_c \in \mathbb{R}^{1 \times h}$ is a bias parameter.

Memory Cells

We can control flow of information in the hidden state use input, forget, and output gates with an element value range between $[0, 1]$. This is also generally achieved by using multiplication by element (symbol $\odot$). The computation of the current time step memory cell $C_t \in \mathbb{R}^{n \times h}$ combines the information of the previous time step memory cells and the current time step candidate memory cells, and controls the flow of information through forget gate and input gate:

$$C_t = F_t \odot C_{t-1} + I_t \odot \tilde{C}_t.$$

As shown in Figure 6.9, the forget gate controls whether the information in the memory cell C_{t-1} of the last time step is passed to the current time step, and the input gate can control how the input of the current time step X_t flows into the memory cells of the current time step through the candidate memory cell $\tilde{C}_t$. If the forget gate is always approximately 1 and the input gate is always approximately 0, the past memory cells will be saved over time and passed to the current time step. This design can cope with the vanishing gradient problem in recurrent neural networks and better capture dependencies for time series with large time step distances.
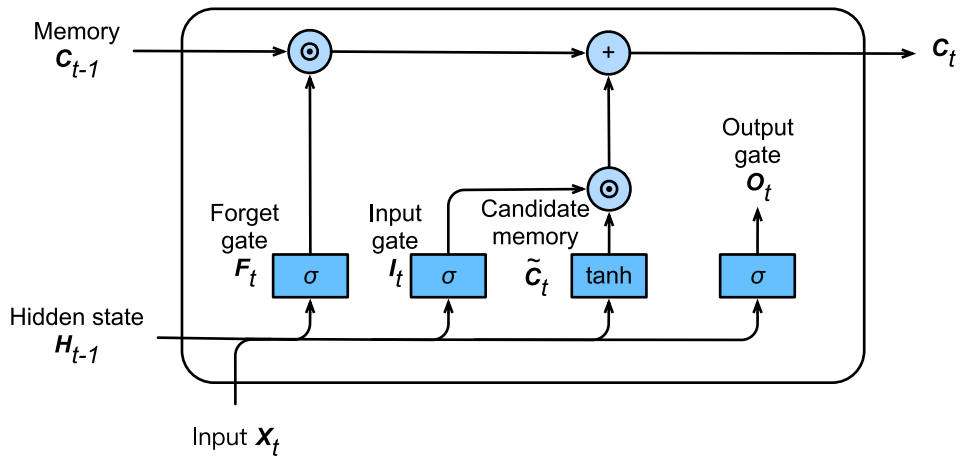
Fig. 7.11: Computation of memory cells in LSTM. Here, the multiplication sign indicates multiplication by element.

Hidden States

With memory cells, we can also control the flow of information from memory cells to the hidden state $H_t \in \mathbb{R}^{n \times h}$ through the output gate:

$$H_t = O_t \odot \tanh(C_t).$$

The tanh function here ensures that the hidden state element value is between -1 and 1. It should be noted that when the output gate is approximately 1, the memory cell information will be passed to the hidden state for use by the output layer; and when the output gate is approximately 0, the memory cell information is only retained by itself. Figure 6.10 shows the computation of the hidden state in LSTM.
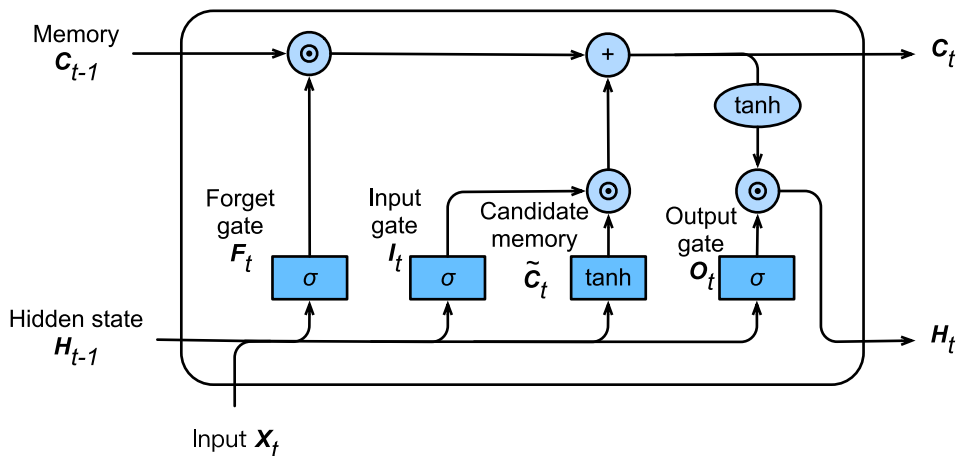
Fig. 7.12: Computation of hidden state in LSTM. Here, the multiplication sign indicates multiplication by element.

7.9.2 Read the Data Set

Below we begin to implement and display LSTM. As with the experiments in the previous sections, we still use the lyrics of the Jay Chou data set to train the model to write lyrics.

```
In [1]: import sys
        sys.path.insert(0, '..')

        import d2l
        from mxnet import nd
        from mxnet.gluon import rnn

        (corpus_indices, char_to_idx, idx_to_char,
         vocab_size) = d2l.load_data_time_machine()
```

7.9.3 Implementation from Scratch

First, we will discuss how to implement LSTM from scratch.

Initialize Model Parameters

The code below initializes the model parameters. The hyper-parameter `num_hiddens` defines the number of hidden units.

```
In [2]: num_inputs, num_hiddens, num_outputs = vocab_size, 256, vocab_size
        ctx = d2l.try_gpu()

        def get_params():
            def _one(shape):
                return nd.random.normal(scale=0.01, shape=shape, ctx=ctx)

            def _three():
                return (_one((num_inputs, num_hiddens)),
                        _one((num_hiddens, num_hiddens)),
                        nd.zeros(num_hiddens, ctx=ctx))

            W_xi, W_hi, b_i = _three()  # Input gate parameters
            W_xf, W_hf, b_f = _three()  # Forget gate parameters
            W_xo, W_ho, b_o = _three()  # Output gate parameters
            W_xc, W_hc, b_c = _three()  # Candidate cell parameters
            # Output layer parameters
            W_hq = _one((num_hiddens, num_outputs))
            b_q = nd.zeros(num_outputs, ctx=ctx)
            # Create gradient
            params = [W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc,
                      b_c, W_hq, b_q]
            for param in params:
                param.attach_grad()
            return params
```

7.9.4 Define the Model

In the initialization function, the hidden state of the LSTM needs to return an additional memory cell with a value of 0 and a shape of (batch size, number of hidden units).

```
In [3]: def init_lstm_state(batch_size, num_hiddens, ctx):
            return (nd.zeros(shape=(batch_size, num_hiddens), ctx=ctx),
                    nd.zeros(shape=(batch_size, num_hiddens), ctx=ctx))
```

Below, we defined the model based on LSTM computing expressions. It should be noted that only the hidden state will be passed into the output layer, and the memory cells do not participate in the computation of the output layer.

```
In [4]: def lstm(inputs, state, params):
            [W_xi, W_hi, b_i, W_xf, W_hf, b_f, W_xo, W_ho, b_o, W_xc, W_hc, b_c,
             W_hq, b_q] = params
            (H, C) = state
            outputs = []
            for X in inputs:
                I = nd.sigmoid(nd.dot(X, W_xi) + nd.dot(H, W_hi) + b_i)
                F = nd.sigmoid(nd.dot(X, W_xf) + nd.dot(H, W_hf) + b_f)
                O = nd.sigmoid(nd.dot(X, W_xo) + nd.dot(H, W_ho) + b_o)
                C_tilda = nd.tanh(nd.dot(X, W_xc) + nd.dot(H, W_hc) + b_c)
                C = F * C + I * C_tilda
                H = O * C.tanh()
                Y = nd.dot(H, W_hq) + b_q
                outputs.append(Y)
            return outputs, (H, C)
```

Train the Model and Write Lyrics

As in the previous section, during model training, we only use adjacent sampling. After setting the hyper-parameters, we train and model and create a 50 character string of lyrics based on the prefixes separate and not separated.

```
In [5]: num_epochs, num_steps, batch_size, lr, clipping_theta = 160, 35, 32, 1e2, 1e-2
        pred_period, pred_len, prefixes = 40, 50, ['traveller', 'time traveller']
```

We create a string of lyrics based on the currently trained model every 40 epochs.

```
In [6]: d2l.train_and_predict_rnn(lstm, get_params, init_lstm_state, num_hiddens,
                                   vocab_size, ctx, corpus_indices, idx_to_char,
                                   char_to_idx, False, num_epochs, num_steps, lr,
                                   clipping_theta, batch_size, pred_period, pred_len,
                                   prefixes)
```

```
epoch 40, perplexity 7.846585, time 0.79 sec
 - traveller the the the the the the the the the the the the the t
 - time traveller the the the the the the the the the the the the t
epoch 80, perplexity 3.893780, time 0.80 sec
 - traveller for instance, have it all the traveller comension
 - time traveller of some the grom the time traveller of the back a
epoch 120, perplexity 1.976419, time 0.79 sec
 - traveller a flithed as a conection along the time traveller
 - time traveller smiled ffours direction of momer a move about in
epoch 160, perplexity 1.286961, time 0.79 sec
 - traveller. 'yealles. 'really this is what is meant by the f
 - time traveller. 'you can show black is white by argument,' said
```

7.9.5 Concise Implementation

In Gluon, we can directly call the LSTM class in the rnn module.

```
In [7]: lstm_layer = rnn.LSTM(num_hiddens)
        model = d2l.RNNModel(lstm_layer, vocab_size)
        d2l.train_and_predict_rnn_gluon(model, num_hiddens, vocab_size, ctx,
                                        corpus_indices, idx_to_char, char_to_idx,
                                        num_epochs, num_steps, lr, clipping_theta,
                                        batch_size, pred_period, pred_len, prefixes)
```

```
epoch 40, perplexity 8.472295, time 0.05 sec
 - traveller the the the the the the the the the the the the the t
 - time traveller the the the the the the the the the the the the t
epoch 80, perplexity 4.548608, time 0.05 sec
 - traveller cound do not here are are of the gromensions of t
 - time traveller cone of the dimensions of spaced the pery his lin
epoch 120, perplexity 2.342877, time 0.05 sec
 - traveller. 'it' said filby, 'why cankller-difce at realle
 - time traveller. 'it' said filby, 'why cankller-difce at realle
epoch 160, perplexity 1.429114, time 0.05 sec
 - traveller. 'but i have experimental verification,' cried i
 - time traveller, after the parsmucal and for sravely hard abouth
```

Summary

- The hidden layer output of LSTM includes hidden states and memory cells. Only hidden states are passed into the output layer.

- The input, forget, and output gates in LSTM can control the flow of information.

- LSTM can cope with the gradient attenuation problem in the recurrent neural networks and better capture dependencies for time series with large time step distances.

Exercises

- Adjust the hyper-parameters and observe and analyze the impact on running time, perplexity, and the written lyrics.

- Under the same conditions, compare the running time of an LSTM, GRU and recurrent neural network without gates.

- Since the candidate memory cells ensure that the value range is between -1 and 1 using the tanh function, why does the hidden state need to use the tanh function again to ensure that the output value range is between -1 and 1?

References

[1] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8), 1735-1780.

Scan the QR Code to Discuss

7.10 Deep Recurrent Neural Networks

Up to now, we have only discussed recurrent neural networks with a single unidirectional hidden layer. In deep learning applications, we generally use recurrent neural networks that contain multiple hidden layers. These are also called deep recurrent neural networks. Figure 6.11 demonstrates a deep recurrent neural network with L hidden layers. Each hidden state is continuously passed to the next time step of the current layer and the next layer of the current time step.
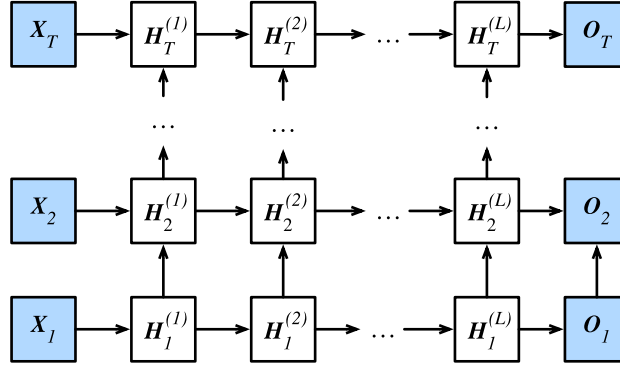
Fig. 7.13: Architecture of a deep recurrent neural network.

In time step t, we assume the mini-batch input is given as $\boldsymbol{X}_t \in \mathbb{R}^{n \times d}$ (number of examples: n, number of inputs: d). The hidden state of hidden layer ℓ ($\ell = 1, \ldots, T$) is $\boldsymbol{H}_t^{(\ell)} \in \mathbb{R}^{n \times h}$ (number of hidden units: h), the output layer variable is $\boldsymbol{O}_t \in \mathbb{R}^{n \times q}$ (number of outputs: q), and the hidden layer activation function is ϕ. The hidden state of hidden layer 1 is calculated in the same way as before:

$$\boldsymbol{H}_t^{(1)} = \phi(\boldsymbol{X}_t \boldsymbol{W}_{xh}^{(1)} + \boldsymbol{H}_{t-1}^{(1)} \boldsymbol{W}_{hh}^{(1)} + \boldsymbol{b}_h^{(1)}),$$

Here, the weight parameters $\boldsymbol{W}_{xh}^{(1)} \in \mathbb{R}^{d \times h} and \boldsymbol{W}_{hh}^{(1)} \in \mathbb{R}^{h \times h}$ and bias parameter $\boldsymbol{b}_h^{(1)} \in \mathbb{R}^{1 \times h}$ are the model parameters of hidden layer 1.

When $1 < \ell \leq L$, the hidden state of hidden layer ℓ is expressed as follows:

$$\boldsymbol{H}_t^{(\ell)} = \phi(\boldsymbol{H}_t^{(\ell-1)} \boldsymbol{W}_{xh}^{(\ell)} + \boldsymbol{H}_{t-1}^{(\ell)} \boldsymbol{W}_{hh}^{(\ell)} + \boldsymbol{b}_h^{(\ell)}),$$

Here, the weight parameters $\boldsymbol{W}_{xh}^{(\ell)} \in \mathbb{R}^{h \times h} and \boldsymbol{W}_{hh}^{(\ell)} \in \mathbb{R}^{h \times h}$ and bias parameter $\boldsymbol{b}_h^{(\ell)} \in \mathbb{R}^{1 \times h}$ are the model parameters of hidden layer ℓ.

Finally, the output of the output layer is only based on the hidden state of hidden layer L:

$$\boldsymbol{O}_t = \boldsymbol{H}_t^{(L)} \boldsymbol{W}_{hq} + \boldsymbol{b}_q,$$

Here, the weight parameter $\boldsymbol{W}_{hq} \in \mathbb{R}^{h \times q}$ and bias parameter $\boldsymbol{b}_q \in \mathbb{R}^{1 \times q}$ are the model parameters of the output layer.

Just as with multilayer perceptrons, the number of hidden layers L and number of hidden units h are hyper parameters. In addition, we can create a deep gated recurrent neural network by replacing hidden state computation with GRU or LSTM computation.

7. Recurrent Neural Networks

Summary

- In deep recurrent neural networks, hidden state information is continuously passed to the next time step of the current layer and the next layer of the current time step.

Exercises

- Alter the model in the *Implementation of a Recurrent Neural Network from Scratch* section to create a recurrent neural network with two hidden layers. Observe and analyze the experimental phenomena.

Scan the QR Code to Discuss

7.11 Bidirectional Recurrent Neural Networks

All the recurrent neural network models so far discussed have assumed that the current time step is determined by the series of earlier time steps. Therefore, they all pass information through hidden states in a forward direction. Sometimes, however, the current time step can be determined by later time steps. For example, when we write a statement, we may modify the words at the beginning of the statement based on the words at the end. Bidirectional recurrent neural networks add a hidden layer that passes information in a backward direction to more flexibly process such information. Figure 6.12 demonstrates the architecture of a bidirectional recurrent neural network with a single hidden layer.
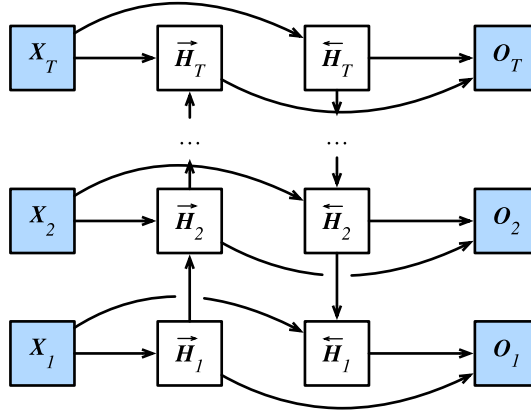
Fig. 7.14: Architecture of a bidirectional recurrent neural network.

Now, we will look at the specifics of such a network. For a given time step t, the mini-batch input is $\boldsymbol{X}_t \in \mathbb{R}^{n \times d}$ (number of examples: n, number of inputs: d) and the hidden layer activation function is ϕ. In the bidirectional architecture: We assume the forward hidden state for this time step is $\overrightarrow{\boldsymbol{H}}_t \in \mathbb{R}^{n \times h}$ (number of forward hidden units: h) and the backward hidden state is $\overleftarrow{\boldsymbol{H}}_t \in \mathbb{R}^{n \times h}$ (number of backward hidden units: h). Thus, we can compute the forward and backward hidden states:

$$\overrightarrow{\boldsymbol{H}}_t = \phi(\boldsymbol{X}_t \boldsymbol{W}_{xh}^{(f)} + \overrightarrow{\boldsymbol{H}}_{t-1} \boldsymbol{W}_{hh}^{(f)} + \boldsymbol{b}_h^{(f)}),$$
$$\overleftarrow{\boldsymbol{H}}_t = \phi(\boldsymbol{X}_t \boldsymbol{W}_{xh}^{(b)} + \overleftarrow{\boldsymbol{H}}_{t+1} \boldsymbol{W}_{hh}^{(b)} + \boldsymbol{b}_h^{(b)}),$$

Here, the weight parameters $\boldsymbol{W}_{xh}^{(f)} \in \mathbb{R}^{d \times h}, \boldsymbol{W}_{hh}^{(f)} \in \mathbb{R}^{h \times h}, \boldsymbol{W}_{xh}^{(b)} \in \mathbb{R}^{d \times h}, and \boldsymbol{W}_{hh}^{(b)} \in \mathbb{R}^{h \times h}$ and bias parameters $\boldsymbol{b}_h^{(f)} \in \mathbb{R}^{1 \times h} and \boldsymbol{b}_h^{(b)} \in \mathbb{R}^{1 \times h}$ are all model parameters.

Then we concatenate the forward and backward hidden states $\overrightarrow{\boldsymbol{H}}_t$ and $\overleftarrow{\boldsymbol{H}}_t$ to obtain the hidden state $\boldsymbol{H}_t \in \mathbb{R}^{n \times 2h}$ and input it to the output layer. The output layer computes the output $\boldsymbol{O}_t \in \mathbb{R}^{n \times q}$ (number of outputs: q):

$$\boldsymbol{O}_t = \boldsymbol{H}_t \boldsymbol{W}_{hq} + \boldsymbol{b}_q,$$

Here, the weight parameter $\boldsymbol{W}_{hq} \in \mathbb{R}^{2h \times q}$ and bias parameter $\boldsymbol{b}_q \in \mathbb{R}^{1 \times q}$ are the model parameters of the output layer. The two directions can have different numbers of hidden units.

Summary

- In bidirectional recurrent neural networks, the hidden state for each time step is simultaneously determined by the subseries before and after this time step (including the input for the current time step).

Exercises

- If the different directions use a different number of hidden units, how will the shape of H_t change?
- Referring to figures 6.11 and 6.12, design a bidirectional recurrent neural network with multiple hidden layers.

Scan the QR Code to Discuss

8

Optimization Algorithms

If you have read this book in order to this point, then you have already used optimization algorithms to train deep learning models. Specifically, when training models, we use optimization algorithms to continue updating the model parameters to reduce the value of the model loss function. When iteration ends, model training ends along with it. The model parameters we get here are the parameters that the model learned through training.

Optimization algorithms are important for deep learning. On the one hand, training a complex deep learning model can take hours, days, or even weeks. The performance of the optimization algorithm directly affects the model's training efficiency. On the other hand, understanding the principles of different optimization algorithms and the meanings of their hyperparameters will enable us to tune the hyperparameters in a targeted manner to improve the performance of deep learning models.

In this chapter, we explore common deep learning optimization algorithms in depth.

8.1 Optimization and Deep Learning

In this section, we will discuss the relationship between optimization and deep learning as well as the challenges of using optimization in deep learning. For a deep learning problem, we will usually define a loss function first. Once we have the loss function, we can use an optimization algorithm in attempt to minimize the loss. In optimization, a loss function is often referred to as the objective function of the optimization problem. Traditionally, optimization algorithms usually only consider minimizing the

objective function. In fact, any maximization problem can be easily transformed into a minimization problem: we just need to use the opposite of the objective function as the new objective function.

8.1.1 The Relationship Between Optimization and Deep Learning

Although optimization provides a way to minimize the loss function for deep learning, in essence, the goals of optimization and deep learning are different. In the Model Selection, Underfitting and Overfitting section, we discussed the difference between the training error and generalization error. Because the objective function of the optimization algorithm is usually a loss function based on the training data set, the goal of optimization is to reduce the training error. However, the goal of deep learning is to reduce the generalization error. In order to reduce the generalization error, we need to pay attention to overfitting in addition to using the optimization algorithm to reduce the training error.

In this chapter, we are going to focus specifically on the performance of the optimization algorithm in minimizing the objective function, rather than the model's generalization error.

8.1.2 Optimization Challenges in Deep Learning

In the Linear Regression section, we differentiated between analytical solutions and numerical solutions in optimization problems. In deep learning, most objective functions are complicated. Therefore, many optimization problems do not have analytical solutions. Instead, we must use optimization algorithms based on the numerical method to find approximate solutions, which also known as numerical solutions. The optimization algorithms discussed here are all numerical method-based algorithms. In order to minimize the numerical solution of the objective function, we will reduce the value of the loss function as much as possible by using optimization algorithms to finitely update the model parameters.

There are many challenges in deep learning optimization. Two such challenges are discussed below: local minimums and saddle points. To better describe the problem, we first import the packages or modules required for the experiments in this section.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mpl_toolkits import mplot3d
        import numpy as np
```
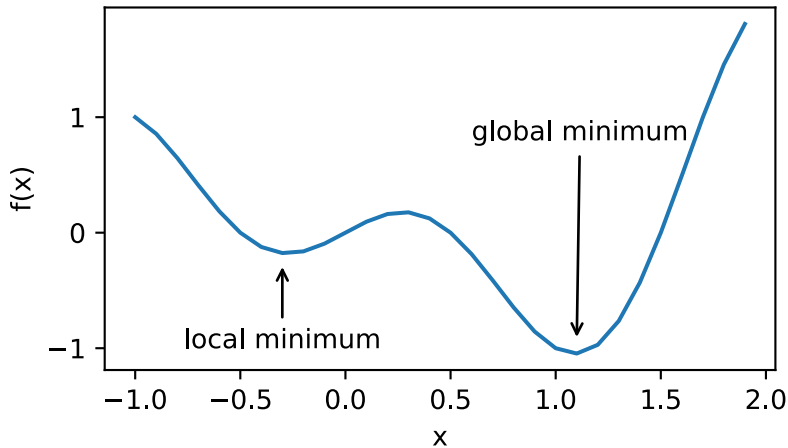
Local Minimums

For the objective function $f(x)$, if the value of $f(x)$ at x is smaller than the values of $f(x)$ at any other points in the vicinity of x, then $f(x)$ could be a local minimum. If the value of $f(x)$ at x is the minimum of the objective function over the entire domain, then $f(x)$ is the global minimum.

For example, given the function

$$f(x) = x \cdot \cos(\pi x), \qquad -1.0 \leq x \leq 2.0,$$

we can approximate the local minimum and global minimum of this function. Please note that the arrows in the figure only indicate the approximate positions.

```
In [2]: def f(x):
            return x * np.cos(np.pi * x)

        d2l.set_figsize((4.5, 2.5))
        x = np.arange(-1.0, 2.0, 0.1)
        fig,  = d2l.plt.plot(x, f(x))
        fig.axes.annotate('local minimum', xy=(-0.3, -0.25), xytext=(-0.77, -1.0),
                          arrowprops=dict(arrowstyle='->'))
        fig.axes.annotate('global minimum', xy=(1.1, -0.95), xytext=(0.6, 0.8),
                          arrowprops=dict(arrowstyle='->'))
        d2l.plt.xlabel('x')
        d2l.plt.ylabel('f(x)');
```


The objective function of the deep learning model may have several local optimums. When the numerical solution of an optimization problem is near the local optimum, the numerical solution obtained by the final iteration may only minimize the objective function locally, rather than globally, as the gradient of the objective function's solutions approaches or becomes zero.
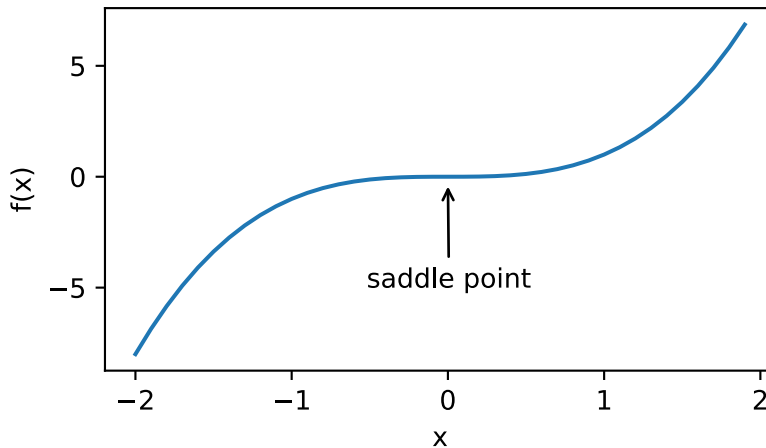
Saddle Points

As we just mentioned, one possible explanation for a gradient that approaches or becomes zero is that the current solution is close to a local optimum. In fact, there is another possibility. The current solution

could be near a saddle point. For example, given the function

$$f(x) = x^3,$$

we can find the position of the saddle point of this function.

```
In [3]: x = np.arange(-2.0, 2.0, 0.1)
        fig, = d2l.plt.plot(x, x**3)
        fig.axes.annotate('saddle point', xy=(0, -0.2), xytext=(-0.52, -5.0),
                          arrowprops=dict(arrowstyle='->'))
        d2l.plt.xlabel('x')
        d2l.plt.ylabel('f(x)');
```


Now, we will use another example of a two-dimensional function, defined as follows:

$$f(x, y) = x^2 - y^2.$$

We can find the position of the saddle point of this function. Perhaps you have noticed that the function looks just like a saddle, and the saddle point happens to be the center point of the seat.

```
In [4]: x, y = np.mgrid[-1: 1: 31j, -1: 1: 31j]
        z = x**2 - y**2

        ax = d2l.plt.figure().add_subplot(111, projection='3d')
        ax.plot_wireframe(x, y, z, **{'rstride': 2, 'cstride': 2})
        ax.plot([0], [0], [0], 'rx')
        ticks = [-1,  0, 1]
        d2l.plt.xticks(ticks)
        d2l.plt.yticks(ticks)
        ax.set_zticks(ticks)
        d2l.plt.xlabel('x')
        d2l.plt.ylabel('y');
```

8. Optimization Algorithms
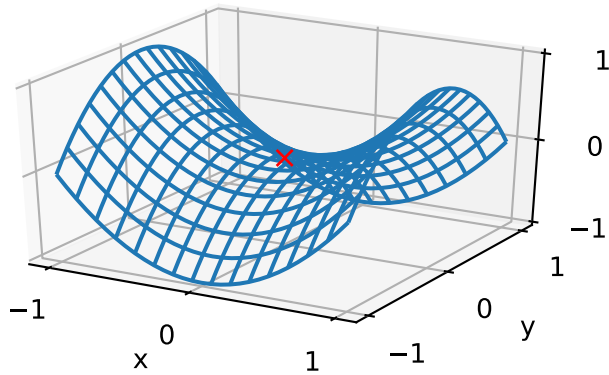
In the figure above, the objective function's local minimum and local maximum can be found on the x axis and y axis respectively at the position of the saddle point.

We assume that the input of a function is a k-dimensional vector and its output is a scalar, so its Hessian matrix will have k eigenvalues (see the *Mathematical Foundation* section). The solution of the function could be a local minimum, a local maximum, or a saddle point at a position where the function gradient is zero:

- When the eigenvalues of the function's Hessian matrix at the zero-gradient position are all positive, we have a local minimum for the function.

- When the eigenvalues of the function's Hessian matrix at the zero-gradient position are all negative, we have a local maximum for the function.

- When the eigenvalues of the function's Hessian matrix at the zero-gradient position are negative and positive, we have a saddle point for the function.

The random matrix theory tells us that, for a large Gaussian random matrix, the probability for any eigenvalue to be positive or negative is 0.5[1]. Thus, the probability of the first case above is 0.5^k. Since, generally, the parameters of deep learning models are high-dimensional (k is large), saddle points in the objective function are more commonly seen than local minimums.

In deep learning, it is difficult, but also not necessary, to find the global optimal solution of the objective function. In the subsequent sections of this chapter, we will introduce the optimization algorithms commonly used in deep learning one by one. These algorithms have trained some very effective deep learning models that have tackled practical problems.

Summary

- Because the objective function of the optimization algorithm is usually a loss function based on the training data set, the goal of optimization is to reduce the training error.

- Since, generally, the parameters of deep learning models are high-dimensional, saddle points in the objective function are more commonly seen than local minimums.

Exercises

- What other challenges involved in deep learning optimization can you think of?

Reference

[1] Wigner, E. P. (1958). On the distribution of the roots of certain symmetric matrices. Annals of Mathematics, 325-327.

Scan the QR Code to Discuss

8.2 Gradient Descent and Stochastic Gradient Descent

In this section, we are going to introduce the basic principles of gradient descent. Although it is not common for gradient descent to be used directly in deep learning, an understanding of gradients and the reason why the value of an objective function might decline when updating the independent variable along the opposite direction of the gradient is the foundation for future studies on optimization algorithms. Next, we are going to introduce stochastic gradient descent (SGD).

8.2.1 Gradient Descent in One-Dimensional Space

Here, we will use a simple gradient descent in one-dimensional space as an example to explain why the gradient descent algorithm may reduce the value of the objective function. We assume that the input and output of the continuously differentiable function $f : \mathbb{R} \to \mathbb{R}$ are both scalars. Given ϵ with a small enough absolute value, according to the Taylor's expansion formula from the *Mathematical basics* section, we get the following approximation:

$$f(x + \epsilon) \approx f(x) + \epsilon f'(x).$$

Here, $f'(x)$ is the gradient of function f at x. The gradient of a one-dimensional function is a scalar, also known as a derivative.

Next, find a constant $\eta > 0$, to make $|\eta f'(x)|$ sufficiently small so that we can replace ϵ with $-\eta f'(x)$ and get

$$f(x - \eta f'(x)) \approx f(x) - \eta f'(x)^2.$$

If the derivative $f'(x) \neq 0$, then $\eta f'(x)^2 > 0$, so

$$f(x - \eta f'(x)) \lesssim f(x).$$

This means that, if we use

$$x \leftarrow x - \eta f'(x)$$

to iterate x, the value of function $f(x)$ might decline. Therefore, in the gradient descent, we first choose an initial value x and a constant $\eta > 0$ and then use them to continuously iterate x until the stop condition is reached, for example, when the value of $f'(x)^2$ is small enough or the number of iterations has reached a certain value.

Now we will use the objective function $f(x) = x^2$ as an example to see how gradient descent is implemented. Although we know that $x = 0$ is the solution to minimize $f(x)$, here we still use this simple function to observe how x is iterated. First, import the packages or modules required for the experiment in this section.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        import math
        from mxnet import nd
        import numpy as np
```

Next, we use $x = 10$ as the initial value and assume $\eta = 0.2$. Using gradient descent to iterate x 10 times, we can see that, eventually, the value of x approaches the optimal solution.

```
In [2]: def gd(eta):
            x = 10
            results = [x]
            for i in range(10):
                x -= eta * 2 * x  # f(x) = x* the derivative of x is f'(x) = 2 * x
                results.append(x)
            print('epoch 10, x:', x)
            return results

        res = gd(0.2)

epoch 10, x: 0.06046617599999997
```
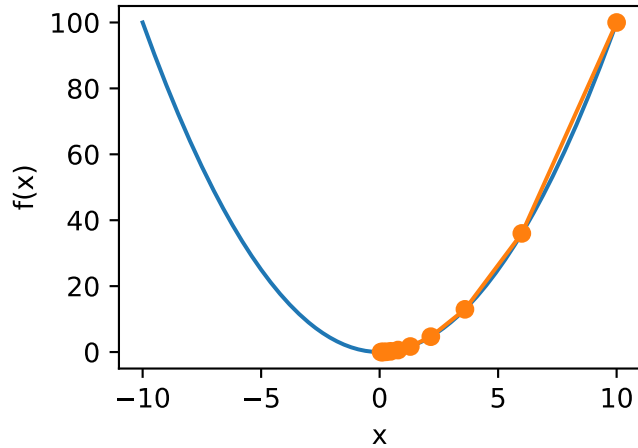
The iterative trajectory of the independent variable x is plotted as follows.

```
In [3]: def show_trace(res):
            n = max(abs(min(res)), abs(max(res)), 10)
            f_line = np.arange(-n, n, 0.1)
            d2l.set_figsize()
            d2l.plt.plot(f_line, [x * x for x in f_line])
            d2l.plt.plot(res, [x * x for x in res], '-o')
            d2l.plt.xlabel('x')
            d2l.plt.ylabel('f(x)')

        show_trace(res)
```


8.2.2 Learning Rate

The positive η in the above gradient descent algorithm is usually called the learning rate. This is a hyper-parameter and needs to be set manually. If we use a learning rate that is too small, it will cause x to update at a very slow speed, requiring more iterations to get a better solution. Here, we have the iterative trajectory of the independent variable x with the learning rate $\eta = 0.05$. As we can see, after iterating 10 times when the learning rate is too small, there is still a large deviation between the final value of x and the optimal solution.

```
In [4]: show_trace(gd(0.05))

epoch 10, x: 3.4867844009999995
```

8. Optimization Algorithms
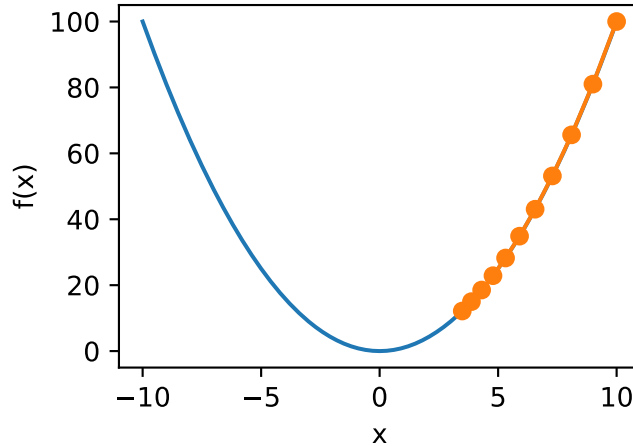
If we use an excessively high learning rate, $|\eta f'(x)|$ might be too large for the first-order Taylor expansion formula mentioned above to hold. In this case, we cannot guarantee that the iteration of x will be able to lower the value of $f(x)$. For example, when we set the learning rate to $\eta = 1.1$, x overshoots the optimal solution $x = 0$ and gradually diverges.

```
In [5]: show_trace(gd(1.1))

epoch 10, x: 61.917364224000096
```
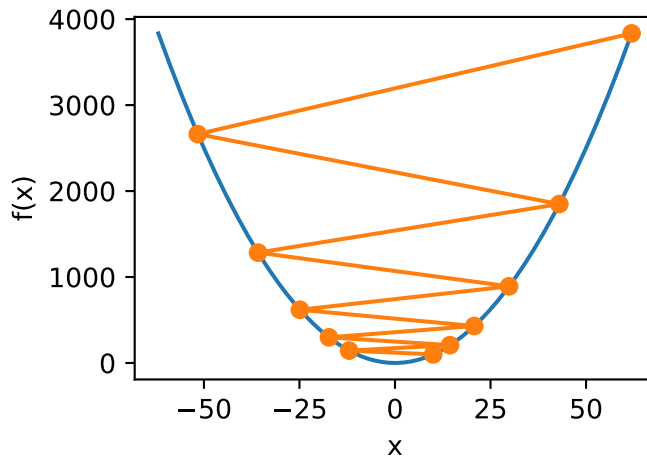

8.2.3 Gradient Descent in Multi-Dimensional Space

Now that we understand gradient descent in one-dimensional space, let us consider a more general case: the input of the objective function is a vector and the output is a scalar. We assume that the input of the

---

target function $f : \mathbb{R}^d \to \mathbb{R}$ is the d-dimensional vector $\boldsymbol{x} = [x_1, x_2, \ldots, x_d]^\top$. The gradient of the objective function $f(\boldsymbol{x})$ with respect to $\boldsymbol{x}$ is a vector consisting of d partial derivatives:

$$\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = \left[\frac{\partial f(\boldsymbol{x})}{\partial x_1}, \frac{\partial f(\boldsymbol{x})}{\partial x_2}, \ldots, \frac{\partial f(\boldsymbol{x})}{\partial x_d} \right]^\top.$$

For brevity, we use $\nabla f(\boldsymbol{x})$ instead of $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$. Each partial derivative element $\partial f(\boldsymbol{x})/\partial x_i$ in the gradient indicates the rate of change of f at $\boldsymbol{x}$ with respect to the input x_i. To measure the rate of change of f in the direction of the unit vector $\boldsymbol{u}$ ($\|\boldsymbol{u}\| = 1$), in multivariate calculus, the directional derivative of f at $\boldsymbol{x}$ in the direction of $\boldsymbol{u}$ is defined as

$$\mathrm{D}_{\boldsymbol{u}} f(\boldsymbol{x}) = \lim_{h \to 0} \frac{f(\boldsymbol{x} + h\boldsymbol{u}) - f(\boldsymbol{x})}{h}.$$

According to the property of directional derivatives [1Chapter 14.6 Theorem 3], the aforementioned directional derivative can be rewritten as

$$\mathrm{D}_{\boldsymbol{u}} f(\boldsymbol{x}) = \nabla f(\boldsymbol{x}) \cdot \boldsymbol{u}.$$

The directional derivative $\mathrm{D}_{\boldsymbol{u}} f(\boldsymbol{x})$ gives all the possible rates of change for f along $\boldsymbol{x}$. In order to minimize f, we hope to find the direction the will allow us to reduce f in the fastest way. Therefore, we can use the unit vector $\boldsymbol{u}$ to minimize the directional derivative $\mathrm{D}_{\boldsymbol{u}} f(\boldsymbol{x})$.

For $\mathrm{D}_{\boldsymbol{u}} f(\boldsymbol{x}) = \|\nabla f(\boldsymbol{x})\| \cdot \|\boldsymbol{u}\| \cdot \cos(\theta) = \|\nabla f(\boldsymbol{x})\| \cdot \cos(\theta)$, Here, θ is the angle between the gradient $\nabla f(\boldsymbol{x})$ and the unit vector $\boldsymbol{u}$. When $\theta = \pi$, $\cos(\theta)$ gives us the minimum value -1. So when $\boldsymbol{u}$ is in a direction that is opposite to the gradient direction $\nabla f(\boldsymbol{x})$, the direction derivative $\mathrm{D}_{\boldsymbol{u}} f(\boldsymbol{x})$ is minimized. Therefore, we may continue to reduce the value of objective function f by the gradient descent algorithm:

$\boldsymbol{x} \leftarrow \boldsymbol{x} - \eta \nabla f(\boldsymbol{x}).$

Similarly, η (positive) is called the learning rate.

Now we are going to construct an objective function $f(\boldsymbol{x}) = x_1^2 + 2x_2^2$ with a two-dimensional vector $\boldsymbol{x} = [x_1, x_2]^\top$ as input and a scalar as the output. So we have the gradient $\nabla f(\boldsymbol{x}) = [2x_1, 4x_2]^\top$. We will observe the iterative trajectory of independent variable $\boldsymbol{x}$ by gradient descent from the initial position $[-5, -2]$. First, we are going to define two helper functions. The first helper uses the given independent variable update function to iterate independent variable $\boldsymbol{x}$ a total of 20 times from the initial position $[-5, -2]$. The second helper will visualize the iterative trajectory of independent variable $\boldsymbol{x}$.

```
In [6]:  # This function is saved in the d2l package for future use
         def train_2d(trainer):
             # s1 and s2 are states of the independent variable and will be used later
             # in the chapter
             x1, x2, s1, s2 = -5, -2, 0, 0
             results = [(x1, x2)]
             for i in range(20):
                 x1, x2, s1, s2 = trainer(x1, x2, s1, s2)
                 results.append((x1, x2))
             print('epoch %d, x1 %f, x2 %f' % (i + 1, x1, x2))
             return results
```

```
# This function is saved in the d2l package for future use
def show_trace_2d(f, results):
    d2l.plt.plot(*zip(*results), '-o', color='#ff7f0e')
    x1, x2 = np.meshgrid(np.arange(-5.5, 1.0, 0.1), np.arange(-3.0, 1.0, 0.1))
    d2l.plt.contour(x1, x2, f(x1, x2), colors='#1f77b4')
    d2l.plt.xlabel('x1')
    d2l.plt.ylabel('x2')
```

Next, we observe the iterative trajectory of the independent variable at learning rate 0.1. After iterating the independent variable x 20 times using gradient descent, we can see that. eventually, the value of x approaches the optimal solution $[0, 0]$.

```
In [7]: eta = 0.1

        def f_2d(x1, x2):   # Objective function
            return x1 ** 2 + 2 * x2 ** 2

        def gd_2d(x1, x2, s1, s2):
            return (x1 - eta * 2 * x1, x2 - eta * 4 * x2, 0, 0)

        show_trace_2d(f_2d, train_2d(gd_2d))
```

```
epoch 20, x1 -0.057646, x2 -0.000073
```
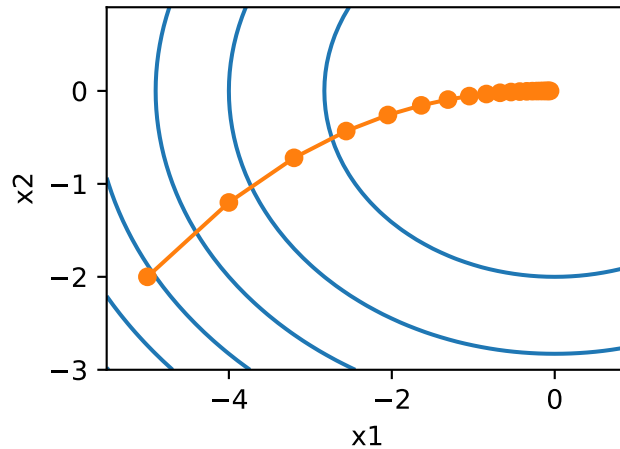

8.2.4 Stochastic Gradient Descent (SGD)

In deep learning, the objective function is usually the average of the loss functions for each example in the training data set. We assume that $f_i(x)$ is the loss function of the training data instance with n examples, an index of i, and parameter vector of x, then we have the objective function

$$f(x) = \frac{1}{n} \sum_{i=1}^{n} f_i(x).$$

The gradient of the objective function at x is computed as

$$\nabla f(\boldsymbol{x}) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\boldsymbol{x}).$$

If gradient descent is used, the computing cost for each independent variable iteration is $\mathcal{O}(n)$, which grows linearly with n. Therefore, when the model training data instance is large, the cost of gradient descent for each iteration will be very high.

Stochastic gradient descent (SGD) reduces computational cost at each iteration. At each iteration of stochastic gradient descent, we uniformly sample an index $i \in \{1, \ldots, n\}$ for data instances at random, and compute the gradient $\nabla f_i(\boldsymbol{x})$ to update $\boldsymbol{x}$:

$$\boldsymbol{x} \leftarrow \boldsymbol{x} - \eta \nabla f_i(\boldsymbol{x}).$$

Here, η is the learning rate. We can see that the computing cost for each iteration drops from $\mathcal{O}(n)$ of the gradient descent to the constant $\mathcal{O}(1)$. We should mention that the stochastic gradient $\nabla f_i(\boldsymbol{x})$ is the unbiased estimate of gradient $\nabla f(\boldsymbol{x})$.

$$\mathbb{E}_i \nabla f_i(\boldsymbol{x}) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(\boldsymbol{x}) = \nabla f(\boldsymbol{x}).$$

This means that, on average, the stochastic gradient is a good estimate of the gradient.

Now, we will compare it to gradient descent by adding random noise with a mean of 0 to the gradient to simulate ab SGD.

```
In [8]: def sgd_2d(x1, x2, s1, s2):
            return (x1 - eta * (2 * x1 + np.random.normal(0.1)),
                    x2 - eta * (4 * x2 + np.random.normal(0.1)), 0, 0)

        show_trace_2d(f_2d, train_2d(sgd_2d))
epoch 20, x1 -0.118918, x2 -0.112748
```
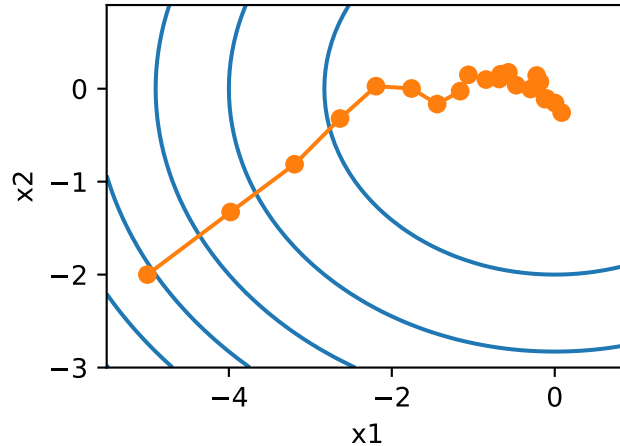
As we can see, the iterative trajectory of the independent variable in the SGD is more tortuous than in the gradient descent. This is due to the noise added in the experiment, which reduced the accuracy of the simulated stochastic gradient. In practice, such noise usually comes from individual examples in the training data set.

Summary

- If we use a more suitable learning rate and update the independent variable in the opposite direction of the gradient, the value of the objective function might be reduced. Gradient descent repeats this update process until a solution that meets the requirements is obtained.

- Problems occur when the learning rate is tool small or too large. A suitable learning rate is usually found only after multiple experiments.

- When there are more examples in the training data set, it costs more to compute each iteration for gradient descent, so SGD is preferred in these cases.

Exercises

- Using a different objective function, observe the iterative trajectory of the independent variable in gradient descent and the SGD.

- In the experiment for gradient descent in two-dimensional space, try to use different learning rates to observe and analyze the experimental phenomena.

8.3 Mini-batch Stochastic Gradient Descent

In each iteration, the gradient descent uses the entire training data set to compute the gradient, so it is sometimes referred to as batch gradient descent. Stochastic gradient descent (SGD) only randomly select one example in each iteration to compute the gradient. Just like in the previous chapters, we can perform random uniform sampling for each iteration to form a mini-batch and then use this mini-batch to compute the gradient. Now, we are going to discuss mini-batch stochastic gradient descent.

Set objective function $f(\boldsymbol{x}) : \mathbb{R}^d \to \mathbb{R}$. The time step before the start of iteration is set to 0. The independent variable of this time step is $\boldsymbol{x}_0 \in \mathbb{R}^d$ and is usually obtained by random initialization. In each subsequent time step $t > 0$, mini-batch SGD uses random uniform sampling to get a mini-batch $\mathcal{B}_t$ made of example indices from the training data set. We can use sampling with replacement or sampling without replacement to get a mini-batch example. The former method allows duplicate examples in the same mini-batch, the latter does not and is more commonly used. We can use either of the two methods

$$\boldsymbol{g}_t \leftarrow \nabla f_{\mathcal{B}_t}(\boldsymbol{x}_{t-1}) = \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \nabla f_i(\boldsymbol{x}_{t-1})$$

to compute the gradient $\boldsymbol{g}_t$ of the objective function at $\boldsymbol{x}_{t-1}$ with mini-batch $\mathcal{B}_t$ at time step t. Here, $|\mathcal{B}|$ is the size of the batch, which is the number of examples in the mini-batch. This is a hyper-parameter. Just like the stochastic gradient, the mini-batch SGD $\boldsymbol{g}_t$ obtained by sampling with replacement is also the unbiased estimate of the gradient $\nabla f(\boldsymbol{x}_{t-1})$. Given the learning rate η_t (positive), the iteration of the mini-batch SGD on the independent variable is as follows:

$$\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} - \eta_t \boldsymbol{g}_t.$$

The variance of the gradient based on random sampling cannot be reduced during the iterative process, so in practice, the learning rate of the (mini-batch) SGD can self-decay during the iteration, such as $\eta_t = \eta t^\alpha$ (usually $\alpha = -1$ or -0.5), $\eta_t = \eta \alpha^t$ (e.g $\alpha = 0.95$), or learning rate decay once per iteration or after several iterations. As a result, the variance of the learning rate and the (mini-batch) SGD will decrease. Gradient descent always uses the true gradient of the objective function during the iteration, without the need to self-decay the learning rate.

The cost for computing each iteration is $\mathcal{O}(|\mathcal{B}|)$. When the batch size is 1, the algorithm is an SGD; when the batch size equals the example size of the training data, the algorithm is a gradient descent. When the batch size is small, fewer examples are used in each iteration, which will result in parallel processing and

reduce the RAM usage efficiency. This makes it more time consuming to compute examples of the same size than using larger batches. When the batch size increases, each mini-batch gradient may contain more redundant information. To get a better solution, we need to compute more examples for a larger batch size, such as increasing the number of epochs.

8.3.1 Reading Data

In this chapter, we will use a data set developed by NASA to test the wing noise from different aircraft to compare these optimization algorithms[1]. We will use the first 1500 examples of the data set, 5 features, and a normalization method to preprocess the data.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet import autograd, gluon, init, nd
        from mxnet.gluon import nn, data as gdata, loss as gloss
        import numpy as np
        import time

        # This function is saved in the d2l package for future use
        def get_data_ch7():
            data = np.genfromtxt('../data/airfoil_self_noise.dat', delimiter='\t')
            data = (data - data.mean(axis=0)) / data.std(axis=0)
            return nd.array(data[:1500, :-1]), nd.array(data[:1500, -1])

        features, labels = get_data_ch7()
        features.shape
Out[1]: (1500, 5)
```

8.3.2 Implementation from Scratch

We have already implemented the mini-batch SGD algorithm in the Linear Regression Implemented From Scratch section. We have made its input parameters more generic here, so that we can conveniently use the same input for the other optimization algorithms introduced later in this chapter. Specifically, we add the status input `states` and place the hyper-parameter in dictionary `hyperparams`. In addition, we will average the loss of each mini-batch example in the training function, so the gradient in the optimization algorithm does not need to be divided by the batch size.

```
In [2]: def sgd(params, states, hyperparams):
            for p in params:
                p[:] -= hyperparams['lr'] * p.grad
```

Next, we are going to implement a generic training function to facilitate the use of the other optimization algorithms introduced later in this chapter. It initializes a linear regression model and can then be used to train the model with the mini-batch SGD and other algorithms introduced in subsequent sections.

```
In [3]: # This function is saved in the d2l package for future use
        def train_ch7(trainer_fn, states, hyperparams, features, labels,
                      batch_size=10, num_epochs=2):
            # Initialize model parameters
            net, loss = d2l.linreg, d2l.squared_loss
            w = nd.random.normal(scale=0.01, shape=(features.shape[1], 1))
            b = nd.zeros(1)
            w.attach_grad()
            b.attach_grad()

            def eval_loss():
                return loss(net(features, w, b), labels).mean().asscalar()

            ls = [eval_loss()]
            data_iter = gdata.DataLoader(
                gdata.ArrayDataset(features, labels), batch_size, shuffle=True)
            for _ in range(num_epochs):
                start = time.time()
                for batch_i, (X, y) in enumerate(data_iter):
                    with autograd.record():
                        l = loss(net(X, w, b), y).mean()  # Average the loss
                    l.backward()
                    # Update model parameters
                    trainer_fn([w, b], states, hyperparams)
                    if (batch_i + 1) * batch_size % 100 == 0:
                        # Record the current training error for every 100 examples
                        ls.append(eval_loss())
            # Print and plot the results.
            print('loss: %f, %f sec per epoch' % (ls[-1], time.time() - start))
            d2l.set_figsize()
            d2l.plt.plot(np.linspace(0, num_epochs, len(ls)), ls)
            d2l.plt.xlabel('epoch')
            d2l.plt.ylabel('loss')
```

When the batch size equals 1500 (the total number of examples), we use gradient descent for optimization. The model parameters will be iterated only once for each epoch of the gradient descent. As we can see, the downward trend of the value of the objective function (training loss) flattened out after 6 iterations.
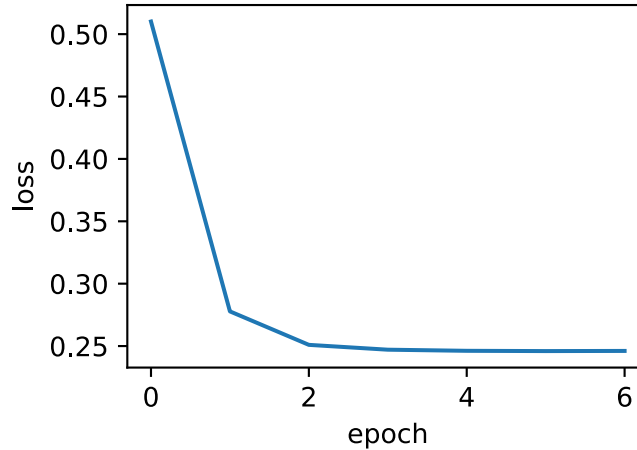
```
In [4]: def train_sgd(lr, batch_size, num_epochs=2):
            train_ch7(sgd, None, {'lr': lr}, features, labels, batch_size, num_epochs)

        train_sgd(1, 1500, 6)

loss: 0.246095, 0.023489 sec per epoch
```
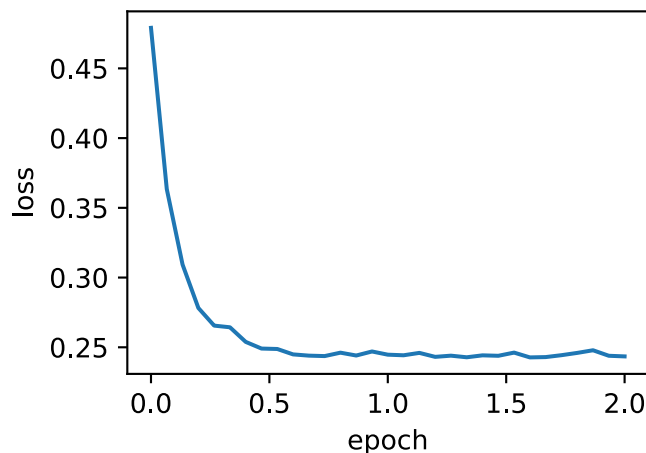
When the batch size equals 1, we use SGD for optimization. In order to simplify the implementation, we did not self-decay the learning rate. Instead, we simply used a small constant for the learning rate in the (mini-batch) SGD experiment. In SGD, the independent variable (model parameter) is updated whenever an example is processed. Thus it is updated 1500 times in one epoch. As we can see, the decline in the value of the objective function slows down after one epoch.

Although both the procedures processed 1500 examples within one epoch, SGD consumes more time than gradient descent in our experiment. This is because SGD performed more iterations on the independent variable within one epoch, and it is harder for single-example gradient computation to use parallel computing effectively.

```
In [5]: train_sgd(0.005, 1)

loss: 0.243466, 1.965798 sec per epoch
```

When the batch size equals 10, we use mini-batch SGD for optimization. The time required for one epoch is between the time needed for gradient descent and SGD to complete the same epoch.

```
In [6]: train_sgd(0.05, 10)
```

```
loss: 0.243906, 0.186648 sec per epoch
```
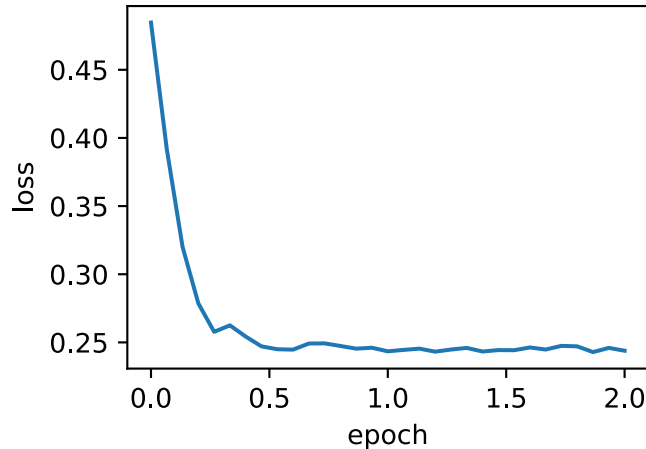

8.3.3 Concise Implementation

In Gluon, we can use the `Trainer` class to call optimization algorithms. Next, we are going to implement a generic training function that uses the optimization name `trainer name` and hyperparameter `trainer_hyperparameter` to create the instance `Trainer`.

```
In [7]: # This function is saved in the d2l package for future use
        def train_gluon_ch7(trainer_name, trainer_hyperparams, features, labels,
                            batch_size=10, num_epochs=2):
            # Initialize model parameters
            net = nn.Sequential()
            net.add(nn.Dense(1))
            net.initialize(init.Normal(sigma=0.01))
            loss = gloss.L2Loss()

            def eval_loss():
                return loss(net(features), labels).mean().asscalar()

            ls = [eval_loss()]
            data_iter = gdata.DataLoader(
                gdata.ArrayDataset(features, labels), batch_size, shuffle=True)
            # Create the instance "Trainer" to update model parameter(s)
            trainer = gluon.Trainer(
                net.collect_params(), trainer_name, trainer_hyperparams)
            for _ in range(num_epochs):
                start = time.time()
                for batch_i, (X, y) in enumerate(data_iter):
```

```
        with autograd.record():
            l = loss(net(X), y)
        l.backward()
        # Average the gradient in the "Trainer" instance
        trainer.step(batch_size)
        if (batch_i + 1) * batch_size % 100 == 0:
            ls.append(eval_loss())
    # Print and plot the results
    print('loss: %f, %f sec per epoch' % (ls[-1], time.time() - start))
    d2l.set_figsize()
    d2l.plt.plot(np.linspace(0, num_epochs, len(ls)), ls)
    d2l.plt.xlabel('epoch')
    d2l.plt.ylabel('loss')
```

Use Gluon to repeat the last experiment.

```
In [8]: train_gluon_ch7('sgd', {'learning_rate': 0.05}, features, labels, 10)

loss: 0.251214, 0.181255 sec per epoch
```
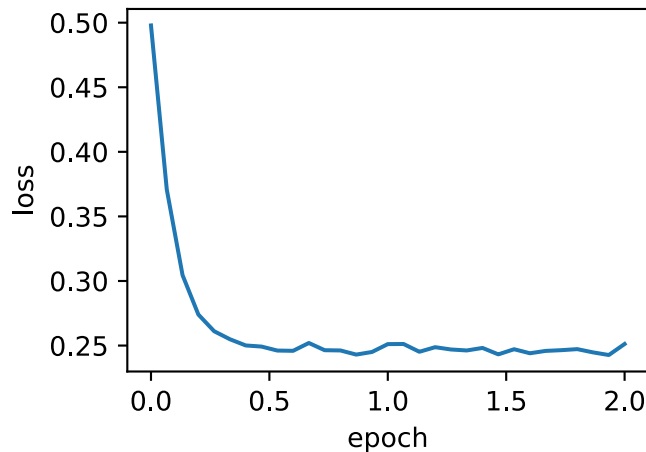

Summary

- Mini-batch stochastic gradient uses random uniform sampling to get a mini-batch training example for gradient computation.

- In practice, learning rates of the (mini-batch) SGD can self-decay during iteration.

- In general, the time consumption per epoch for mini-batch stochastic gradient is between what takes for gradient descent and SGD to complete the same epoch.

Exercises

- Modify the batch size and learning rate and observe the rate of decline for the value of the objective function and the time consumed in each epoch.

- Read the MXNet documentation and use the `Trainer` class `set_learning_rate` function to reduce the learning rate of the mini-batch SGD to 1/10 of its previous value after each epoch.

Reference

[1] Aircraft wing noise data set. https://archive.ics.uci.edu/ml/datasets/Airfoil+Self-Noise

Scan the QR Code to Discuss

8.4 Momentum

In the *Gradient Descent and Stochastic Gradient Descent* section, we mentioned that the gradient of the objective function's independent variable represents the direction of the objective function's fastest descend at the current position of the independent variable. Therefore, gradient descent is also called steepest descent. In each iteration, the gradient descends according to the current position of the independent variable while updating the latter along the current position of the gradient. However, this can lead to problems if the iterative direction of the independent variable relies exclusively on the current position of the independent variable.

8.4.1 Exercises with Gradient Descent

Now, we will consider an objective function $f(\boldsymbol{x}) = 0.1x_1^2 + 2x_2^2$, whose input and output are a two-dimensional vector $\boldsymbol{x} = [x_1, x_2]$ and a scalar, respectively. In contrast to the *Gradient Descent and Stochastic Gradient Descent* section, here, the coefficient x_1^2 is reduced from 1 to 0.1. We are going to implement gradient descent based on this objective function, and demonstrate the iterative trajectory of the independent variable using the learning rate $0.4.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
```

```
import d2l
from mxnet import nd

eta = 0.4

def f_2d(x1, x2):
    return 0.1 * x1 ** 2 + 2 * x2 ** 2

def gd_2d(x1, x2, s1, s2):
    return (x1 - eta * 0.2 * x1, x2 - eta * 4 * x2, 0, 0)

d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))
```
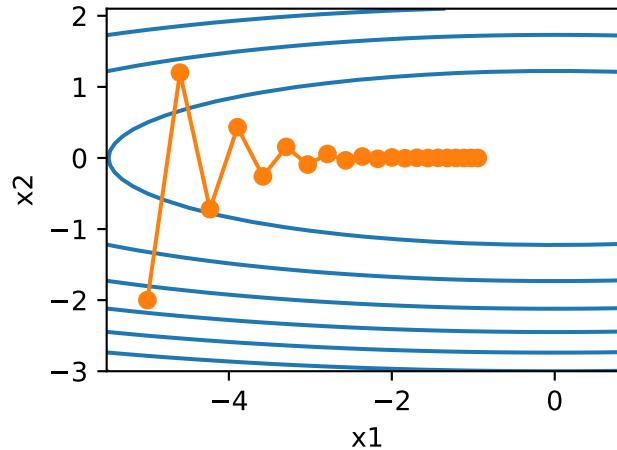
epoch 20, x1 -0.943467, x2 -0.000073

As we can see, at the same position, the slope of the objective function has a larger absolute value in the vertical direction (x_2 axis direction) than in the horizontal direction (x_1 axis direction). Therefore, given the learning rate, using gradient descent for interaction will cause the independent variable to move more in the vertical direction than in the horizontal one. So we need a small learning rate to prevent the independent variable from overshooting the optimal solution for the objective function in the vertical direction. However, it will cause the independent variable to move slower toward the optimal solution in the horizontal direction.
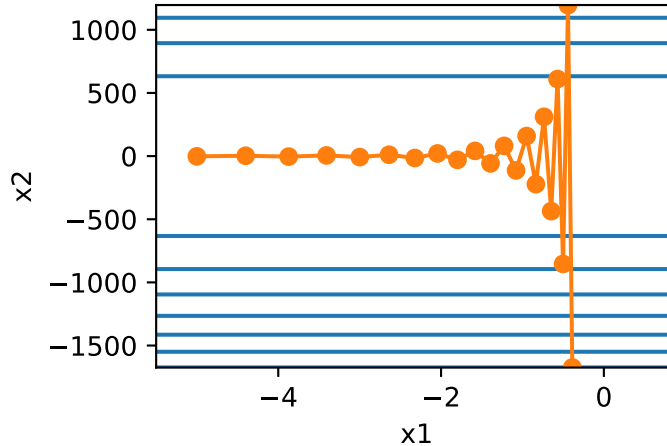
Now, we try to make the learning rate slightly larger, so the independent variable will continuously overshoot the optimal solution in the vertical direction and gradually diverge.

```
In [2]: eta = 0.6
        d2l.show_trace_2d(f_2d, d2l.train_2d(gd_2d))
```

epoch 20, x1 -0.387814, x2 -1673.365109

8.4.2 The Momentum Method

The momentum method was proposed to solve the gradient descent problem described above. Since mini-batch stochastic gradient descent is more general than gradient descent, the subsequent discussion in this chapter will continue to use the definition for mini-batch stochastic gradient descent $\boldsymbol{g}_t$ at time step t given in the *Mini-batch Stochastic Gradient Descent* section. We set the independent variable at time step t to $\boldsymbol{x}_t$ and the learning rate to η_t. At time step 0, momentum creates the velocity variable $\boldsymbol{v}_0$ and initializes its elements to zero. At time step $t > 0$, momentum modifies the steps of each iteration as follows:

$$\boldsymbol{v}_t \leftarrow \gamma \boldsymbol{v}_{t-1} + \eta_t \boldsymbol{g}_t,$$
$$\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} - \boldsymbol{v}_t,$$

Here, the momentum hyperparameter γ satisfies $0 \leq \gamma < 1$. When $\gamma = 0$, momentum is equivalent to a mini-batch SGD.

Before explaining the mathematical principles behind the momentum method, we should take a look at the iterative trajectory of the gradient descent after using momentum in the experiment.

```
In [3]: def momentum_2d(x1, x2, v1, v2):
            v1 = gamma * v1 + eta * 0.2 * x1
            v2 = gamma * v2 + eta * 4 * x2
            return x1 - v1, x2 - v2, v1, v2

        eta, gamma = 0.4, 0.5
        d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))

epoch 20, x1 -0.062843, x2 0.001202
```
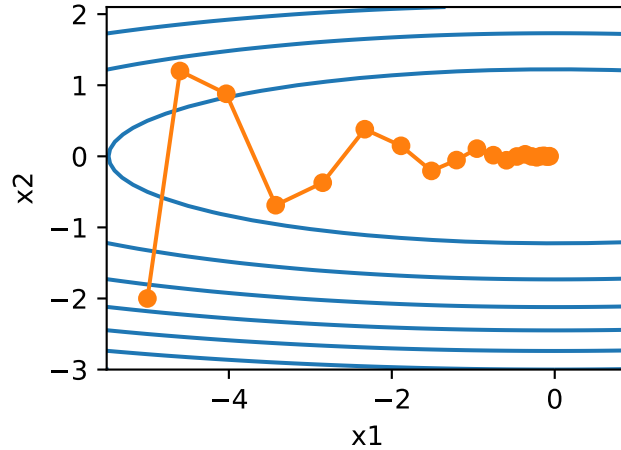
As we can see, when using a smaller learning rate ($\eta = 0.4$) and momentum hyperparameter ($\gamma = 0.5$), momentum moves more smoothly in the vertical direction and approaches the optimal solution faster in the horizontal direction. Now, when we use a larger learning rate ($\eta = 0.6$), the independent variable will no longer diverge.

```
In [4]: eta = 0.6
        d2l.show_trace_2d(f_2d, d2l.train_2d(momentum_2d))
```

```
epoch 20, x1 0.007188, x2 0.002553
```
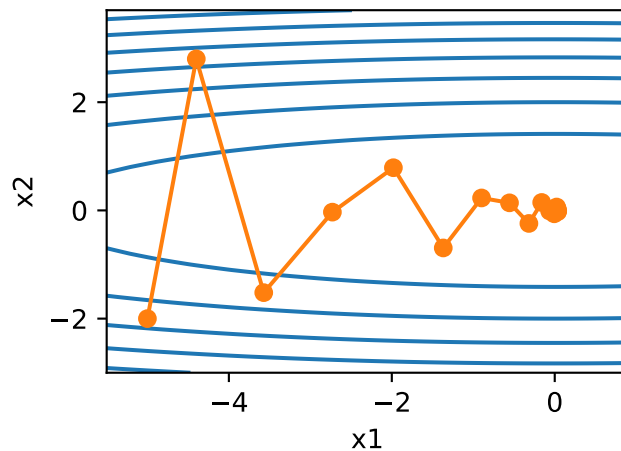
Exponentially Weighted Moving Average (EWMA)

In order to understand the momentum method mathematically, we must first explain the exponentially weighted moving average (EWMA). Given hyperparameter $0 \leq \gamma < 1$, the variable y_t of the current time step t is the linear combination of variable y_{t-1} from the previous time step $t-1$ and another variable x_t of the current step.

$$y_t = \gamma y_{t-1} + (1 - \gamma) x_t.$$

We can expand y_t:

$$
\begin{aligned}
y_t &= (1 - \gamma) x_t + \gamma y_{t-1} \\
&= (1 - \gamma) x_t + (1 - \gamma) \cdot \gamma x_{t-1} + \gamma^2 y_{t-2} \\
&= (1 - \gamma) x_t + (1 - \gamma) \cdot \gamma x_{t-1} + (1 - \gamma) \cdot \gamma^2 x_{t-2} + \gamma^3 y_{t-3} \\
&\cdots
\end{aligned}
$$

Let $n = 1/(1 - \gamma)$, so $(1 - 1/n)^n = \gamma^{1/(1-\gamma)}$. Because

$$\lim_{n \to \infty} \left(1 - \frac{1}{n} \right)^n = \exp(-1) \approx 0.3679,$$

when $\gamma \to 1$, $\gamma^{1/(1-\gamma)} = \exp(-1)$. For example, $0.95^{20} \approx \exp(-1)$. If we treat $\exp(-1)$ as a relatively small number, we can ignore all the terms that have $\gamma^{1/(1-\gamma)}$ or coefficients of higher order than $\gamma^{1/(1-\gamma)}$ in them. For example, when $\gamma = 0.95$,

$$y_t \approx 0.05 \sum_{i=0}^{19} 0.95^i x_{t-i}.$$

Therefore, in practice, we often treat y_t as the weighted average of the x_t values from the last $1/(1 - \gamma)$ time steps. For example, when $\gamma = 0.95$, y_t can be treated as the weighted average of the x_t values from the last 20 time steps; when $\gamma = 0.9$, y_t can be treated as the weighted average of the x_t values from the last 10 time steps. Additionally, the closer the x_t value is to the current time step t, the greater the value's weight (closer to 1).

Understanding Momentum through EWMA

Now, we are going to deform the velocity variable of momentum:

$$\boldsymbol{v}_t \leftarrow \gamma \boldsymbol{v}_{t-1} + (1 - \gamma) \left(\frac{\eta_t}{1 - \gamma} \boldsymbol{g}_t \right).$$

By the form of EWMA, velocity variable $\boldsymbol{v}_t$ is actually an EWMA of time series $\{\eta_{t-i} \boldsymbol{g}_{t-i}/(1 - \gamma) : i = 0, \ldots, 1/(1 - \gamma) - 1\}$. In other words, considering mini-batch SGD, the update of an independent variable with momentum at each time step approximates the EWMA of the updates in the last $1/(1 - \gamma)$

8. Optimization Algorithms

time steps without momentum, divided by $1 - \gamma$. Thus, with momentum, the movement size at each direction not only depends on the current gradient, but also depends on whether the past gradients are aligned at each direction. In the optimization problem mentioned earlier in this section, all the gradients are positive in the horizontal direction (rightward), but are occasionally positive (up) or negative (down) in the vertical direction. As a result, we can use a larger learning rate to allow the independent variable move faster towards the optimum.

8.4.3 Implementation from Scratch

Compared with mini-batch SGD, the momentum method needs to maintain a velocity variable of the same shape for each independent variable and a momentum hyperparameter is added to the hyperparameter category. In the implementation, we use the state variable `states` to represent the velocity variable in a more general sense.

```
In [5]: features, labels = d2l.get_data_ch7()

        def init_momentum_states():
            v_w = nd.zeros((features.shape[1], 1))
            v_b = nd.zeros(1)
            return (v_w, v_b)

        def sgd_momentum(params, states, hyperparams):
            for p, v in zip(params, states):
                v[:] = hyperparams['momentum'] * v + hyperparams['lr'] * p.grad
                p[:] -= v
```

When we set the momentum hyperparameter `momentum` to 0.5, it can be treated as a mini-batch SGD: the mini-batch gradient here is the weighted average of twice the mini-batch gradient of the last two time steps.

```
In [6]: d2l.train_ch7(sgd_momentum, init_momentum_states(),
                    {'lr': 0.02, 'momentum': 0.5}, features, labels)

loss: 0.245601, 0.295858 sec per epoch
```
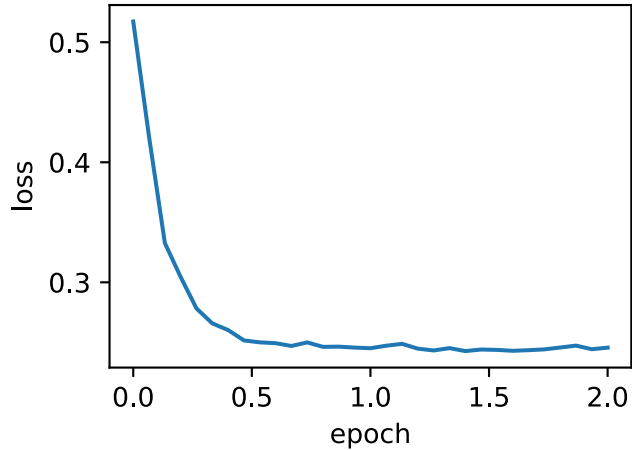
When we increase the momentum hyperparameter `momentum` to 0.9, it can still be treated as a mini-batch SGD: the mini-batch gradient here will be the weighted average of ten times the mini-batch gradient of the last 10 time steps. Now we keep the learning rate at 0.02.

```
In [7]: d2l.train_ch7(sgd_momentum, init_momentum_states(),
                       {'lr': 0.02, 'momentum': 0.9}, features, labels)
```

```
loss: 0.261216, 0.295778 sec per epoch
```
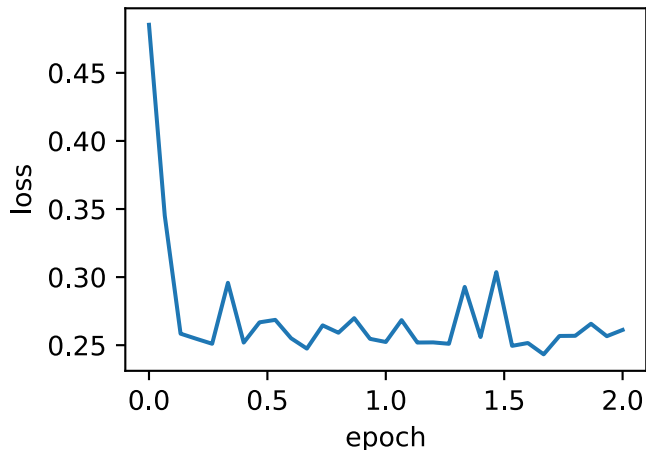

We can see that the value change of the objective function is not smooth enough at later stages of iteration. Intuitively, ten times the mini-batch gradient is five times larger than two times the mini-batch gradient, so we can try to reduce the learning rate to 1/5 of its original value. Now, the value change of the objective function becomes smoother after its period of decline.

```
In [8]: d2l.train_ch7(sgd_momentum, init_momentum_states(),
```

8. Optimization Algorithms

```
                          {'lr': 0.004, 'momentum': 0.9}, features, labels)
loss: 0.243154, 0.292594 sec per epoch
```
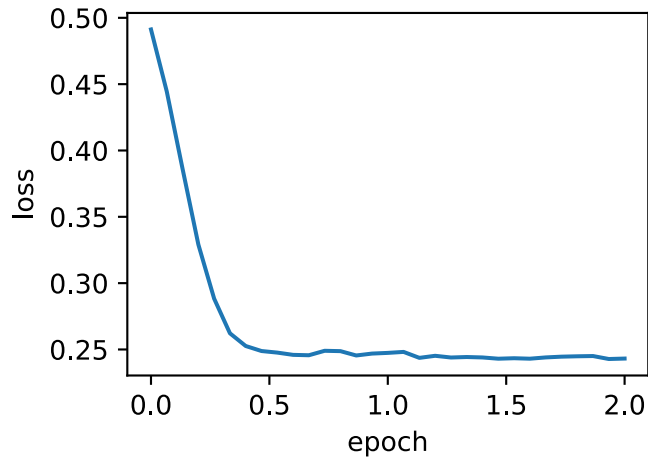

8.4.4 Concse Implementation

In Gluon, we only need to use `momentum` to define the momentum hyperparameter in the `Trainer` instance to implement momentum.

```
In [9]: d2l.train_gluon_ch7('sgd', {'learning_rate': 0.004, 'momentum': 0.9},
                            features, labels)
loss: 0.246690, 0.228055 sec per epoch
```

Summary

- The momentum method uses the EWMA concept. It takes the weighted average of past time steps, with weights that decay exponentially by the time step.
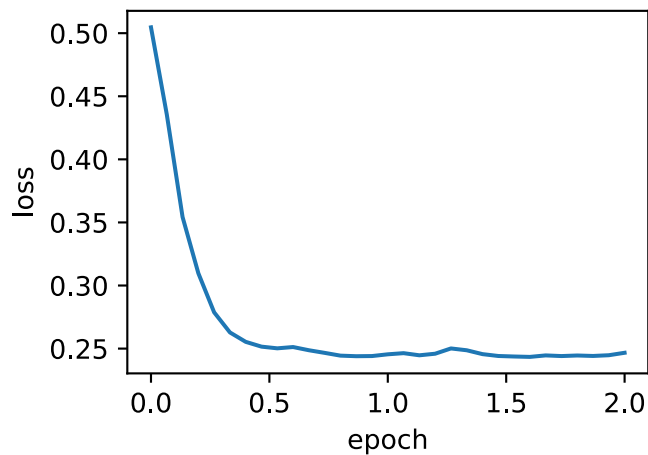
- Momentum makes independent variable updates for adjacent time steps more consistent in direction.

Exercises

- Use other combinations of momentum hyperparameters and learning rates and observe and analyze the different experimental results.

Scan the QR Code to Discuss

8.5 Adagrad

In the optimization algorithms we introduced previously, each element of the objective function's independent variables uses the same learning rate at the same time step for self-iteration. For example, if we assume that the objective function is f and the independent variable is a two-dimensional vector $[x_1, x_2]^\top$, each element in the vector uses the same learning rate when iterating. For example, in gradient descent with the learning rate η, element x_1 and x_2 both use the same learning rate η for iteration:

$$x_1 \leftarrow x_1 - \eta \frac{\partial f}{\partial x_1}, \quad x_2 \leftarrow x_2 - \eta \frac{\partial f}{\partial x_2}.$$

In the *Momentum* section, we can see that, when there is a big difference between the gradient values x_1 and x_2, a sufficiently small learning rate needs to be selected so that the independent variable will not diverge in the dimension of larger gradient values. However, this will cause the independent variables to iterate too slowly in the dimension with smaller gradient values. The momentum method relies on the exponentially weighted moving average (EWMA) to make the direction of the independent variable more consistent, thus reducing the possibility of divergence. In this section, we are going to introduce Adagrad, an algorithm that adjusts the learning rate according to the gradient value of the independent variable in each dimension to eliminate problems caused when a unified learning rate has to adapt to all dimensions.

8.5.1 The Algorithm

The Adadelta algorithm uses the cumulative variable s_t obtained from a square by element operation on the mini-batch stochastic gradient g_t. At time step 0, Adagrad initializes each element in s_0 to 0. At time step t, we first sum the results of the square by element operation for the mini-batch gradient g_t to get the variable s_t:

$$s_t \leftarrow s_{t-1} + g_t \odot g_t,$$

Here, $\odot$ is the symbol for multiplication by element. Next, we re-adjust the learning rate of each element in the independent variable of the objective function using element operations:

$$x_t \leftarrow x_{t-1} - \frac{\eta}{\sqrt{s_t + \epsilon}} \odot g_t,$$

Here, eta is the learning rate while ϵ is a constant added to maintain numerical stability, such as 10^{-6}. Here, the square root, division, and multiplication operations are all element operations. Each element in the independent variable of the objective function will have its own learning rate after the operations by elements.

8.5.2 Features

We should emphasize that the cumulative variable s_t produced by a square by element operation on the mini-batch stochastic gradient is part of the learning rate denominator. Therefore, if an element in the independent variable of the objective function has a constant and large partial derivative, the learning rate of this element will drop faster. On the contrary, if the partial derivative of such an element remains small, then its learning rate will decline more slowly. However, since s_t accumulates the square by element gradient, the learning rate of each element in the independent variable declines (or remains unchanged) during iteration. Therefore, when the learning rate declines very fast during early iteration, yet the current solution is still not desirable, Adagrad might have difficulty finding a useful solution because the learning rate will be too small at later stages of iteration.

Below we will continue to use the objective function $f(x) = 0.1x_1^2 + 2x_2^2$ as an example to observe the iterative trajectory of the independent variable in Adagrad. We are going to implement Adagrad using the same learning rate as the experiment in last section, 0.4. As we can see, the iterative trajectory of the independent variable is smoother. However, due to the cumulative effect of s_t, the learning rate continuously decays, so the independent variable does not move as much during later stages of iteration.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        import math
        from mxnet import nd

        def adagrad_2d(x1, x2, s1, s2):
```

```
        # The first two terms are the independent variable gradients
        g1, g2, eps = 0.2 * x1, 4 * x2, 1e-6
        s1 += g1 ** 2
        s2 += g2 ** 2
        x1 -= eta / math.sqrt(s1 + eps) * g1
        x2 -= eta / math.sqrt(s2 + eps) * g2
        return x1, x2, s1, s2

    def f_2d(x1, x2):
        return 0.1 * x1 ** 2 + 2 * x2 ** 2

    eta = 0.4
    d2l.show_trace_2d(f_2d, d2l.train_2d(adagrad_2d))
```

```
epoch 20, x1 -2.382563, x2 -0.158591
```
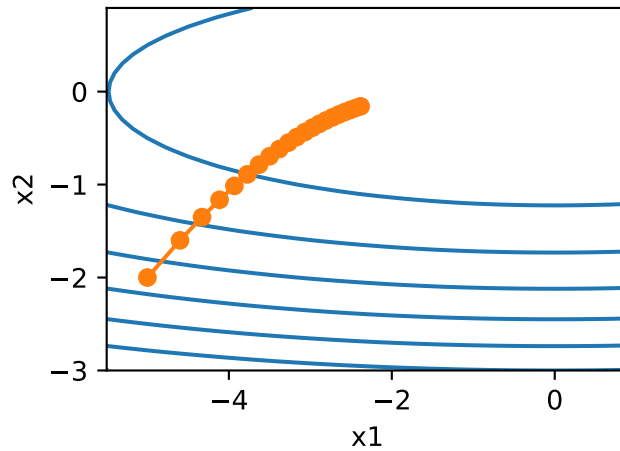

Now, we are going to increase the learning rate to 2. As we can see, the independent variable approaches the optimal solution more quickly.

```
In [2]: eta = 2
        d2l.show_trace_2d(f_2d, d2l.train_2d(adagrad_2d))
```

```
epoch 20, x1 -0.002295, x2 -0.000000
```

8. Optimization Algorithms
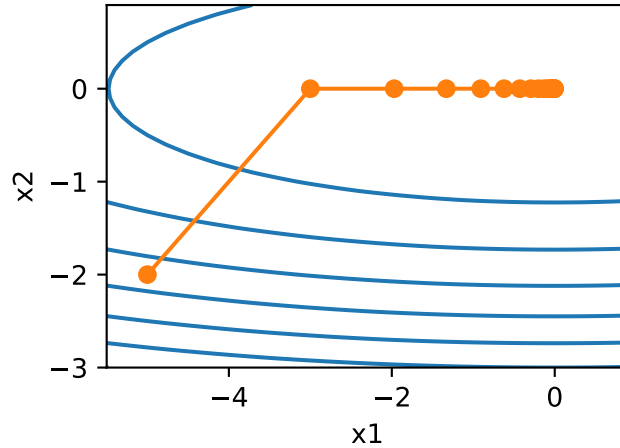
8.5.3 Implementation from Scratch

Like the momentum method, Adagrad needs to maintain a state variable of the same shape for each independent variable. We use the formula from the algorithm to implement Adagrad.

```
In [3]: features, labels = d2l.get_data_ch7()

        def init_adagrad_states():
            s_w = nd.zeros((features.shape[1], 1))
            s_b = nd.zeros(1)
            return (s_w, s_b)

        def adagrad(params, states, hyperparams):
            eps = 1e-6
            for p, s in zip(params, states):
                s[:] += p.grad.square()
                p[:] -= hyperparams['lr'] * p.grad / (s + eps).sqrt()
```

Compared with the experiment in the *Mini-Batch Stochastic Gradient Descent* section, here, we use a larger learning rate to train the model.

```
In [4]: d2l.train_ch7(adagrad, init_adagrad_states(), {'lr': 0.1}, features, labels)

loss: 0.243752, 0.348521 sec per epoch
```
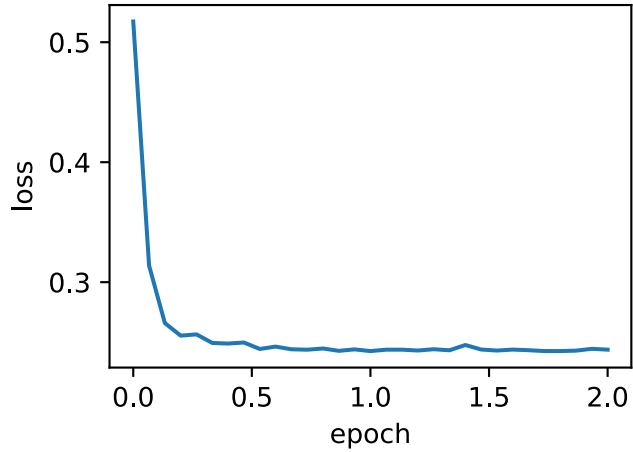
8.5.4 Concise Implementation

Using the `Trainer` instance of the algorithm named adagrad, we can implement the Adagrad algorithm with Gluon to train models.

```
In [5]: d2l.train_gluon_ch7('adagrad', {'learning_rate': 0.1}, features, labels)
```

```
loss: 0.243160, 0.414894 sec per epoch
```
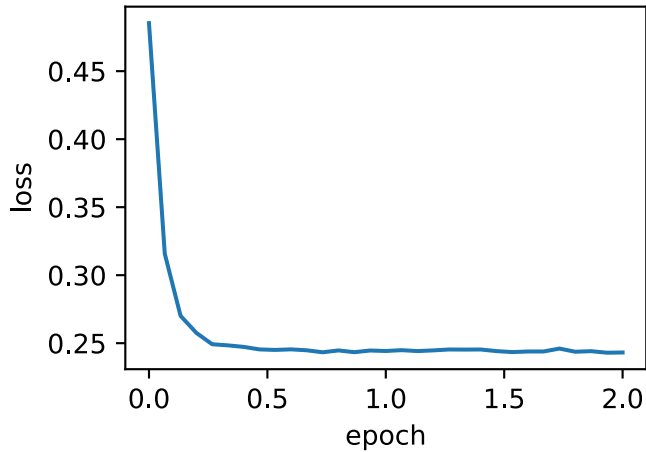
Summary

- Adagrad constantly adjusts the learning rate during iteration to give each element in the independent variable of the objective function its own learning rate.

- When using Adagrad, the learning rate of each element in the independent variable decreases (or remains unchanged) during iteration.

Exercises

- When introducing the features of Adagrad, we mentioned a potential problem. What solutions can you think of to fix this problem?

- Try to use other initial learning rates in the experiment. How does this change the results?

Reference

[1] Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. Journal of Machine Learning Research, 12(Jul), 2121-2159.

Scan the QR Code to Discuss

8.6 RMSProp

In the experiment in the *Adagrad* section, the learning rate of each element in the independent variable of the objective function declines (or remains unchanged) during iteration because the variable s_t in the denominator is increased by the square by element operation of the mini-batch stochastic gradient, adjusting the learning rate. Therefore, when the learning rate declines very fast during early iteration, yet the current solution is still not desirable, Adagrad might have difficulty finding a useful solution because the learning rate will be too small at later stages of iteration. To tackle this problem, the RMSProp algorithm made a small modification to Adagrad[1].

8.6.1 The Algorithm

We introduced EWMA (exponentially weighted moving average) in the *Momentum* section. Unlike in Adagrad, the state variable s_t is the sum of the square by element all the mini-batch stochastic gradients g_t up to the time step t, RMSProp uses the EWMA on the square by element results of these gradients. Specifically, given the hyperparameter $0 \leq \gamma < 1$, RMSProp is computed at time step $t > 0$.

$$s_t \leftarrow \gamma s_{t-1} + (1 - \gamma) g_t \odot g_t.$$

Like Adagrad, RMSProp re-adjusts the learning rate of each element in the independent variable of the objective function with element operations and then updates the independent variable.

$$x_t \leftarrow x_{t-1} - \frac{\eta}{\sqrt{s_t + \epsilon}} \odot g_t,$$

Here, eta is the learning rate while ϵ is a constant added to maintain numerical stability, such as 10^{-6}. Because the state variable of RMSProp is an EWMA of the squared term $g_t \odot g_t$, it can be seen as the weighted average of the mini-batch stochastic gradient's squared terms from the last $1/(1-\gamma)$ time steps. Therefore, the learning rate of each element in the independent variable will not always decline (or remain unchanged) during iteration.

By convention, we will use the objective function $f(x) = 0.1x_1^2 + 2x_2^2$ to observe the iterative trajectory of the independent variable in RMSProp. Recall that in the *Adagrad* section, when we used Adagrad with a learning rate of 0.4, the independent variable moved less in later stages of iteration. However, at the same learning rate, RMSProp can approach the optimal solution faster.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        import math
        from mxnet import nd

        def rmsprop_2d(x1, x2, s1, s2):
            g1, g2, eps = 0.2 * x1, 4 * x2, 1e-6
            s1 = gamma * s1 + (1 - gamma) * g1 ** 2
            s2 = gamma * s2 + (1 - gamma) * g2 ** 2
            x1 -= eta / math.sqrt(s1 + eps) * g1
            x2 -= eta / math.sqrt(s2 + eps) * g2
            return x1, x2, s1, s2

        def f_2d(x1, x2):
            return 0.1 * x1 ** 2 + 2 * x2 ** 2

        eta, gamma = 0.4, 0.9
        d2l.show_trace_2d(f_2d, d2l.train_2d(rmsprop_2d))

epoch 20, x1 -0.010599, x2 0.000000
```
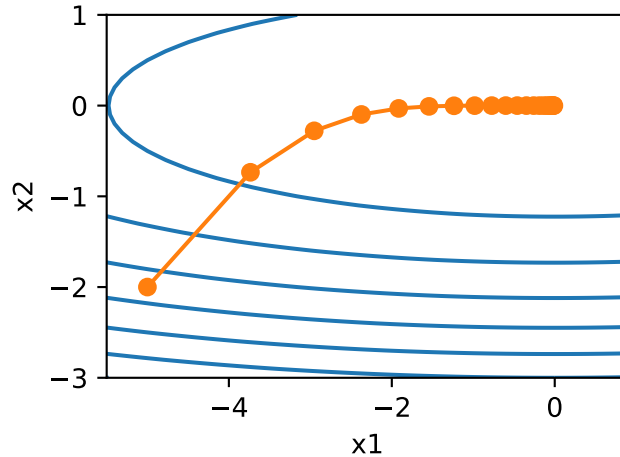
8.6.2 Implementation from Scratch

Next, we implement RMSProp with the formula in the algorithm.

```
In [2]:
        features, labels = d2l.get_data_ch7()
        def init_rmsprop_states():
            s_w = nd.zeros((features.shape[1], 1))
            s_b = nd.zeros(1)
            return (s_w, s_b)

        def rmsprop(params, states, hyperparams):
            gamma, eps = hyperparams['gamma'], 1e-6
            for p, s in zip(params, states):
                s[:] = gamma * s + (1 - gamma) * p.grad.square()
                p[:] -= hyperparams['lr'] * p.grad / (s + eps).sqrt()
```

We set the initial learning rate to 0.01 and the hyperparameter γ to 0.9. Now, the variable s_t can be treated as the weighted average of the square term $g_t \odot g_t$ from the last $1/(1-0.9) = 10$ time steps.

```
In [3]: d2l.train_ch7(rmsprop, init_rmsprop_states(), {'lr': 0.01, 'gamma': 0.9},
                        features, labels)

loss: 0.243551, 0.394985 sec per epoch
```
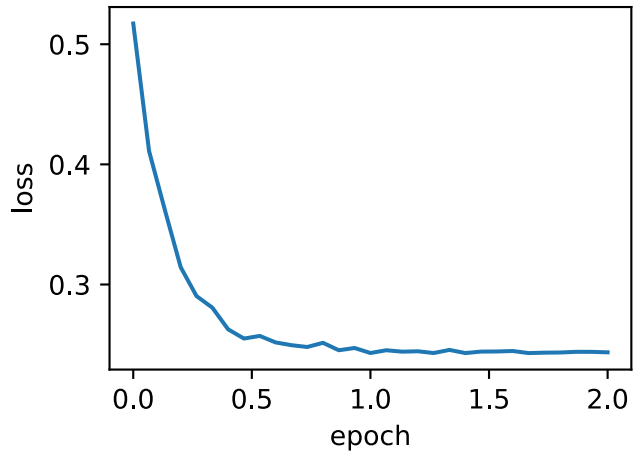
8.6.3 Concise Implementation

From the `Trainer` instance of the algorithm named rmsprop, we can implement the RMSProp algorithm with Gluon to train models. Note that the hyperparameter γ is assigned by `gamma1`.

```
In [4]: d2l.train_gluon_ch7('rmsprop', {'learning_rate': 0.01, 'gamma1': 0.9},
                            features, labels)

loss: 0.243954, 0.247184 sec per epoch
```
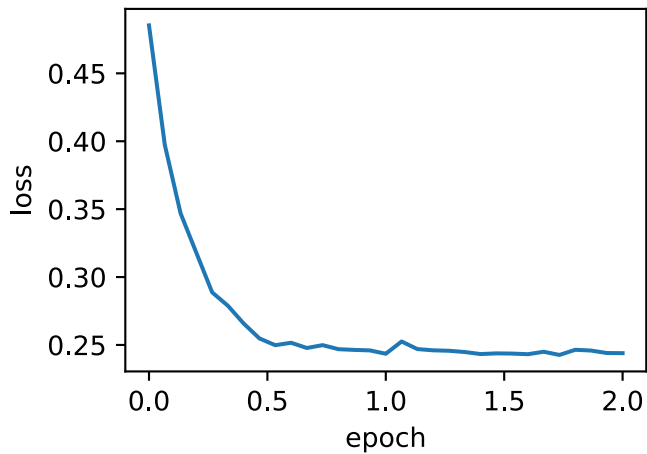
Summary

- The difference between RMSProp and Adagrad is that RMSProp uses an EWMA on the squares of elements in the mini-batch stochastic gradient to adjust the learning rate.

Exercises

- What happens to the experimental results if we set the value of γ to 1? Why?

- Try using other combinations of initial learning rates and γ hyperparameters and observe and analyze the experimental results.

Reference

[1] Tieleman, T., & Hinton, G. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural networks for machine learning, 4(2), 26-31.

Scan the QR Code to Discuss

8.7 Adadelta

In addition to RMSProp, Adadelta is another common optimization algorithm that helps improve the chances of finding useful solutions at later stages of iteration, which is difficult to do when using the Adagrad algorithm for the same purpose[1]. The interesting thing is that there is no learning rate hyperparameter in the Adadelta algorithm.

8.7.1 The Algorithm

Like RMSProp, the Adadelta algorithm uses the variable s_t, which is an EWMA on the squares of elements in mini-batch stochastic gradient g_t. At time step 0, all the elements are initialized to 0. Given the hyperparameter $0 \leq \rho < 1$ (counterpart of γ in RMSProp), at time step $t > 0$, compute using the same method as RMSProp:

$$s_t \leftarrow \rho s_{t-1} + (1 - \rho) g_t \odot g_t.$$

Unlike RMSProp, Adadelta maintains an additional state variable, $\Delta \boldsymbol{x}_t$ the elements of which are also initialized to 0 at time step 0. We use $\Delta \boldsymbol{x}_{t-1}$ to compute the variation of the independent variable:

$$\boldsymbol{g}_t' \leftarrow \sqrt{\frac{\Delta \boldsymbol{x}_{t-1} + \epsilon}{\boldsymbol{s}_t + \epsilon}} \odot \boldsymbol{g}_t,$$

Here, ϵ is a constant added to maintain the numerical stability, such as 10^{-5}. Next, we update the independent variable:

$$\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} - \boldsymbol{g}_t'.$$

Finally, we use $\Delta \boldsymbol{x}$ to record the EWMA on the squares of elements in $\boldsymbol{g}'$, which is the variation of the independent variable.

$$\Delta \boldsymbol{x}_t \leftarrow \rho \Delta \boldsymbol{x}_{t-1} + (1 - \rho)\boldsymbol{g}_t' \odot \boldsymbol{g}_t'.$$

As we can see, if the impact of ϵ is not considered here, Adadelta differs from RMSProp in its replacement of the hyperparameter η with $\sqrt{\Delta \boldsymbol{x}_{t-1}}$.

8.7.2 Implementation from Scratch

Adadelta needs to maintain two state variables for each independent variable, $\boldsymbol{s}_t$ and $\Delta \boldsymbol{x}_t$. We use the formula from the algorithm to implement Adadelta.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet import nd

        features, labels = d2l.get_data_ch7()

        def init_adadelta_states():
            s_w, s_b = nd.zeros((features.shape[1], 1)), nd.zeros(1)
            delta_w, delta_b = nd.zeros((features.shape[1], 1)), nd.zeros(1)
            return ((s_w, delta_w), (s_b, delta_b))

        def adadelta(params, states, hyperparams):
            rho, eps = hyperparams['rho'], 1e-5
            for p, (s, delta) in zip(params, states):
                s[:] = rho * s + (1 - rho) * p.grad.square()
                g = ((delta + eps).sqrt() / (s + eps).sqrt()) * p.grad
                p[:] -= g
                delta[:] = rho * delta + (1 - rho) * g * g
```

Then, we train the model with the hyperparameter $\rho = 0.9$.

```
In [2]: d2l.train_ch7(adadelta, init_adadelta_states(), {'rho': 0.9}, features,
                      labels)
```

8. Optimization Algorithms

```
loss: 0.244183, 0.461075 sec per epoch
```
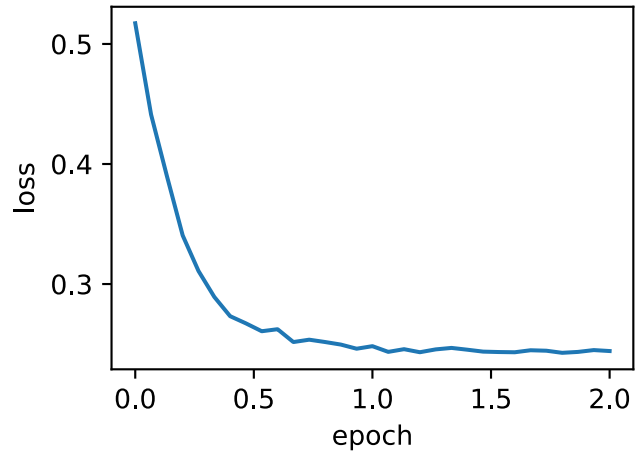

8.7.3 Concise Implementation

From the `Trainer` instance for the algorithm named adadelta, we can implement Adadelta in Gluon. Its hyperparameters can be specified by `rho`.

```
In [3]: d2l.train_gluon_ch7('adadelta', {'rho': 0.9}, features, labels)
```

```
loss: 0.242890, 0.426978 sec per epoch
```
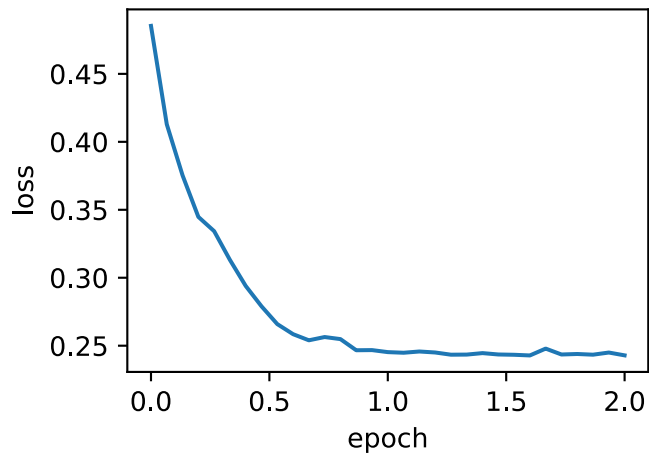
Summary

- Adadelta has no learning rate hyperparameter, it uses an EWMA on the squares of elements in the variation of the independent variable to replace the learning rate.

Exercises

- Adjust the value of ρ and observe the experimental results.

Reference

[1] Zeiler, M. D. (2012). ADADELTA: an adaptive learning rate method. arXiv preprint arXiv:1212.5701.

Scan the QR Code to Discuss

8.8 Adam

Created on the basis of RMSProp, Adam also uses EWMA on the mini-batch stochastic gradient[1]. Here, we are going to introduce this algorithm.

8.8.1 The Algorithm

Adam uses the momentum variable v_t and variable s_t, which is an EWMA on the squares of elements in the mini-batch stochastic gradient from RMSProp, and initializes each element of the variables to 0 at time step 0. Given the hyperparameter $0 \leq \beta_1 < 1$ (the author of the algorithm suggests a value of 0.9), the momentum variable v_t at time step t is the EWMA of the mini-batch stochastic gradient g_t:

$$v_t \leftarrow \beta_1 v_{t-1} + (1 - \beta_1)g_t.$$

Just as in RMSProp, given the hyperparameter $0 \leq \beta_2 < 1$ (the author of the algorithm suggests a value of 0.999), After taken the squares of elements in the mini-batch stochastic gradient, find $g_t \odot g_t$ and

perform EWMA on it to obtain s_t:

$$s_t \leftarrow \beta_2 s_{t-1} + (1 - \beta_2) g_t \odot g_t.$$

Since we initialized elements in v_0 and s_0 to 0, we get $v_t = (1 - \beta_1) \sum_{i=1}^{t} \beta_1^{t-i} g_i$ at time step t. Sum the mini-batch stochastic gradient weights from each previous time step to get $(1 - \beta_1) \sum_{i=1}^{t} \beta_1^{t-i} = 1 - \beta_1^t$. Notice that when t is small, the sum of the mini-batch stochastic gradient weights from each previous time step will be small. For example, when $\beta_1 = 0.9$, $v_1 = 0.1 g_1$. To eliminate this effect, for any time step t, we can divide v_t by $1 - \beta_1^t$, so that the sum of the mini-batch stochastic gradient weights from each previous time step is 1. This is also called bias correction. In the Adam algorithm, we perform bias corrections for variables v_t and s_t:

$$\hat{v}_t \leftarrow \frac{v_t}{1 - \beta_1^t},$$

$$\hat{s}_t \leftarrow \frac{s_t}{1 - \beta_2^t}.$$

Next, the Adam algorithm will use the bias-corrected variables $\hat{v}_t$ and $\hat{s}_t$ from above to re-adjust the learning rate of each element in the model parameters using element operations.

$$g_t' \leftarrow \frac{\eta \hat{v}_t}{\sqrt{\hat{s}_t} + \epsilon},$$

Here, eta is the learning rate while ϵ is a constant added to maintain numerical stability, such as 10^{-8}. Just as for Adagrad, RMSProp, and Adadelta, each element in the independent variable of the objective function has its own learning rate. Finally, use g_t' to iterate the independent variable:

$$x_t \leftarrow x_{t-1} - g_t'.$$

8.8.2 Implementation from Scratch

We use the formula from the algorithm to implement Adam. Here, time step t uses `hyperparams` to input parameters to the `adam` function.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet import nd

        features, labels = d2l.get_data_ch7()

        def init_adam_states():
            v_w, v_b = nd.zeros((features.shape[1], 1)), nd.zeros(1)
            s_w, s_b = nd.zeros((features.shape[1], 1)), nd.zeros(1)
            return ((v_w, s_w), (v_b, s_b))
```

```
def adam(params, states, hyperparams):
    beta1, beta2, eps = 0.9, 0.999, 1e-6
    for p, (v, s) in zip(params, states):
        v[:] = beta1 * v + (1 - beta1) * p.grad
        s[:] = beta2 * s + (1 - beta2) * p.grad.square()
        v_bias_corr = v / (1 - beta1 ** hyperparams['t'])
        s_bias_corr = s / (1 - beta2 ** hyperparams['t'])
        p[:] -= hyperparams['lr'] * v_bias_corr / (s_bias_corr.sqrt() + eps)
    hyperparams['t'] += 1
```

Use Adam to train the model with a learning rate of 0.01.

```
In [2]: d2l.train_ch7(adam, init_adam_states(), {'lr': 0.01, 't': 1}, features,
                      labels)
```

loss: 0.243627, 0.535598 sec per epoch
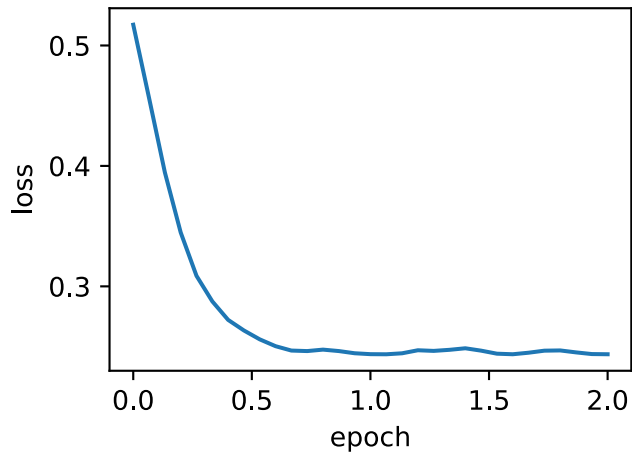
8.8.3 Concise Implementation

From the `Trainer` instance of the algorithm named adam, we can implement Adam with Gluon.

```
In [3]: d2l.train_gluon_ch7('adam', {'learning_rate': 0.01}, features, labels)
```

loss: 0.245230, 0.237746 sec per epoch
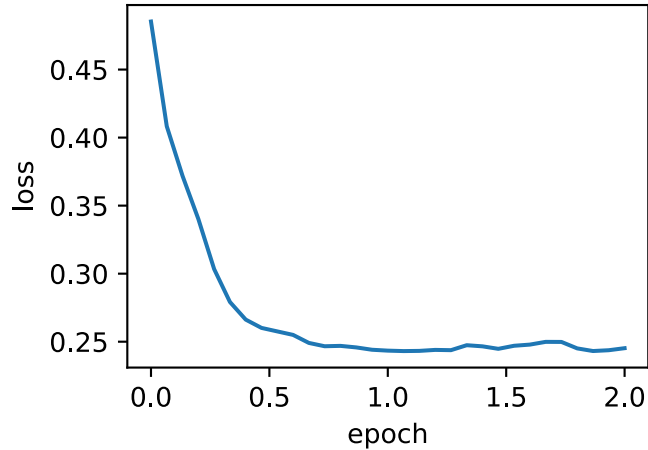
Summary

- Created on the basis of RMSProp, Adam also uses EWMA on the mini-batch stochastic gradient
- Adam uses bias correction.

Exercises

- Adjust the learning rate and observe and analyze the experimental results.
- Some people say that Adam is a combination of RMSProp and momentum. Why do you think they say this?

Reference

[1] Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

Scan the QR Code to Discuss

8. Optimization Algorithms

9

Computational Performance

In deep learning, data sets are usually large and model computation is complex. Therefore, we are always very concerned about computing performance. This chapter will focus on the important factors that affect computing performance: imperative programming, symbolic programming, asynchronous programing, automatic parallel computation, and multi-GPU computation. By studying this chapter, you should be able to further improve the computing performance of the models that have been implemented in the previous chapters, for example, by reducing the model training time without affecting the accuracy of the model.

9.1 A Hybrid of Imperative and Symbolic Programming

So far, this book has focused on imperative programming, which makes use of programming statements to change a program's state. Consider the following example of simple imperative programming code.

```
In [1]: def add(a, b):
            return a + b

        def fancy_func(a, b, c, d):
            e = add(a, b)
            f = add(c, d)
            g = add(e, f)
            return g

        fancy_func(1, 2, 3, 4)
```

```
Out[1]: 10
```

As expected, Python will perform an addition when running the statement `e = add(a, b)`, and will store the result as the variable `e`, thereby changing the program's state. The next two statements `f = add(c, d)` and `g = add(e, f)` will similarly perform additions and store the results as variables.

Although imperative programming is convenient, it may be inefficient. On the one hand, even if the `add` function is repeatedly called throughout the `fancy_func` function, Python will execute the three function calling statements individually, one after the other. On the other hand, we need to save the variable values of `e` and `f` until all the statements in `fancy_func` have been executed. This is because we do not know whether the variables `e` and `f` will be used by other parts of the program after the statements `e = add(a, b)` and `f = add(c, d)` have been executed.

Contrary to imperative programming, symbolic programming is usually performed after the computational process has been fully defined. Symbolic programming is used by multiple deep learning frameworks, including Theano and TensorFlow. The process of symbolic programming generally requires the following three steps:

1. Define the computation process.
2. Compile the computation process into an executable program.
3. Provide the required inputs and call on the compiled program for execution.

In the example below, we utilize symbolic programming to re-implement the imperative programming code provided at the beginning of this section.

```python
In [2]: def add_str():
            return '''
        def add(a, b):
            return a + b
        '''

        def fancy_func_str():
            return '''
        def fancy_func(a, b, c, d):
            e = add(a, b)
            f = add(c, d)
            g = add(e, f)
            return g
        '''

        def evoke_str():
            return add_str() + fancy_func_str() + '''
        print(fancy_func(1, 2, 3, 4))
        '''

        prog = evoke_str()
        print(prog)
        y = compile(prog, '', 'exec')
        exec(y)


def add(a, b):
    return a + b
```

```
def fancy_func(a, b, c, d):
    e = add(a, b)
    f = add(c, d)
    g = add(e, f)
    return g

print(fancy_func(1, 2, 3, 4))

10
```

The three functions defined above will only return the results of the computation process as a string. Finally, the complete computation process is compiled and run using the `compile` function. This leaves more room to optimize computation, since the system is able to view the entire program during its compilation. For example, during compilation, the program can be rewritten as `print((1 + 2) + (3 + 4))` or even directly rewritten as `print(10)`. Apart from reducing the amount of function calls, this process also saves memory.

A comparison of these two programming methods shows that

- imperative programming is easier. When imperative programming is used in Python, the majority of the code is straightforward and easy to write. At the same time, it is easier to debug imperative programming code. This is because it is easier to obtain and print all relevant intermediate variable values, or make use of Python's built-in debugging tools.

- Symbolic programming is more efficient and easier to port. Symbolic programming makes it easier to better optimize the system during compilation, while also having the ability to port the program into a format independent of Python. This allows the program to be run in a non-Python environment, thus avoiding any potential performance issues related to the Python interpreter.

9.1.1 Hybrid programming provides the best of both worlds.

Most deep learning frameworks choose either imperative or symbolic programming. For example, both Theano and TensorFlow (inspired by the latter) make use of symbolic programming, while Chainer and PyTorch utilize imperative programming. When designing Gluon, developers considered whether it was possible to harness the benefits of both imperative and symbolic programming. The developers believed that users should be able to develop and debug using pure imperative programming, while having the ability to convert most programs into symbolic programming to be run when product-level computing performance and deployment are required This was achieved by Gluon through the introduction of hybrid programming.

In hybrid programming, we can build models using either the HybridBlock or the HybridSequential classes. By default, they are executed in the same way Block or Sequential classes are executed in imperative programming. When the `hybridize` function is called, Gluon will convert the program's execution into the style used in symbolic programming. In fact, most models can make use of hybrid programming's execution style.

Through the use of experiments, this section will demonstrate the benefits of hybrid programming.

9.1.2 Constructing Models Using the HybridSequential Class

Previously, we learned how to use the Sequential class to concatenate multiple layers. Next, we will replace the Sequential class with the HybridSequential class in order to make use of hybrid programming.

```
In [3]: from mxnet import nd, sym
        from mxnet.gluon import nn
        import time

        def get_net():
            net = nn.HybridSequential()  # Here we use the class HybridSequential
            net.add(nn.Dense(256, activation='relu'),
                    nn.Dense(128, activation='relu'),
                    nn.Dense(2))
            net.initialize()
            return net

        x = nd.random.normal(shape=(1, 512))
        net = get_net()
        net(x)
Out[3]:
        [[0.08827581 0.00505182]]
        <NDArray 1x2 @cpu(0)>
```

By calling the `hybridize` function, we are able to compile and optimize the computation of the concatenation layer in the HybridSequential instance. The model's computation result remains unchanged.

```
In [4]: net.hybridize()
        net(x)
Out[4]:
        [[0.08827581 0.00505182]]
        <NDArray 1x2 @cpu(0)>
```

It should be noted that only the layers inheriting the HybridBlock class will be optimized during computation. For example, the HybridSequential and `Dense` classes provided by Gluon are all subclasses of HybridBlock class, meaning they will both be optimized during computation. A layer will not be optimized if it inherits from the Block class rather than the HybridBlock class.

Computing Performance

To demonstrate the performance improvement gained by the use of symbolic programming, we will compare the computation time before and after calling the `hybridize` function. Here we time 1000 `net` model computations. The model computations are based on imperative and symbolic programming, respectively, before and after `net` has called the `hybridize` function.

```
In [5]: def benchmark(net, x):
            start = time.time()
            for i in range(1000):
                _ = net(x)
            # To facilitate timing, we wait for all computations to be completed
            nd.waitall()
            return time.time() - start
```

```
        net = get_net()
        print('before hybridizing: %.4f sec' % (benchmark(net, x)))
        net.hybridize()
        print('after hybridizing: %.4f sec' % (benchmark(net, x)))

before hybridizing: 0.3207 sec
after hybridizing: 0.2592 sec
```

As is observed in the above results, after a HybridSequential instance calls the `hybridize` function, computing performance is improved through the use of symbolic programming.

Achieving Symbolic Programming

We can save the symbolic program and model parameters to the hard disk through the use of the `export` function after the `net` model has finished computing the output based on the input, such as in the case of `net(x)` in the `benchmark` function.

```
In [6]: net.export('my_mlp')
```

The .json and .params files generated during this process are a symbolic program and a model parameter, respectively. They can be read by other front-end languages supported by Python or MXNet, such as C++, R, Scala, and Perl. This allows us to deploy trained models to other devices and easily use other front-end programming languages. At the same time, because symbolic programming was used during deployment, the computing performance is often superior to that based on imperative programming.

In MXNet, a symbolic program refers to a program that makes use of the Symbol type. We know that, when the NDArray input `x` is provided to `net`, `net(x)` will directly calculate the model output and return a result based on `x`. For models that have called the `hybridize` function, we can also provide a Symbol-type input variable, and `net(x)` will return Symbol type results.

```
In [7]: x = sym.var('data')
        net(x)

Out[7]: <Symbol dense5_fwd>
```

9.1.3 Constructing Models Using the HybridBlock Class

Similar to the correlation between the Sequential Block classes, the HybridSequential class is a HybridBlock subclass. Contrary to the Block instance, which needs to use the `forward` function, for a HybridBlock instance we need to use the `hybrid_forward` function.

Earlier, we demonstrated that, after calling the `hybridize` function, the model is able to achieve superior computing performance and portability. In addition, model flexibility can be affected after calling the `hybridize` function. We will demonstrate this by constructing a model using the HybridBlock class.

```
In [8]: class HybridNet(nn.HybridBlock):
            def __init__(self, **kwargs):
                super(HybridNet, self).__init__(**kwargs)
                self.hidden = nn.Dense(10)
                self.output = nn.Dense(2)
```

```
        def hybrid_forward(self, F, x):
            print('F: ', F)
            print('x: ', x)
            x = F.relu(self.hidden(x))
            print('hidden: ', x)
            return self.output(x)
```

We need to add the additional input F to the `hybrid_forward` function when inheriting the Hybrid-Block class. We already know that MXNet uses both an NDArray class and a Symbol class, which are based on imperative programming and symbolic programming, respectively. Since these two classes perform very similar functions, MXNet will determine whether F will call NDArray or Symbol based on the input provided.

The following creates a HybridBlock instance. As we can see, by default, F uses NDArray. We also printed out the x input as well as the hidden layer's output using the ReLU activation function.

```
In [9]: net = HybridNet()
        net.initialize()
        x = nd.random.normal(shape=(1, 4))
        net(x)

F:  <module 'mxnet.ndarray' from '/var/lib/jenkins/miniconda2/envs/d2l-en-build/lib/py⌴
 ↪  thon3.6/site-packages/mxnet/ndarray/__init__.py'>
x:
[[-0.12225834  0.5429998  -0.9469352   0.59643304]]
<NDArray 1x4 @cpu(0)>
hidden:
[[0.11134676 0.04770704 0.05341475 0.        0.08091211 0.
  0.         0.04143535 0.        0.        ]]
<NDArray 1x10 @cpu(0)>

Out[9]:
        [[0.00370749 0.00134991]]
        <NDArray 1x2 @cpu(0)>
```

Repeating the forward computation will achieve the same results.

```
In [10]: net(x)

F:  <module 'mxnet.ndarray' from '/var/lib/jenkins/miniconda2/envs/d2l-en-build/lib/py⌴
 ↪  thon3.6/site-packages/mxnet/ndarray/__init__.py'>
x:
[[-0.12225834  0.5429998  -0.9469352   0.59643304]]
<NDArray 1x4 @cpu(0)>
hidden:
[[0.11134676 0.04770704 0.05341475 0.        0.08091211 0.
  0.         0.04143535 0.        0.        ]]
<NDArray 1x10 @cpu(0)>

Out[10]:
        [[0.00370749 0.00134991]]
        <NDArray 1x2 @cpu(0)>
```

Next, we will see what happens after we call the `hybridize` function.

```
In [11]: net.hybridize()
         net(x)
```

```
F:  <module 'mxnet.symbol' from '/var/lib/jenkins/miniconda2/envs/d2l-en-build/lib/pyt
 ↪  hon3.6/site-packages/mxnet/symbol/__init__.py'>
x:  <Symbol data>
hidden:  <Symbol hybridnet0_relu0>
```

```
Out[11]:
        [[0.00370749 0.00134991]]
        <NDArray 1x2 @cpu(0)>
```

We can see that `F` turns into a Symbol. Moreover, even though the input data is still NDArray, the same input and intermediate output will all be converted to Symbol type in the `hybrid_forward` function.

Now, we repeat the forward computation.

```
In [12]: net(x)
```

```
Out[12]:
        [[0.00370749 0.00134991]]
        <NDArray 1x2 @cpu(0)>
```

We can see that the three lines of print statements defined in the `hybrid_forward` function will not print anything. This is because a symbolic program has been produced since the last time `net(x)` was run by calling the `hybridize` function. Afterwards, when we run `net(x)` again, MXNet will no longer need to access Python code, but can directly perform symbolic programming at the C++ backend. This is another reason why model computing performance will be improve after the `hybridize` function is called. However, there is always the potential that any programs we write will suffer a loss in flexibility. If we want to use the three lines of print statements to debug the code in the above example, they will be skipped over and we would not be able to print when the symbolic program is executed. Additionally, in the case of a few functions not supported by Symbol (like `asnumpy`), and operations in-place like `a  +=  b` and `a[:]  = a + b` (must be rewritten as `a = a + b`). Therefore, we will not be able to use the `hybrid_forward` function or perform forward computation after the `hybridize` function has been called.

Summary

- Both imperative and symbolic programming have their advantages as well as their disadvantages. Through hybrid programming, MXNet is able to combine the advantages of both.

- Models constructed by the HybridSequential and HybridBlock classes are able to convert imperative program into symbolic program by calling the `hybridize` function. We recommend using this method to improve computing performance.

Exercises

- Add `x.asnumpy()` to the first line of the `hybrid_forward` function of the HybridNet class in this section, run all the code in this section, and observe any error types and locations

- What happens if we add the Python statements `if` and `for` in the `hybrid_forward` function?

- Review the models that interest you in the previous chapters and use the HybridBlock class or HybridSequential class to implement them.

Scan the QR Code to Discuss

9.2 Asynchronous Computing

MXNet utilizes asynchronous programming to improve computing performance. Understanding how asynchronous programming works helps us to develop more efficient programs, by proactively reducing computational requirements and thereby minimizing the memory overhead required in the case of limited memory resources. First, we will import the package or module needed for this section's experiment.

```
In [1]: from mxnet import autograd, gluon, nd
        from mxnet.gluon import loss as gloss, nn
        import os
        import subprocess
        import time
```

9.2.1 Asynchronous Programming in MXNet

Broadly speaking, MXNet includes the front-end directly used by users for interaction, as well as the back-end used by the system to perform the computation. For example, users can write MXNet programs in various front-end languages, such as Python, R, Scala and C++. Regardless of the front-end programming language used, the execution of MXNet programs occurs primarily in the back-end of C++ implementations. In other words, front-end MXNet programs written by users are passed on to the back-end to be computed. The back-end possesses its own threads that continuously collect and execute queued tasks.

Through the interaction between front-end and back-end threads, MXNet is able to implement asynchronous programming. Asynchronous programming means that the front-end threads continue to execute subsequent instructions without having to wait for the back-end threads to return the results from the current instruction. For simplicity's sake, assume that the Python front-end thread calls the following four instructions.

```
In [2]: a = nd.ones((1, 2))
        b = nd.ones((1, 2))
        c = a * b + 2
        c
```

```
Out[2]:
        [[3. 3.]]
        <NDArray 1x2 @cpu(0)>
```

In Asynchronous Computing, whenever the Python front-end thread executes one of the first three statements, it simply returns the task to the back-end queue. When the last statement's results need to be printed, the Python front-end thread will wait for the C++ back-end thread to finish computing result of the variable `c`. One benefit of such as design is that the Python front-end thread in this example does not need to perform actual computations. Thus, there is little impact on the program's overall performance, regardless of Python's performance. MXNet will deliver consistently high performance, regardless of the front-end language's performance, provided the C++ back-end can meet the efficiency requirements.

To further demonstrate the asynchronous computation's performance, we will implement a simple timing class.

```
In [3]: # This class has been saved in the Gluonbook module for future use
        class Benchmark():
            def __init__(self, prefix=None):
                self.prefix = prefix + ' ' if prefix else ''

            def __enter__(self):
                self.start = time.time()

            def __exit__(self, *args):
                print('%stime: %.4f sec' % (self.prefix, time.time() - self.start))
```

The following example uses timing to demonstrate the effect of asynchronous programming. As we can see, when `y = nd.dot(x, x).sum()` is returned, it does not actually wait for the variable `y` to be calculated. Only when the `print` function needs to print the variable `y` must the function wait for it to be calculated.

```
In [4]: with Benchmark('Workloads are queued.'):
            x = nd.random.uniform(shape=(2000, 2000))
            y = nd.dot(x, x).sum()

        with Benchmark('Workloads are finished.'):
            print('sum =', y)

Workloads are queued. time: 0.0007 sec
sum =
[2.0003661e+09]
<NDArray 1 @cpu(0)>
Workloads are finished. time: 0.1744 sec
```

In truth, whether or not the current result is already calculated in the memory is irrelevant, unless we need to print or save the computation results. So long as the data is stored in NDArray and the operators provided by MXNet are used, MXNet will utilize asynchronous programming by default to attain superior computing performance.

9.2.2 Use of the Synchronization Function to Allow the Front-End to Wait for the Computation Results

In addition to the `print` function we just introduced, there are other ways to make the front-end thread wait for the completion of the back-end computations. The `wait_to_read` function can be used to make the front-end wait for the complete computation of the NDArray results, and then execute following statement. Alternatively, we can use the `waitall` function to make the front-end wait for the completion of all previous computations. The latter is a common method used in performance testing.

Below, we use the `wait_to_read` function as an example. The time output includes the calculation time of `y`.

```
In [5]: with Benchmark():
            y = nd.dot(x, x)
            y.wait_to_read()

time: 0.0664 sec
```

Below, we use `waitall` as an example. The time output includes the calculation time of `y` and `z` respectively.

```
In [6]: with Benchmark():
            y = nd.dot(x, x)
            z = nd.dot(x, x)
            nd.waitall()

time: 0.1301 sec
```

Additionally, any operation that does not support asynchronous programming but converts the NDArray into another data structure will cause the front-end to have to wait for computation results. For example, calling the `asnumpy` and `asscalar` functions:

```
In [7]: with Benchmark():
            y = nd.dot(x, x)
            y.asnumpy()

time: 0.0683 sec
```

```
In [8]: with Benchmark():
            y = nd.dot(x, x)
            y.norm().asscalar()

time: 0.1597 sec
```

The `wait_to_read`, `waitall`, `asnumpy`, `asscalar` and the`print` functions described above will cause the front-end to wait for the back-end computation results. Such functions are often referred to as synchronization functions.

9.2.3 Using Asynchronous Programming to Improve Computing Performance

In the following example, we will use the for loop to continuously assign values to the variable `y`. Asynchronous programming is not used in tasks when the synchronization function `wait_to_read` is used

in the for loop. However, when the synchronization function `waitall` is used outside of the for loop, asynchronous programming is used.

```
In [9]: with Benchmark('synchronous.'):
            for _ in range(1000):
                y = x + 1
                y.wait_to_read()

        with Benchmark('asynchronous.'):
            for _ in range(1000):
                y = x + 1
            nd.waitall()

synchronous. time: 0.4397 sec
asynchronous. time: 0.3672 sec
```

We have observed that certain aspects of computing performance can be improved by making use of asynchronous programming. To explain this, we will slightly simplify the interaction between the Python front-end thread and the C++ back-end thread. In each loop, the interaction between front and back-ends can be largely divided into three stages:

1. The front-end orders the back-end to insert the calculation task `y = x + 1` into the queue.

2. The back-end then receives the computation tasks from the queue and performs the actual computations.

3. The back-end then returns the computation results to the front-end.

Assume that the durations of these three stages are t_1, t_2, t_3, respectively. If we do not use asynchronous programming, the total time taken to perform 1000 computations is approximately $1000(t_1 + t_2 + t_3)$. If asynchronous programming is used, the total time taken to perform 1000 computations can be reduced to $t_1 + 1000t_2 + t_3$ (assuming $1000t_2 > 999t_1$), since the front-end does not have to wait for the back-end to return computation results for each loop.

9.2.4 The Impact of Asynchronous Programming on Memory

In order to explain the impact of asynchronous programming on memory usage, recall what we learned in the previous chapters. Throughout the model training process implemented in the previous chapters, we usually evaluated things like the loss or accuracy of the model in each mini-batch. Detail-oriented readers may have discovered that such evaluations often make use of synchronization functions, such as `asscalar` or `asnumpy`. If these synchronization functions are removed, the front-end will pass a large number of mini-batch computing tasks to the back-end in a very short time, which might cause a spike in memory usage. When the mini-batches makes use of synchronization functions, on each iteration, the front-end will only pass one mini-batch task to the back-end to be computed, which will typically reduce memory use.

Because the deep learning model is usually large and memory resources are usually limited, we recommend the use of synchronization functions for each mini-batch throughout model training, for example by using the `asscalar` or `asnumpy` functions to evaluate model performance. Similarly, we also rec-

ommend utilizing synchronization functions for each mini-batch prediction (such as directly printing out the current batch's prediction results), in order to reduce memory usage during model prediction.

Next, we will demonstrate asynchronous programming's impact on memory. We will first define a data retrieval function `data_iter`, which upon being called, will start timing and regularly print out the time taken to retrieve data batches.

```
In [10]: def data_iter():
             start = time.time()
             num_batches, batch_size = 100, 1024
             for i in range(num_batches):
                 X = nd.random.normal(shape=(batch_size, 512))
                 y = nd.ones((batch_size,))
                 yield X, y
                 if (i + 1) % 50 == 0:
                     print('batch %d, time %f sec' % (i + 1, time.time() - start))
```

The multilayer perceptron, optimization algorithm, and loss function are defined below.

```
In [11]: net = nn.Sequential()
         net.add(nn.Dense(2048, activation='relu'),
                 nn.Dense(512, activation='relu'),
                 nn.Dense(1))
         net.initialize()
         trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.005})
         loss = gloss.L2Loss()
```

A helper function to monitor memory use is defined here. It should be noted that this function can only be run on Linux or MacOS operating systems.

```
In [12]: def get_mem():
             res = subprocess.check_output(['ps', 'u', '-p', str(os.getpid())])
             return int(str(res).split()[15]) / 1e3
```

Now we can begin testing. To initialize the `net` parameters we will try running the system once. See the section *Deferred Initialization of Model Parameters* for further discussions related to initialization.

```
In [13]: for X, y in data_iter():
             break
         loss(y, net(X)).wait_to_read()
```

For the `net` training model, the synchronization function `asscalar` can naturally be used to record the loss of each mini-batch output by the NDArray format and to print out the model loss after each iteration. At this point, the generation interval of each mini-batch increases, but with a small memory overhead.

```
In [14]: l_sum, mem = 0, get_mem()
         for X, y in data_iter():
             with autograd.record():
                 l = loss(y, net(X))
             # Use of the Asscalar synchronization function
             l_sum += l.mean().asscalar()
             l.backward()
             trainer.step(X.shape[0])
         nd.waitall()
         print('increased memory: %f MB' % (get_mem() - mem))
```

```
batch 50, time 3.369981 sec
batch 100, time 6.768061 sec
increased memory: 3.056000 MB
```

Even though each mini-batch's generation interval is shorter, the memory usage may still be high during training if the synchronization function is removed. This is because, in default asynchronous programming, the front-end will pass on all mini-batch computations to the back-end in a short amount of time. As a result of this, a large amount of intermediate results cannot be released and may end up piled up in memory. In this experiment, we can see that all data (X and y) is generated in under a second. However, because of an insufficient training speed, this data can only be stored in the memory and cannot be cleared in time, resulting in extra memory usage.

```
In [15]: mem = get_mem()
         for X, y in data_iter():
             with autograd.record():
                 l = loss(y, net(X))
             l.backward()
             trainer.step(x.shape[0])
         nd.waitall()
         print('increased memory: %f MB' % (get_mem() - mem))

batch 50, time 0.077338 sec
batch 100, time 0.152817 sec
increased memory: 198.904000 MB
```

Summary

- MXNet includes the front-end used directly by users for interaction and the back-end used by the system to perform the computation.

- MXNet can improve computing performance through the use of asynchronous programming.

- We recommend using at least one synchronization function for each mini-batch training or prediction to avoid passing on too many computation tasks to the back-end in a short period of time.

Exercises

- In the section Use of Asynchronous Programming to Improve Computing Performance, we mentioned that using asynchronous computation can reduce the total amount of time needed to perform 1000 computations to $t_1 + 1000t_2 + t_3$. Why do we have to assume $1000t_2 > 999t_1$ here?

9.3 Automatic Parallelism

MXNet automatically constructs computational graphs at the back end. Using a computational graph, the system is aware of all the computational dependencies, and can selectively execute multiple non-interdependent tasks in parallel to improve computing performance. For instance, the first example in the *Asynchronous Computing* section executes `a = nd.ones((1, 2))` and `b = nd.ones((1, 2))` in turn. There is no dependency between these two steps, so the system can choose to execute them in parallel.

Typically, a single operator will use all the computational resources on all CPUs or a single GPU. For example, the `dot` operator will use all threads on all CPUs (even if there are multiple CPU processors on a single machine) or a single GPU. If computational load of each operator is large enough and multiple operators are run in parallel on only on the CPU or a single GPU, then the operations of each operator can only receive a portion of computational resources of CPU or single GPU. Even if these computations can be parallelized, the ultimate increase in computing performance may not be significant. In this section, our discussion of automatic parallel computation mainly focuses on parallel computation using both CPUs and GPUs, as well as the parallelization of computation and communication.

First, import the required packages or modules for experiment in this section. Note that we need at least one GPU to run the experiment in this section.

```
In [1]: import sys
        sys.path.insert(0, '..')

        import d2l
        import mxnet as mx
        from mxnet import nd
```

9.3.1 Parallel Computation using CPUs and GPUs

First, we will discuss parallel computation using CPUs and GPUs, for example, when computation in a program occurs both on the CPU and a GPU. First, define the `run` function so that it performs 10 matrix multiplications.

```
In [2]: def run(x):
            return [nd.dot(x, x) for _ in range(10)]
```

Next, create an NDArray on both the CPU and GPU.

```
In [3]: x_cpu = nd.random.uniform(shape=(2000, 2000))
        x_gpu = nd.random.uniform(shape=(6000, 6000), ctx=mx.gpu(0))
```

Then, use the two NDArrays to run the run function on both the CPU and GPU and print the time required.

```
In [4]: run(x_cpu)  # Warm-up begins
        run(x_gpu)
        nd.waitall()  # Warm-up ends

        with d2l.Benchmark('Run on CPU.'):
            run(x_cpu)
            nd.waitall()

        with d2l.Benchmark('Then run on GPU.'):
            run(x_gpu)
            nd.waitall()

Run on CPU. time: 0.6221 sec
Then run on GPU. time: 1.2154 sec
```

We remove nd.waitall() between the two computing tasks run(x_cpu) and run(x_gpu) and hope the system can automatically parallel these two tasks.

```
In [5]: with d2l.Benchmark('Run on both CPU and GPU in parallel.'):
            run(x_cpu)
            run(x_gpu)
            nd.waitall()

Run on both CPU and GPU in parallel. time: 1.2210 sec
```

As we can see, when two computing tasks are executed together, the total execution time is less than the sum of their separate execution times. This means that MXNet can effectively automate parallel computation on CPUs and GPUs.

9.3.2 Parallel Computation of Computing and Communication

In computations that use both the CPU and GPU, we often need to copy data between the CPU and GPU, resulting in data communication. In the example below, we compute on the GPU and then copy the results back to the CPU. We print the GPU computation time and the communication time from the GPU to CPU.

```
In [6]: def copy_to_cpu(x):
            return [y.copyto(mx.cpu()) for y in x]

        with d2l.Benchmark('Run on GPU.'):
            y = run(x_gpu)
            nd.waitall()

        with d2l.Benchmark('Then copy to CPU.'):
            copy_to_cpu(y)
            nd.waitall()

Run on GPU. time: 1.2194 sec
Then copy to CPU. time: 0.5218 sec
```

We remove the `waitall` function between computation and communication and print the total time need to complete both tasks.

```
In [7]: with d2l.Benchmark('Run and copy in parallel.'):
            y = run(x_gpu)
            copy_to_cpu(y)
            nd.waitall()

Run and copy in parallel. time: 1.2736 sec
```

As we can see, the total time required to perform computation and communication is less than the sum of their separate execution times. It should be noted that this computation and communication task is different from the parallel computation task that simultaneously used the CPU and GPU described earlier in this section. Here, there is a dependency between execution and communication: `y[i]` must be computed before it can be copied to the CPU. Fortunately, the system can copy `y[i-1]` when computing `y[i]` to reduce the total running time of computation and communication.

Summary

- MXNet can improve computing performance through automatic parallel computation, such as parallel computation using the CPU and GPU and the parallelization of computation and communication.

Exercises

- 10 operations were performed in the `run` function defined in this section. There are no dependencies between them. Design an experiment to see if MXNet will automatically execute them in parallel.

- Designing computation tasks that include more complex data dependencies, and run experiments to see if MXNet can obtain the correct results and improve computing performance.

- When the computational load of an operator is small enough, parallel computation on only the CPU or a single GPU may also improve the computing performance. Design an experiment to verify this.

Scan the QR Code to Discuss

9.4 Multi-GPU Computation Implementation from Scratch

In this section, we will show how to use multiple GPU for computation. For example, we can train the same model using multiple GPUs. As you might expect, running the programs in this section requires at least two GPUs. In fact, installing multiple GPUs on a single machine is common because there are usually multiple PCIe slots on the motherboard. If the NVIDIA driver is properly installed, we can use the `nvidia-smi` command to view all GPUs on the current computer.

```
In [1]: !nvidia-smi

Mon Mar  4 20:19:20 2019
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 396.37                 Driver Version: 396.37                     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Tesla M60           Off  | 00000000:00:1D.0 Off |                    0 |
| N/A   28C    P0    43W / 150W |      0MiB /  7618MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
|   1  Tesla M60           Off  | 00000000:00:1E.0 Off |                    0 |
| N/A   33C    P0    41W / 150W |      0MiB /  7618MiB |     97%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

As we discussed in the *Automatic Parallel Computation* section, most operations can use all the computational resources of all CPUs, or all computational resources of a single GPU. However, if we use multiple GPUs for model training, we still need to implement the corresponding algorithms. Of these, the most commonly used algorithm is called data parallelism.

9.4.1 Data Parallelism

In the deep learning field, Data Parallelism is currently the most widely used method for dividing model training tasks among multiple GPUs. Recall the process for training models using optimization algorithms described in the *Mini-batch Stochastic Gradient Descent* section. Now, we will demonstrate how data parallelism works using mini-batch stochastic gradient descent as an example.

Assume there are k GPUs on a machine. Given the model to be trained, each GPU will maintain a complete set of model parameters independently. In any iteration of model training, given a random mini-batch, we divide the examples in the batch into k portions and distribute one to each GPU. Then, each GPU will calculate the local gradient of the model parameters based on the mini-batch subset it was assigned and the model parameters it maintains. Next, we add together the local gradients on the k GPUs to get the current mini-batch stochastic gradient. After that, each GPU uses this mini-batch

stochastic gradient to update the complete set of model parameters that it maintains. Figure 8.1 depicts the mini-batch stochastic gradient calculation using data parallelism and two GPUs.
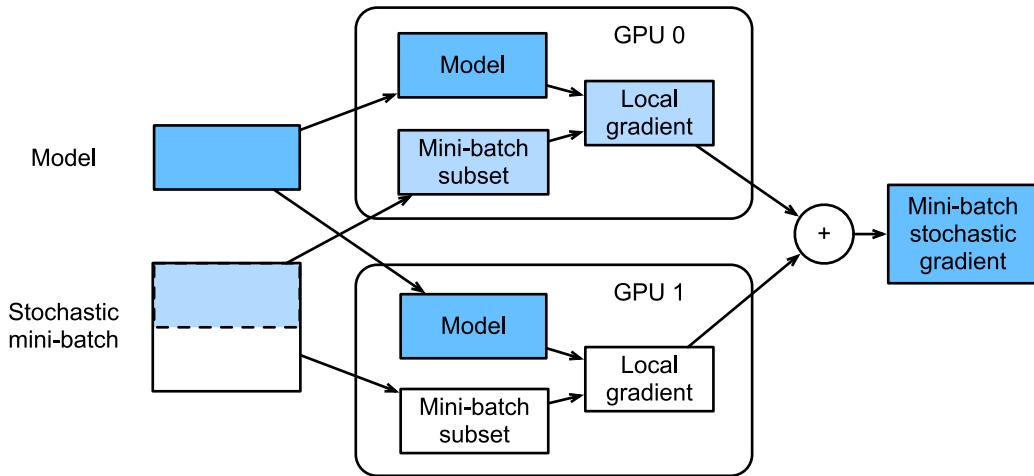
Fig. 9.1: Calculation of mini-batch stochastic gradient using data parallelism and two GPUs.

In order to implement data parallelism in a multi-GPU training scenario from scratch, we first import the required packages or modules.

```
In [2]: import sys
        sys.path.insert(0, '..')

        import d2l
        import mxnet as mx
        from mxnet import autograd, nd
        from mxnet.gluon import loss as gloss
        import time
```

9.4.2 Define the Model

We use LeNet, introduced in the *Convolutional Neural Networks (LeNet)* section, as the sample model for this section. Here, the model implementation only uses NDArray.

```
In [3]: # Initialize model parameters
        scale = 0.01
        W1 = nd.random.normal(scale=scale, shape=(20, 1, 3, 3))
        b1 = nd.zeros(shape=20)
        W2 = nd.random.normal(scale=scale, shape=(50, 20, 5, 5))
        b2 = nd.zeros(shape=50)
        W3 = nd.random.normal(scale=scale, shape=(800, 128))
        b3 = nd.zeros(shape=128)
        W4 = nd.random.normal(scale=scale, shape=(128, 10))
        b4 = nd.zeros(shape=10)
```

```
            params = [W1, b1, W2, b2, W3, b3, W4, b4]

            # Define the model
            def lenet(X, params):
                h1_conv = nd.Convolution(data=X, weight=params[0], bias=params[1],
                                         kernel=(3, 3), num_filter=20)
                h1_activation = nd.relu(h1_conv)
                h1 = nd.Pooling(data=h1_activation, pool_type='avg', kernel=(2, 2),
                                stride=(2, 2))
                h2_conv = nd.Convolution(data=h1, weight=params[2], bias=params[3],
                                         kernel=(5, 5), num_filter=50)
                h2_activation = nd.relu(h2_conv)
                h2 = nd.Pooling(data=h2_activation, pool_type='avg', kernel=(2, 2),
                                stride=(2, 2))
                h2 = nd.flatten(h2)
                h3_linear = nd.dot(h2, params[4]) + params[5]
                h3 = nd.relu(h3_linear)
                y_hat = nd.dot(h3, params[6]) + params[7]
                return y_hat

            # Cross-entropy loss function
            loss = gloss.SoftmaxCrossEntropyLoss()
```

9.4.3 Synchronize Data Among Multiple GPUs

We need to implement some auxiliary functions to synchronize data among the multiple GPUs. The following `get_params` function copies the model parameters to a specific GPU and initializes the gradient.

```
In [4]: def get_params(params, ctx):
            new_params = [p.copyto(ctx) for p in params]
            for p in new_params:
                p.attach_grad()
            return new_params
```

Try to copy the model parameter `params` to `gpu(0)`.

```
In [5]: new_params = get_params(params, mx.gpu(0))
        print('b1 weight:', new_params[1])
        print('b1 grad:', new_params[1].grad)

b1 weight:
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
<NDArray 20 @gpu(0)>
b1 grad:
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
<NDArray 20 @gpu(0)>
```

Here, the data is distributed among multiple GPUs. The following `allreduce` function adds up the data on each GPU and then broadcasts it to all the GPUs.

```
In [6]: def allreduce(data):
            for i in range(1, len(data)):
                data[0][:] += data[i].copyto(data[0].context)
```

```
            for i in range(1, len(data)):
                data[0].copyto(data[i])
```

Perform a simple test of the `allreduce` function.

```
In [7]: data = [nd.ones((1, 2), ctx=mx.gpu(i)) * (i + 1) for i in range(2)]
        print('before allreduce:', data)
        allreduce(data)
        print('after allreduce:', data)
```

```
before allreduce: [
[[1. 1.]]
<NDArray 1x2 @gpu(0)>,
[[2. 2.]]
<NDArray 1x2 @gpu(1)>]
after allreduce: [
[[3. 3.]]
<NDArray 1x2 @gpu(0)>,
[[3. 3.]]
<NDArray 1x2 @gpu(1)>]
```

Given a batch of data instances, the following `split_and_load` function can split the sample and copy it to each GPU.

```
In [8]: def split_and_load(data, ctx):
            n, k = data.shape[0], len(ctx)
            m = n // k  # For simplicity, we assume the data is divisible
            assert m * k == n, '# examples is not divided by # devices.'
            return [data[i * m: (i + 1) * m].as_in_context(ctx[i]) for i in range(k)]
```

Now, we try to divide the 6 data instances equally between 2 GPUs using the `split_and_load` function.

```
In [9]: batch = nd.arange(24).reshape((6, 4))
        ctx = [mx.gpu(0), mx.gpu(1)]
        splitted = split_and_load(batch, ctx)
        print('input: ', batch)
        print('load into', ctx)
        print('output:', splitted)
```

```
input:
[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]
 [12. 13. 14. 15.]
 [16. 17. 18. 19.]
 [20. 21. 22. 23.]]
<NDArray 6x4 @cpu(0)>
load into [gpu(0), gpu(1)]
output: [
[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]]
<NDArray 3x4 @gpu(0)>,
[[12. 13. 14. 15.]
 [16. 17. 18. 19.]
 [20. 21. 22. 23.]]
<NDArray 3x4 @gpu(1)>]
```

9.4.4 Multi-GPU Training on a Single Mini-batch

Now we can implement multi-GPU training on a single mini-batch. Its implementation is primarily based on the data parallelism approach described in this section. We will use the auxiliary functions we just discussed, `allreduce` and `split_and_load`, to synchronize the data among multiple GPUs.

```
In [10]: def train_batch(X, y, gpu_params, ctx, lr):
             # When ctx contains multiple GPUs, mini-batches of data instances are
             # divided and copied to each GPU
             gpu_Xs, gpu_ys = split_and_load(X, ctx), split_and_load(y, ctx)
             with autograd.record():  # Loss is calculated separately on each GPU
                 ls = [loss(lenet(gpu_X, gpu_W), gpu_y)
                       for gpu_X, gpu_y, gpu_W in zip(gpu_Xs, gpu_ys, gpu_params)]
             for l in ls:  # Back Propagation is performed separately on each GPU
                 l.backward()
             # Add up all the gradients from each GPU and then broadcast them to all
             # the GPUs
             for i in range(len(gpu_params[0])):
                 allreduce([gpu_params[c][i].grad for c in range(len(ctx))])
             # The model parameters are updated separately on each GPU
             for param in gpu_params:
                 d2l.sgd(param, lr, X.shape[0])  # Here, we use a full-size batch
```

9.4.5 Training Functions

Now, we can define the training function. Here the training function is slightly different from the one used in the previous chapter. For example, here, we need to copy all the model parameters to multiple GPUs based on data parallelism and perform multi-GPU training on a single mini-batch for each iteration.

```
In [11]: def train(num_gpus, batch_size, lr):
             train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
             ctx = [mx.gpu(i) for i in range(num_gpus)]
             print('running on:', ctx)
             # Copy model parameters to num_gpus GPUs
             gpu_params = [get_params(params, c) for c in ctx]
             for epoch in range(4):
                 start = time.time()
                 for X, y in train_iter:
                     # Perform multi-GPU training for a single mini-batch
                     train_batch(X, y, gpu_params, ctx, lr)
                     nd.waitall()
                 train_time = time.time() - start

                 def net(x):  # Verify the model on GPU 0
                     return lenet(x, gpu_params[0])

                 test_acc = d2l.evaluate_accuracy(test_iter, net, ctx[0])
                 print('epoch %d, time: %.1f sec, test acc: %.2f'
                       % (epoch + 1, train_time, test_acc))
```

9.4.6 Multi-GPU Training Experiment

We will start by training with a single GPU. Assume the batch size is 256 and the learning rate is 0.2.

```
In [12]: train(num_gpus=1, batch_size=256, lr=0.2)

running on: [gpu(0)]
epoch 1, time: 2.7 sec, test acc: 0.10
epoch 2, time: 2.2 sec, test acc: 0.67
epoch 3, time: 2.2 sec, test acc: 0.76
epoch 4, time: 2.2 sec, test acc: 0.73
```

By keeping the batch size and learning rate unchanged and changing the number of GPUs to 2, we can see that the improvement in test accuracy is roughly the same as in the results from the previous experiment. Because of the extra communication overhead, we did not observe a significant reduction in the training time.

```
In [13]: train(num_gpus=2, batch_size=256, lr=0.2)

running on: [gpu(0), gpu(1)]
epoch 1, time: 2.5 sec, test acc: 0.10
epoch 2, time: 2.3 sec, test acc: 0.70
epoch 3, time: 2.6 sec, test acc: 0.76
epoch 4, time: 2.6 sec, test acc: 0.76
```

Summary

- We can use data parallelism to more fully utilize the computational resources of multiple GPUs to implement multi-GPU model training.

- With the same hyper-parameters, the training accuracy of the model is roughly equivalent when we change the number of GPUs.

Exercises

- In a multi-GPU training experiment, use 2 GPUs for training and double the `batch_size` to 512. How does the training time change? If we want a test accuracy comparable with the results of single-GPU training, how should the learning rate be adjusted?

- Change the model prediction part of the experiment to multi-GPU prediction.

Scan the QR Code to Discuss

9.5 Concise Implementation of Multi-GPU Computation

In Gluon, we can conveniently use data parallelism to perform multi-GPU computation. For example, we do not need to implement the helper function to synchronize data among multiple GPUs, as described in the *Multi-GPU Computation* section, ourselves.

First, import the required packages or modules for the experiment in this section. Running the programs in this section requires at least two GPUs.

```
In [1]: import sys
        sys.path.insert(0, '..')

        import d2l
        import mxnet as mx
        from mxnet import autograd, gluon, init, nd
        from mxnet.gluon import loss as gloss, nn, utils as gutils
        import time
```

9.5.1 Initialize Model Parameters on Multiple GPUs

In this section, we use ResNet-18 as a sample model. Since the input images in this section are original size (not enlarged), the model construction here is different from the ResNet-18 structure described in the *ResNet* section. This model uses a smaller convolution kernel, stride, and padding at the beginning and removes the maximum pooling layer.

```
In [2]: # This function is saved in the d2l package for future use
        def resnet18(num_classes):
            def resnet_block(num_channels, num_residuals, first_block=False):
                blk = nn.Sequential()
                for i in range(num_residuals):
                    if i == 0 and not first_block:
                        blk.add(d2l.Residual(
                            num_channels, use_1x1conv=True, strides=2))
                    else:
                        blk.add(d2l.Residual(num_channels))
                return blk

            net = nn.Sequential()
            # This model uses a smaller convolution kernel, stride, and padding and
            # removes the maximum pooling layer
            net.add(nn.Conv2D(64, kernel_size=3, strides=1, padding=1),
                    nn.BatchNorm(), nn.Activation('relu'))
            net.add(resnet_block(64, 2, first_block=True),
                    resnet_block(128, 2),
                    resnet_block(256, 2),
                    resnet_block(512, 2))
            net.add(nn.GlobalAvgPool2D(), nn.Dense(num_classes))
            return net

        net = resnet18(10)
```

Previously, we discussed how to use the `initialize` function's `ctx` parameter to initialize model

---

parameters on a CPU or a single GPU. In fact, `ctx` can accept a range of CPUs and GPUs so as to copy initialized model parameters to all CPUs and GPUs in `ctx`.

```
In [3]: ctx = [mx.gpu(0), mx.gpu(1)]
        net.initialize(init=init.Normal(sigma=0.01), ctx=ctx)
```

Gluon provides the `split_and_load` function implemented in the previous section. It can divide a mini-batch of data instances and copy them to each CPU or GPU. Then, the model computation for the data input to each CPU or GPU occurs on that same CPU or GPU.

```
In [4]: x = nd.random.uniform(shape=(4, 1, 28, 28))
        gpu_x = gutils.split_and_load(x, ctx)
        net(gpu_x[0]), net(gpu_x[1])
Out[4]: (
          [[ 5.4814936e-06 -8.3371094e-07 -1.6316770e-06 -6.3674099e-07
            -3.8216162e-06 -2.3514044e-06 -2.5469599e-06 -9.4784696e-08
            -6.9033558e-07  2.5756231e-06]
           [ 5.4710872e-06 -9.4246496e-07 -1.0494070e-06  9.8081841e-08
            -3.3251815e-06 -2.4862918e-06 -3.3642798e-06  1.0455864e-07
            -6.1001344e-07  2.0327841e-06]]
          <NDArray 2x10 @gpu(0)>,
          [[ 5.6176345e-06 -1.2837586e-06 -1.4605541e-06  1.8302967e-07
            -3.5511653e-06 -2.4371013e-06 -3.5731798e-06 -3.0974860e-07
            -1.1016571e-06  1.8909889e-06]
           [ 5.1418697e-06 -1.3729932e-06 -1.1520088e-06  1.1507450e-07
            -3.7372811e-06 -2.8289724e-06 -3.6477197e-06  1.5781629e-07
            -6.0733043e-07  1.9712013e-06]]
          <NDArray 2x10 @gpu(1)>)
```

Now we can access the initialized model parameter values through `data`. It should be noted that `weight.data()` will return the parameter values on the CPU by default. Since we specified 2 GPUs to initialize the model parameters, we need to specify the GPU to access parameter values. As we can see, the same parameters have the same values on different GPUs.

```
In [5]: weight = net[0].params.get('weight')

        try:
            weight.data()
        except RuntimeError:
            print('not initialized on', mx.cpu())
        weight.data(ctx[0])[0], weight.data(ctx[1])[0]

not initialized on cpu(0)

Out[5]: (
          [[[-0.01473444 -0.01073093 -0.01042483]
            [-0.01327885 -0.01474966 -0.00524142]
            [ 0.01266256  0.00895064 -0.00601594]]]
          <NDArray 1x3x3 @gpu(0)>,
          [[[-0.01473444 -0.01073093 -0.01042483]
            [-0.01327885 -0.01474966 -0.00524142]
            [ 0.01266256  0.00895064 -0.00601594]]]
          <NDArray 1x3x3 @gpu(1)>)
```

9.5.2 Multi-GPU Model Training

When we use multiple GPUs to train the model, the `Trainer` instance will automatically perform data parallelism, such as dividing mini-batches of data instances and copying them to individual GPUs and summing the gradients of each GPU and broadcasting the result to all GPUs. In this way, we can easily implement the training function.

```
In [6]: def train(num_gpus, batch_size, lr):
            train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
            ctx = [mx.gpu(i) for i in range(num_gpus)]
            print('running on:', ctx)
            net.initialize(init=init.Normal(sigma=0.01), ctx=ctx, force_reinit=True)
            trainer = gluon.Trainer(
                net.collect_params(), 'sgd', {'learning_rate': lr})
            loss = gloss.SoftmaxCrossEntropyLoss()
            for epoch in range(4):
                start = time.time()
                for X, y in train_iter:
                    gpu_Xs = gutils.split_and_load(X, ctx)
                    gpu_ys = gutils.split_and_load(y, ctx)
                    with autograd.record():
                        ls = [loss(net(gpu_X), gpu_y)
                                for gpu_X, gpu_y in zip(gpu_Xs, gpu_ys)]
                    for l in ls:
                        l.backward()
                    trainer.step(batch_size)
                nd.waitall()
                train_time = time.time() - start
                test_acc = d2l.evaluate_accuracy(test_iter, net, ctx[0])
                print('epoch %d, time: %.1f sec, test acc %.2f' % (
                    epoch + 1, train_time, test_acc))
```

First, use a single GPU for training.

```
In [7]: train(num_gpus=1, batch_size=256, lr=0.1)

running on: [gpu(0)]
epoch 1, time: 63.8 sec, test acc 0.88
epoch 2, time: 59.9 sec, test acc 0.90
epoch 3, time: 59.9 sec, test acc 0.92
epoch 4, time: 60.0 sec, test acc 0.92
```

Then we try to use 2 GPUs for training. Compared with the LeNet used in the previous section, ResNet-18 computing is more complicated and the communication time is shorter compared to the calculation time, so parallel computing in ResNet-18 better improves performance.

```
In [8]: train(num_gpus=2, batch_size=512, lr=0.2)

running on: [gpu(0), gpu(1)]
epoch 1, time: 31.9 sec, test acc 0.79
epoch 2, time: 30.7 sec, test acc 0.86
epoch 3, time: 30.6 sec, test acc 0.85
epoch 4, time: 30.8 sec, test acc 0.77
```

Summary

- In Gluon, we can conveniently perform multi-GPU computations, such as initializing model parameters and training models on multiple GPUs.

Exercises

- This section uses ResNet-18. Try different epochs, batch sizes, and learning rates. Use more GPUs for computation if conditions permit.

- Sometimes, different devices provide different computing power. Some can use CPUs and GPUs at the same time, or GPUs of different models. How should we divide mini-batches among different CPUs or GPUs?

Scan the QR Code to Discuss

10

Computer Vision

Many applications in the area of computer vision are closely related to our daily lives, now and in the future, whether medical diagnostics, driverless vehicles, camera monitoring, or smart filters. In recent years, deep learning technology has greatly enhanced computer vision systems' performance. It can be said that the most advanced computer vision applications are nearly inseparable from deep learning.

We have introduced deep learning models commonly used in the area of computer vision in the chapter Convolutional Neural Networks and have practiced simple image classification tasks. In this chapter, we will introduce image augmentation and fine tuning methods and apply them to image classification. Then, we will explore various methods of object detection. After that, we will learn how to use fully convolutional networks to perform semantic segmentation on images. Then, we explain how to use style transfer technology to generate images that look like the cover of this book. Finally, we will perform practice exercises on two important computer vision data sets to review the content of this chapter and the previous chapters.

10.1 Image Augmentation

We mentioned that large-scale data sets are prerequisites for the successful application of deep neural networks in the *Deep Convolutional Neural Networks (AlexNet)* section. Image augmentation technology expands the scale of training data sets by making a series of random changes to the training images to produce similar, but different, training examples. Another way to explain image augmentation is that randomly changing training examples can reduce a model's dependence on certain properties, thereby improving its capability for generalization. For example, we can crop the images in different ways, so

that the objects of interest appear in different positions, reducing the model's dependence on the position where objects appear. We can also adjust the brightness, color, and other factors to reduce model's sensitivity to color. It can be said that image augmentation technology contributed greatly to the success of AlexNet. In this section we will discuss this technology, which is widely used in computer vision.

First, import the packages or modules required for the experiment in this section.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        import mxnet as mx
        from mxnet import autograd, gluon, image, init, nd
        from mxnet.gluon import data as gdata, loss as gloss, utils as gutils
        import sys
        import time
```

10.1.1 Common Image Augmentation Method

In this experiment, we will use an image with a shape of 400×500 as an example.

```
In [2]: d2l.set_figsize()
        img = image.imread('../img/cat1.jpg')
        d2l.plt.imshow(img.asnumpy())

Out[2]: <matplotlib.image.AxesImage at 0x7f2fc4841898>
```
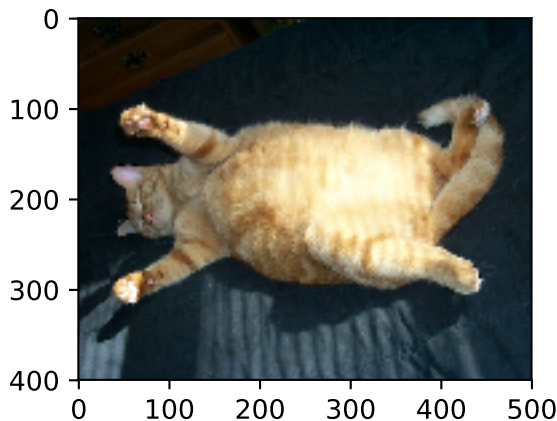

The drawing function `show_images` is defined below.

```
In [3]: # This function is saved in the d2l package for future use
        def show_images(imgs, num_rows, num_cols, scale=2):
            figsize = (num_cols * scale, num_rows * scale)
            _, axes = d2l.plt.subplots(num_rows, num_cols, figsize=figsize)
            for i in range(num_rows):
                for j in range(num_cols):
```

```
                axes[i][j].imshow(imgs[i * num_cols + j].asnumpy())
                axes[i][j].axes.get_xaxis().set_visible(False)
                axes[i][j].axes.get_yaxis().set_visible(False)
        return axes
```

Most image augmentation methods have a certain degree of randomness. To make it easier for us to observe the effect of image augmentation, we next define the auxiliary function `apply`. This function runs the image augmentation method `aug` multiple times on the input image `img` and shows all results.

```
In [4]: def apply(img, aug, num_rows=2, num_cols=4, scale=1.5):
            Y = [aug(img) for _ in range(num_rows * num_cols)]
            show_images(Y, num_rows, num_cols, scale)
```

Flip and Crop

Flipping the image left and right usually does not change the category of the object. This is one of the earliest and most widely used methods of image augmentation. Next, we use the `transforms` module to create the `RandomFlipLeftRight` instance, which introduces a 50% chance that the image is flipped left and right.

```
In [5]: apply(img, gdata.vision.transforms.RandomFlipLeftRight())
```
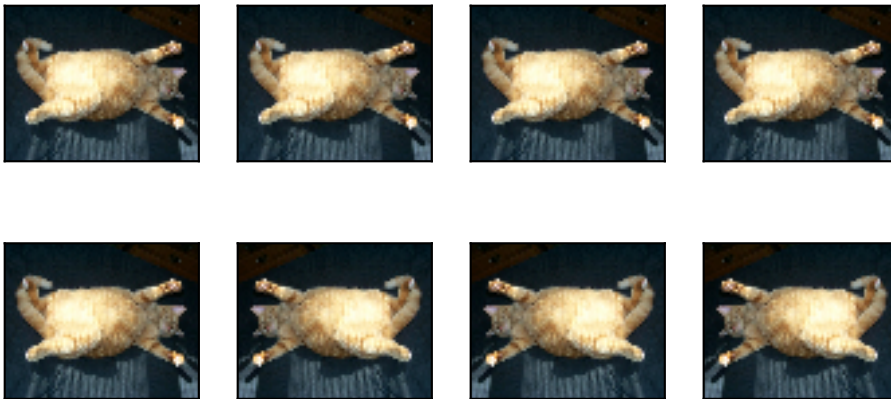

Flipping up and down is not as commonly used as flipping left and right. However, at least for this example image, flipping up and down does not hinder recognition. Next, we create a `RandomFlipTopBottom` instance for a 50% chance of flipping the image up and down.

```
In [6]: apply(img, gdata.vision.transforms.RandomFlipTopBottom())
```

In the example image we used, the cat is in the middle of the image, but this may not be the case for all images. In the *Pooling Layer* section, we explained that the pooling layer can reduce the sensitivity of the convolutional layer to the target location. In addition, we can make objects appear at different positions in the image in different proportions by randomly cropping the image. This can also reduce the sensitivity of the model to the target position.
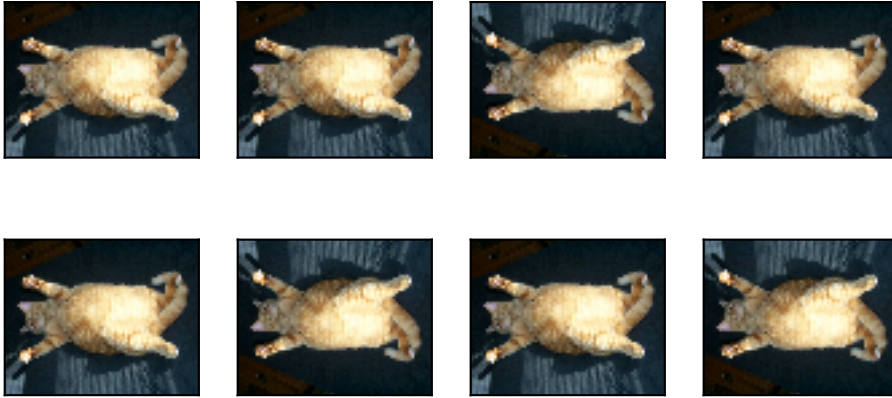
In the following code, we randomly crop a region with an area of 10% to 100% of the original area, and the ratio of width to height of the region is randomly selected from between 0.5 and 2. Then, the width and height of the region are both scaled to 200 pixels. Unless otherwise stated, the random number between a and b in this section refers to a continuous value obtained by uniform sampling in the interval $[a, b]$.

```
In [7]: shape_aug = gdata.vision.transforms.RandomResizedCrop(
            (200, 200), scale=(0.1, 1), ratio=(0.5, 2))
        apply(img, shape_aug)
```

Change Color

Another augmentation method is changing colors. We can change four aspects of the image color: brightness, contrast, saturation, and hue. In the example below, we randomly change the brightness of the image to a value between 50% ($1 - 0.5$) and 150% ($1 + 0.5$) of the original image.

```
In [8]: apply(img, gdata.vision.transforms.RandomBrightness(0.5))
```
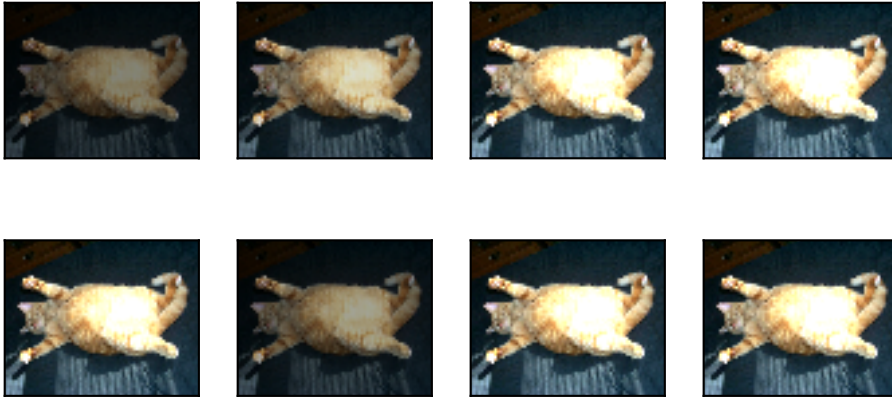

Similarly, we can randomly change the hue of the image.

```
In [9]: apply(img, gdata.vision.transforms.RandomHue(0.5))
```


We can also create a `RandomColorJitter` instance and set how to randomly change the `brightness`, `contrast`, `saturation`, and `hue` of the image at the same time.

```
In [10]: color_aug = gdata.vision.transforms.RandomColorJitter(
              brightness=0.5, contrast=0.5, saturation=0.5, hue=0.5)
         apply(img, color_aug)
```
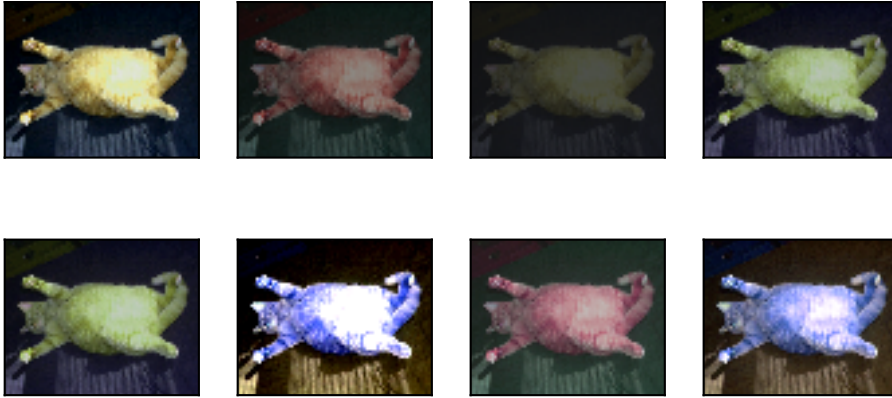
Overlying Multiple Image Augmentation Methods

In practice, we will overlay multiple image augmentation methods. We can overlay the different image augmentation methods defined above and apply them to each image by using a `Compose` instance.

```
In [11]: augs = gdata.vision.transforms.Compose([
            gdata.vision.transforms.RandomFlipLeftRight(), color_aug, shape_aug])
        apply(img, augs)
```
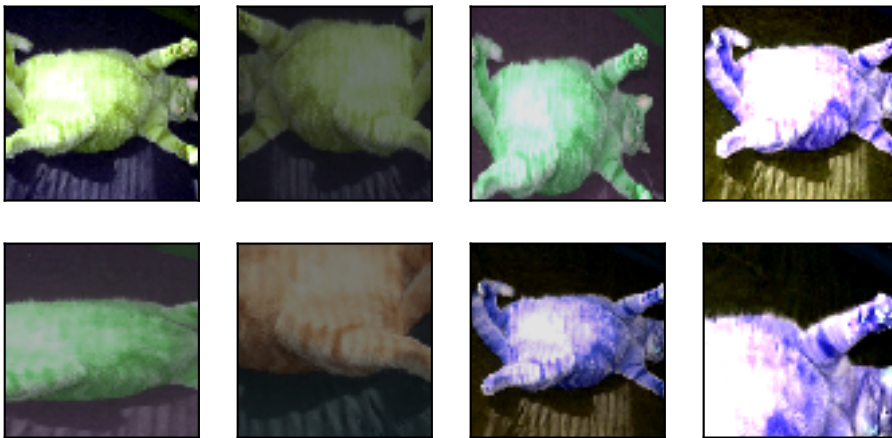

10.1.2 Using an Image Augmentation Training Model

Next, we will look at how to apply image augmentation in actual training. Here, we use the CIFAR-10 data set, instead of the Fashion-MNIST data set we have been using. This is because the position and size

of the objects in the Fashion-MNIST data set have been normalized, and the differences in color and size of the objects in CIFAR-10 data set are more significant. The first 32 training images in the CIFAR-10 data set are shown below.

```
In [12]: show_images(gdata.vision.CIFAR10(train=True)[0:32][0], 4, 8, scale=0.8);
```
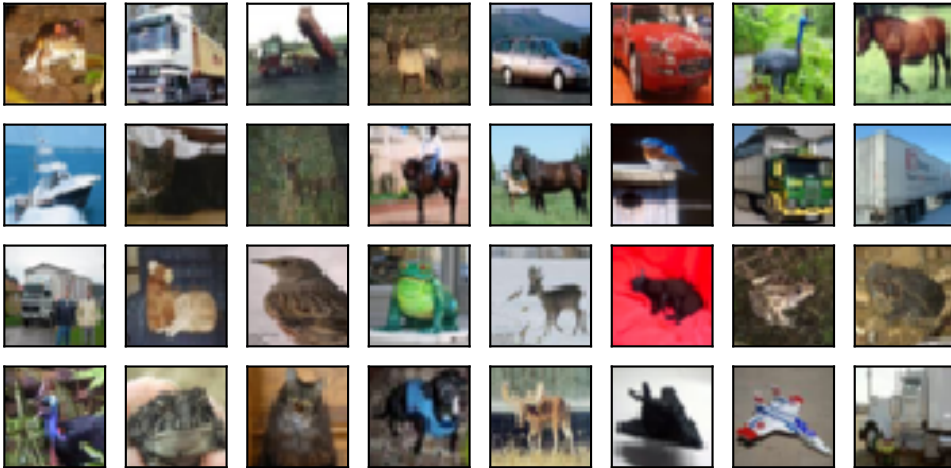

In order to obtain a definitive results during prediction, we usually only apply image augmentation to the training example, and do not use image augmentation with random operations during prediction. Here, we only use the simplest random left-right flipping method. In addition, we use a `ToTensor` instance to convert mini-batch images into the format required by MXNet, i.e. 32-bit floating point numbers with the shape of (batch size, number of channels, height, width) and value range between 0 and 1.

```
In [13]: train_augs = gdata.vision.transforms.Compose([
             gdata.vision.transforms.RandomFlipLeftRight(),
             gdata.vision.transforms.ToTensor()])

         test_augs = gdata.vision.transforms.Compose([
             gdata.vision.transforms.ToTensor()])
```

Next, we define an auxiliary function to make it easier to read the image and apply image augmentation. The `transform_first` function provided by Gluon's data set applies image augmentation to the first element of each training example (image and label), i.e., the element at the top of the image. For detailed description of `DataLoader`, refer to the previous Image Classification Data Set (Fashion-MNIST) section.

```
In [14]: num_workers = 0 if sys.platform.startswith('win32') else 4
         def load_cifar10(is_train, augs, batch_size):
             return gdata.DataLoader(
                 gdata.vision.CIFAR10(train=is_train).transform_first(augs),
                 batch_size=batch_size, shuffle=is_train, num_workers=num_workers)
```

Using a Multi-GPU Training Model

We train the ResNet-18 model described in *ResNet* section on the CIFAR-10 data set. We will also apply the methods described in the *Concise Implementation of Multi-GPU Computation* section, and use a multi-GPU training model.

First, we define the `try_all_gpus` function to get all available GPUs.

```
In [15]: # This function has been saved in the d2l package for future use
         def try_all_gpus():
             ctxes = []
             try:
                 # Assume that the number of GPUs on a machine does not exceed 16
                 for i in range(16):
                     ctx = mx.gpu(i)
                     _ = nd.array([0], ctx=ctx)
                     ctxes.append(ctx)
             except mx.base.MXNetError:
                 pass
             if not ctxes:
                 ctxes = [mx.cpu()]
             return ctxes
```

The auxiliary function `_get_batch` defined below divides the mini-batch data instance `batch` and copy the batches to each GPU contained in the `ctx` variable.

```
In [16]: def _get_batch(batch, ctx):
             features, labels = batch
             if labels.dtype != features.dtype:
                 labels = labels.astype(features.dtype)
             # When ctx contains multiple GPUs, mini-batch data instances are divided
             # and copied to each GPU.
             return (gutils.split_and_load(features, ctx),
                     gutils.split_and_load(labels, ctx), features.shape[0])
```

Then, we define the `evaluate_accuracy` function to evaluate the classification accuracy of the model. Different from `evaluate_accuracy`, the function described in the Softmax Regression Starting from Scratch and *Convolutional Neural Network (LeNet)* sections, the function defined here are more general. It evaluates the model using all GPUs contained in the `ctx` variable by using the auxiliary function `_get_batch`.

```
In [17]: # This function has been saved in the d2l package for future use
         def evaluate_accuracy(data_iter, net, ctx=[mx.cpu()]):
             if isinstance(ctx, mx.Context):
                 ctx = [ctx]
             acc_sum, n = nd.array([0]), 0
             for batch in data_iter:
                 features, labels, _ = _get_batch(batch, ctx)
                 for X, y in zip(features, labels):
                     y = y.astype('float32')
                     acc_sum += (net(X).argmax(axis=1) == y).sum().copyto(mx.cpu())
                     n += y.size
                 acc_sum.wait_to_read()
             return acc_sum.asscalar() / n
```

Next, we define the `train` function to train and evaluate the model using multiple GPUs.

```
In [18]: # This function has been saved in the d2l package for future use
         def train(train_iter, test_iter, net, loss, trainer, ctx, num_epochs):
             print('training on', ctx)
             if isinstance(ctx, mx.Context):
                 ctx = [ctx]
             for epoch in range(num_epochs):
                 train_l_sum, train_acc_sum, n, m, start = 0.0, 0.0, 0, 0, time.time()
                 for i, batch in enumerate(train_iter):
                     Xs, ys, batch_size = _get_batch(batch, ctx)
                     ls = []
                     with autograd.record():
                         y_hats = [net(X) for X in Xs]
                         ls = [loss(y_hat, y) for y_hat, y in zip(y_hats, ys)]
                     for l in ls:
                         l.backward()
                     trainer.step(batch_size)
                     train_l_sum += sum([l.sum().asscalar() for l in ls])
                     n += sum([l.size for l in ls])
                     train_acc_sum += sum([(y_hat.argmax(axis=1) == y).sum().asscalar()
                                          for y_hat, y in zip(y_hats, ys)])
                     m += sum([y.size for y in ys])
                 test_acc = evaluate_accuracy(test_iter, net, ctx)
                 print('epoch %d, loss %.4f, train acc %.3f, test acc %.3f, '
                       'time %.1f sec'
                       % (epoch + 1, train_l_sum / n, train_acc_sum / m, test_acc,
                          time.time() - start))
```

Now, we can define the `train_with_data_aug` function to use image augmentation to train the model. This function obtains all available GPUs and uses Adam as the optimization algorithm for training. It then applies image augmentation to the training data set, and finally calls the `train` function just defined to train and evaluate the model.

```
In [19]: def train_with_data_aug(train_augs, test_augs, lr=0.001):
             batch_size, ctx, net = 256, try_all_gpus(), d2l.resnet18(10)
             net.initialize(ctx=ctx, init=init.Xavier())
             trainer = gluon.Trainer(net.collect_params(), 'adam',
                                    {'learning_rate': lr})
             loss = gloss.SoftmaxCrossEntropyLoss()
             train_iter = load_cifar10(True, train_augs, batch_size)
             test_iter = load_cifar10(False, test_augs, batch_size)
             train(train_iter, test_iter, net, loss, trainer, ctx, num_epochs=10)
```

Now we train the model using image augmentation of random flipping left and right.

```
In [20]: train_with_data_aug(train_augs, test_augs)

training on [gpu(0), gpu(1)]
epoch 1, loss 1.3633, train acc 0.517, test acc 0.559, time 39.2 sec
epoch 2, loss 0.8142, train acc 0.712, test acc 0.739, time 35.0 sec
epoch 3, loss 0.6017, train acc 0.789, test acc 0.746, time 35.1 sec
epoch 4, loss 0.4842, train acc 0.833, test acc 0.777, time 35.0 sec
epoch 5, loss 0.4084, train acc 0.858, test acc 0.800, time 35.1 sec
epoch 6, loss 0.3388, train acc 0.882, test acc 0.833, time 35.1 sec
epoch 7, loss 0.2827, train acc 0.903, test acc 0.793, time 35.1 sec
epoch 8, loss 0.2421, train acc 0.915, test acc 0.833, time 35.2 sec
```

```
epoch 9, loss 0.2006, train acc 0.931, test acc 0.833, time 35.2 sec
epoch 10, loss 0.1713, train acc 0.941, test acc 0.841, time 35.2 sec
```

Summary

- Image augmentation generates random images based on existing training data to cope with overfitting.
- In order to obtain a definitive results during prediction, we usually only apply image augmentation to the training example, and do not use image augmentation with random operations during prediction.
- We can obtain classes related to image augmentation from Gluon's `transforms` module.

Exercises

- Train the model without using image augmentation: `train_with_data_aug(no_aug, no_aug)`. Compare training and testing accuracy when using and not using image augmentation. Can this comparative experiment support the argument that image augmentation can mitigate overfitting? Why?
- Add different image augmentation methods in model training based on the CIFAR-10 data set. Observe the implementation results.
- With reference to the MXNet documentation, what other image augmentation methods are provided in Gluon's `transforms` module?

Scan the QR Code to Discuss

10.2 Fine Tuning

In earlier chapters, we discussed how to train models on the Fashion-MNIST training data set, which only has 60,000 images. We also described ImageNet, the most widely used large-scale image data set in the academic world, with more than 10 million images and objects of over 1000 categories. However, the size of data sets that we often deal with is usually larger than the first, but smaller than the second.

Assume we want to identify different kinds of chairs in images and then push the purchase link to the user. One possible method is to first find a hundred common chairs, take one thousand different images with different angles for each chair, and then train a classification model on the collected image data set. Although this data set may be larger than Fashion-MNIST, the number of examples is still less than one tenth of ImageNet. This may result in the overfitting of the complicated model applicable to ImageNet on this data set. At the same time, because of the limited amount of data, the accuracy of the final trained model may not meet the practical requirements.

In order to deal with the above problems, an obvious solution is to collect more data. However, collecting and labeling data can consume a lot of time and money. For example, in order to collect the ImageNet data sets, researchers have spent millions of dollars of research funding. Although, recently, data collection costs have dropped significantly, the costs still cannot be ignored.

Another solution is to apply transfer learning to migrate the knowledge learned from the source data set to the target data set. For example, although the images in ImageNet are mostly unrelated to chairs, models trained on this data set can extract more general image features that can help identify edges, textures, shapes, and object composition. These similar features may be equally effective for recognizing a chair.

In this section, we introduce a common technique in transfer learning: fine tuning. As shown in Figure 9.1, fine tuning consists of the following four steps:

1. Pre-train a neural network model, i.e. the source model, on a source data set (e.g., the ImageNet data set).

2. Create a new neural network model, i.e. the target model. This replicates all model designs and their parameters on the source model, except the output layer. We assume that these model parameters contain the knowledge learned from the source data set and that this knowledge will be equally applicable to the target data set. We also assume that the output layer of the source model is closely related to the labels of the source data set and is therefore not used in the target model.

3. Add an output layer whose output size is the number of target data set categories to the target model, and randomly initialize the model parameters of this layer.

4. Train the target model on a target data set, such as a chair data set. We will train the output layer from scratch, while the parameters of all remaining layers are fine tuned based on the parameters of the source model.
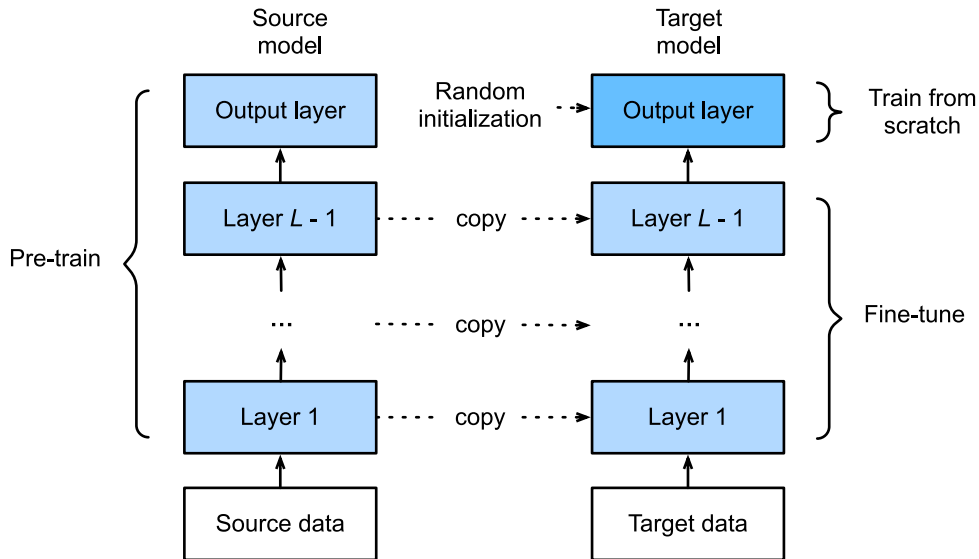
Fig. 10.1: Fine tuning.

10.2.1 Hot Dog Recognition

Next, we will use a specific example for practice: hot dog recognition. We will fine tune the ResNet model trained on the ImageNet data set based on a small data set. This small data set contains thousands of images, some of which contain hot dogs. We will use the model obtained by fine tuning to identify whether an image contains a hot dog.

First, import the packages and modules required for the experiment. Gluon's `model_zoo` package provides a common pre-trained model. If you want to get more pre-trained models for computer vision, you can use the GluonCV Toolkit[1].

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet import gluon, init, nd
        from mxnet.gluon import data as gdata, loss as gloss, model_zoo
        from mxnet.gluon import utils as gutils
        import os
        import zipfile
```

Get the Data Set

The hot dog data set we use was taken from online images and contains 1,400 positive images containing hot dogs and same number of negative images containing other foods. 1,000 images of various classes are used for training and the rest are used for testing.

We first download the compressed data set to the path `../data`. Then, we unzip the downloaded data set in this path and get two folders, `hotdog/train` and `hotdog/test`. Both folders have `hotdog` and `not-hotdog` category subfolders, each of which has corresponding image files.

```
In [2]: data_dir = '../data'
        base_url = 'https://apache-mxnet.s3-accelerate.amazonaws.com/'
        fname = gutils.download(
            base_url + 'gluon/dataset/hotdog.zip',
            path=data_dir, sha1_hash='fba480ffa8aa7e0febbb511d181409f899b9baa5')
        with zipfile.ZipFile(fname, 'r') as z:
            z.extractall(data_dir)
```

We create two `ImageFolderDataset` instances to read all the image files in the training data set and testing data set, respectively.

```
In [3]: train_imgs = gdata.vision.ImageFolderDataset(
            os.path.join(data_dir, 'hotdog/train'))
        test_imgs = gdata.vision.ImageFolderDataset(
            os.path.join(data_dir, 'hotdog/test'))
```

The first 8 positive examples and the last 8 negative images are shown below. As you can see, the images vary in size and aspect ratio.

```
In [4]: hotdogs = [train_imgs[i][0] for i in range(8)]
        not_hotdogs = [train_imgs[-i - 1][0] for i in range(8)]
        d2l.show_images(hotdogs + not_hotdogs, 2, 8, scale=1.4);
```
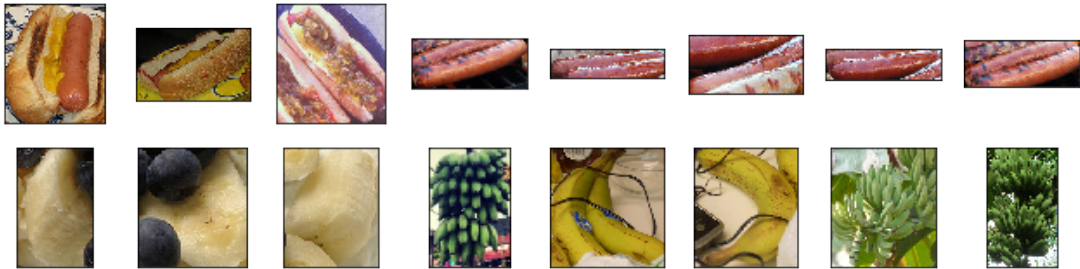

During training, we first crop a random area with random size and random aspect ratio from the image and then scale the area to an input with a height and width of 224 pixels. During testing, we scale the height and width of images to 256 pixels, and then crop the center area with height and width of 224 pixels to use as the input. In addition, we normalize the values of the three RGB (red, green, and blue) color channels. The average of all values of the channel is subtracted from each value and then the result is divided by the standard deviation of all values of the channel to produce the output.

```
In [5]: # We specify the mean and variance of the three RGB channels to normalize the
   ↪    image channel.
```

```
normalize = gdata.vision.transforms.Normalize(
    [0.485, 0.456, 0.406], [0.229, 0.224, 0.225])

train_augs = gdata.vision.transforms.Compose([
    gdata.vision.transforms.RandomResizedCrop(224),
    gdata.vision.transforms.RandomFlipLeftRight(),
    gdata.vision.transforms.ToTensor(),
    normalize])

test_augs = gdata.vision.transforms.Compose([
    gdata.vision.transforms.Resize(256),
    gdata.vision.transforms.CenterCrop(224),
    gdata.vision.transforms.ToTensor(),
    normalize])
```

Define and Initialize the Model

We use ResNet-18, which was pre-trained on the ImageNet data set, as the source model. Here, we specify `pretrained=True` to automatically download and load the pre-trained model parameters. The first time they are used, the model parameters need to be downloaded from the Internet.

```
In [6]: pretrained_net = model_zoo.vision.resnet18_v2(pretrained=True)
```

The pre-trained source model instance contains two member variables: `features` and `output`. The former contains all layers of the model, except the output layer, and the latter is the output layer of the model. The main purpose of this division is to facilitate the fine tuning of the model parameters of all layers except the output layer. The member variable `output` of source model is given below. As a fully connected layer, it transforms ResNet's final global average pooling layer output into 1000 class output on the ImageNet data set.

```
In [7]: pretrained_net.output
```

```
Out[7]: Dense(512 -> 1000, linear)
```

We then build a new neural network to use as the target model. It is defined in the same way as the pre-trained source model, but the final number of outputs is equal to the number of categories in the target data set. In the code below, the model parameters in the member variable `features` of the target model instance `finetune_net` are initialized to model parameters of the corresponding layer of the source model. Because the model parameters in `features` are obtained by pre-training on the ImageNet data set, it is good enough. Therefore, we generally only need to use small learning rates to fine tune these parameters. In contrast, model parameters in the member variable `output` are randomly initialized and generally require a larger learning rate to learn from scratch. Assume the learning rate in the `Trainer` instance is η and use a learning rate of 10η to update the model parameters in the member variable `output`.

```
In [8]: finetune_net = model_zoo.vision.resnet18_v2(classes=2)
        finetune_net.features = pretrained_net.features
        finetune_net.output.initialize(init.Xavier())
        # The model parameters in output will be updated using a learning rate ten
        # times greater
        finetune_net.output.collect_params().setattr('lr_mult', 10)
```

Fine Tune the Model

We first define a training function `train_fine_tuning` that uses fine tuning so it can be called multiple times.

```
In [9]: def train_fine_tuning(net, learning_rate, batch_size=128, num_epochs=5):
            train_iter = gdata.DataLoader(
                train_imgs.transform_first(train_augs), batch_size, shuffle=True)
            test_iter = gdata.DataLoader(
                test_imgs.transform_first(test_augs), batch_size)
            ctx = d2l.try_all_gpus()
            net.collect_params().reset_ctx(ctx)
            net.hybridize()
            loss = gloss.SoftmaxCrossEntropyLoss()
            trainer = gluon.Trainer(net.collect_params(), 'sgd', {
                'learning_rate': learning_rate, 'wd': 0.001})
            d2l.train(train_iter, test_iter, net, loss, trainer, ctx, num_epochs)
```

We set the learning rate in the `Trainer` instance to a smaller value, such as 0.01, in order to fine tune the model parameters obtained in pre-training. Based on the previous settings, we will train the output layer parameters of the target model from scratch using a learning rate ten times greater.

```
In [10]: train_fine_tuning(finetune_net, 0.01)

training on [gpu(0), gpu(1)]
epoch 1, loss 2.9287, train acc 0.709, test acc 0.909, time 17.0 sec
epoch 2, loss 0.3644, train acc 0.899, test acc 0.909, time 12.5 sec
epoch 3, loss 0.3548, train acc 0.898, test acc 0.885, time 12.4 sec
epoch 4, loss 0.4117, train acc 0.889, test acc 0.795, time 12.4 sec
epoch 5, loss 0.6554, train acc 0.878, test acc 0.939, time 12.4 sec
```

For comparison, we define an identical model, but initialize all of its model parameters to random values. Since the entire model needs to be trained from scratch, we can use a larger learning rate.

```
In [11]: scratch_net = model_zoo.vision.resnet18_v2(classes=2)
         scratch_net.initialize(init=init.Xavier())
         train_fine_tuning(scratch_net, 0.1)

training on [gpu(0), gpu(1)]
epoch 1, loss 0.7544, train acc 0.672, test acc 0.840, time 12.5 sec
epoch 2, loss 0.4159, train acc 0.812, test acc 0.776, time 12.4 sec
epoch 3, loss 0.4320, train acc 0.809, test acc 0.843, time 12.4 sec
epoch 4, loss 0.3723, train acc 0.847, test acc 0.868, time 12.3 sec
epoch 5, loss 0.3707, train acc 0.844, test acc 0.845, time 12.3 sec
```

As you can see, the fine-tuned model tends to achieve higher precision in the same epoch because the initial values of the parameters are better.

Summary

- Transfer learning migrates the knowledge learned from the source data set to the target data set. Fine tuning is a common technique for transfer learning.

- The target model replicates all model designs and their parameters on the source model, except the output layer, and fine tunes these parameters based on the target data set. In contrast, the output layer of the target model needs to be trained from scratch.

- Generally, fine tuning parameters use a smaller learning rate, while training the output layer from scratch can use a larger learning rate.

Exercises

- Keep increasing the learning rate of `finetune_net`. How does the precision of the model change?

- Further tune the hyper-parameters of `finetune_net` and `scratch_net` in the comparative experiment. Do they still have different precisions?

- Set the parameters in `finetune_net.features` to the parameters of the source model and do not update them during training. What will happen? You can use the following code.

```
In [12]: finetune_net.features.collect_params().setattr('grad_req', 'null')
```

- In fact, there is also a hotdog class in the `ImageNet` data set. Its corresponding weight parameter at the output layer can be obtained by using the following code. How can we use this parameter?

```
In [13]: weight = pretrained_net.output.weight
         hotdog_w = nd.split(weight.data(), 1000, axis=0)[713]
         hotdog_w.shape
Out[13]: (1, 512)
```

Reference

[1] GluonCV Toolkit. https://gluon-cv.mxnet.io/

Scan the QR Code to Discuss

10.3 Object Detection and Bounding Boxes

In the previous section, we introduced many models for image classification. In image classification tasks, we assume that there is only one main target in the image and we only focus on how to identify the

target category. However, in many situations, there are multiple targets in the image that we are interested in. We not only want to classify them, but also want to obtain their specific positions in the image. In computer vision, we refer to such tasks as object detection (or object detection).

Object detection is widely used in many fields. For example, in self-driving technology, we need to plan routes by identifying the locations of vehicles, pedestrians, roads, and obstacles in the captured video image. Robots often perform this type of task to detect targets of interest. Systems in the security field need to detect abnormal targets, such as intruders or bombs.

In the next few sections, we will introduce multiple deep learning models used for object detection. Before that, we should discuss the concept of target location. First, import the packages and modules required for the experiment.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet import image
```

Next, we will load the sample images that will be used in this section. We can see there is a dog on the left side of the image and a cat on the right. They are the two main targets in this image.

```
In [2]: d2l.set_figsize()
        img = image.imread('../img/catdog.jpg').asnumpy()
        d2l.plt.imshow(img);  # Add a semicolon to only display the image
```
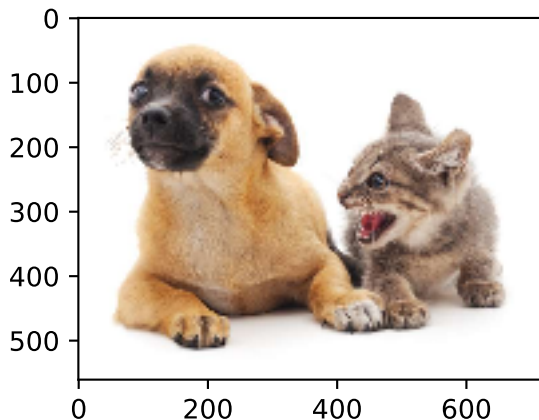

10.3.1 Bounding Box

In object detection, we usually use a bounding box to describe the target location. The bounding box is a rectangular box that can be determined by the x and y axis coordinates in the upper-left corner and the x and y axis coordinates in the lower-right corner of the rectangle. We will define the bounding boxes of the dog and the cat in the image based on the coordinate information in the above image. The origin of

the coordinates in the above image is the upper left corner of the image, and to the right and down are the positive directions of the x axis and the y axis, respectively.

```
In [3]: # bbox is the abbreviation for bounding box
        dog_bbox, cat_bbox = [60, 45, 378, 516], [400, 112, 655, 493]
```

We can draw the bounding box in the image to check if it is accurate. Before drawing the box, we will define a helper function `bbox_to_rect`. It represents the bounding box in the bounding box format of matplotlib.

```
In [4]: # This function has been saved in the d2l package for future use
        def bbox_to_rect(bbox, color):
            # Convert the bounding box (top-left x, top-left y, bottom-right x,
            # bottom-right y) format to matplotlib format: ((upper-left x,
            # upper-left y), width, height)
            return d2l.plt.Rectangle(
                xy=(bbox[0], bbox[1]), width=bbox[2]-bbox[0], height=bbox[3]-bbox[1],
                fill=False, edgecolor=color, linewidth=2)
```

After loading the bounding box on the image, we can see that the main outline of the target is basically inside the box.

```
In [5]: fig = d2l.plt.imshow(img)
        fig.axes.add_patch(bbox_to_rect(dog_bbox, 'blue'))
        fig.axes.add_patch(bbox_to_rect(cat_bbox, 'red'));
```
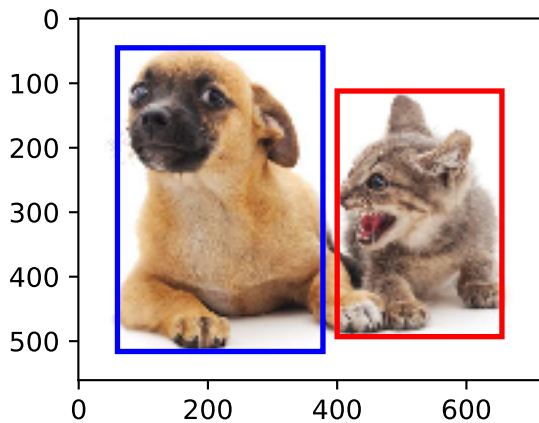

Summary

- In object detection, we not only need to identify all the objects of interest in the image, but also their positions. The positions are generally represented by a rectangular bounding box.

Exercises

- Find some images and try to label a bounding box that contains the target. Compare the difference between the time it takes to label the bounding box and label the category.

Scan the QR Code to Discuss

10.4 Anchor Boxes

Object detection algorithms usually sample a large number of regions in the input image, determine whether these regions contain objects of interest, and adjust the edges of the regions so as to predict the ground-truth bounding box of the target more accurately. Different models may use different region sampling methods. Here, we introduce one such method: it generates multiple bounding boxes with different sizes and aspect ratios while centering on each pixel. These bounding boxes are called anchor boxes. We will practice object detection based on anchor boxes in the following sections.

First, import the packages or modules required for this section. Here, we have introduced the `contrib` package, and modified the printing accuracy of NumPy. Because printing NDArray actually calls the print function of NumPy, the floating-point numbers in NDArray printed in this section are more concise.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet import contrib, gluon, image, nd
        import numpy as np
        np.set_printoptions(2)
```

10.4.1 Generate Multiple Anchor Boxes

Assume the input image has a height of h and width of w. We generate anchor boxes with different shapes centered on each pixel of the image. Assume the size is $s \in (0, 1]$, the aspect ratio is $r > 0$, and the width and height of the anchor box are $ws\sqrt{r}$ and $hs/\sqrt{r}$, respectively. When the center position is given, an anchor box with known width and height is determined.

Below we set a set of sizes $s_1, \ldots, s_n$ and a set of aspect ratios $r_1, \ldots, r_m$. If we use a combination of all sizes and aspect ratios with each pixel as the center, the input image will have a total of $whnm$ anchor

boxes. Although these anchor boxes may cover all ground-truth bounding boxes, the computational complexity is often excessive. Therefore, we are usually only interested in a combination containing s_1 or r_1 sizes and aspect ratios, that is:

$$(s_1, r_1), (s_1, r_2), \ldots, (s_1, r_m), (s_2, r_1), (s_3, r_1), \ldots, (s_n, r_1).$$

That is, the number of anchor boxes centered on the same pixel is $n + m - 1$. For the entire input image, we will generate a total of $wh(n + m - 1)$ anchor boxes.

The above method of generating anchor boxes has been implemented in the `MultiBoxPrior` function. We specify the input, a set of sizes, and a set of aspect ratios, and this function will return all the anchor boxes entered.

```
In [2]: img = image.imread('../img/catdog.jpg').asnumpy()
        h, w = img.shape[0:2]

        print(h, w)
        X = nd.random.uniform(shape=(1, 3, h, w))  # Construct input data
        Y = contrib.nd.MultiBoxPrior(X, sizes=[0.75, 0.5, 0.25], ratios=[1, 2, 0.5])
        Y.shape

561 728

Out[2]: (1, 2042040, 4)
```

We can see that the shape of the returned anchor box variable `y` is (batch size, number of anchor boxes, 4). After changing the shape of the anchor box variable `y` to (image height, image width, number of anchor boxes centered on the same pixel, 4), we can obtain all the anchor boxes centered on a specified pixel position. In the following example, we access the first anchor box centered on (250, 250). It has four elements: the x, y axis coordinates in the upper-left corner and the x, y axis coordinates in the lower-right corner of the anchor box. The coordinate values of the x and y axis are divided by the width and height of the image, respectively, so the value range is between 0 and 1.

```
In [3]: boxes = Y.reshape((h, w, 5, 4))
        boxes[250, 250, 0, :]

Out[3]:
        [0.06 0.07 0.63 0.82]
        <NDArray 4 @cpu(0)>
```

In order to describe all anchor boxes centered on one pixel in the image, we first define the `show_bboxes` function to draw multiple bounding boxes on the image.

```
In [4]: # This function is saved in the d2l package for future use
        def show_bboxes(axes, bboxes, labels=None, colors=None):
            def _make_list(obj, default_values=None):
                if obj is None:
                    obj = default_values
                elif not isinstance(obj, (list, tuple)):
                    obj = [obj]
                return obj

            labels = _make_list(labels)
            colors = _make_list(colors, ['b', 'g', 'r', 'm', 'c'])
            for i, bbox in enumerate(bboxes):
```

```
        color = colors[i % len(colors)]
        rect = d2l.bbox_to_rect(bbox.asnumpy(), color)
        axes.add_patch(rect)
        if labels and len(labels) > i:
            text_color = 'k' if color == 'w' else 'w'
            axes.text(rect.xy[0], rect.xy[1], labels[i],
                      va='center', ha='center', fontsize=9, color=text_color,
                      bbox=dict(facecolor=color, lw=0))
```

As we just saw, the coordinate values of the x and y axis in the variable `boxes` have been divided by the width and height of the image, respectively. When drawing images, we need to restore the original coordinate values of the anchor boxes and therefore define the variable `bbox_scale`. Now, we can draw all the anchor boxes centered on (250, 250) in the image. As you can see, the blue anchor box with a size of 0.75 and an aspect ratio of 1 covers the dog in the image well.

```
In [5]: d2l.set_figsize()
        bbox_scale = nd.array((w, h, w, h))
        fig = d2l.plt.imshow(img)
        show_bboxes(fig.axes, boxes[250, 250, :, :] * bbox_scale,
                    ['s=0.75, r=1', 's=0.5, r=1', 's=0.25, r=1', 's=0.75, r=2',
                     's=0.75, r=0.5'])
```
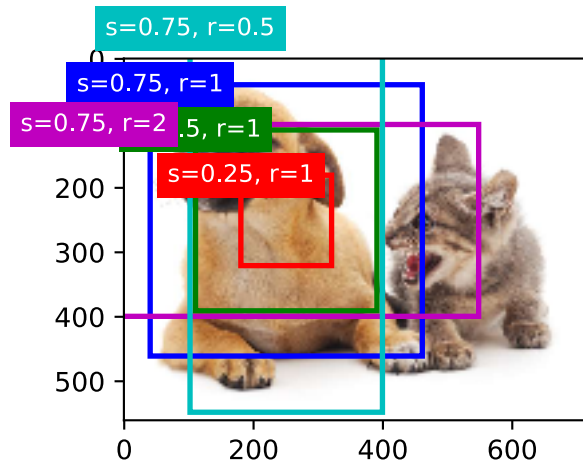

10.4.2 Intersection over Union

We just mentioned that the anchor box covers the dog in the image well. If the ground-truth bounding box of the target is known, how can well here be quantified? An intuitive method is to measure the similarity between anchor boxes and the ground-truth bounding box. We know that the Jaccard index can measure the similarity between two sets. Given sets $\mathcal{A}$ and $\mathcal{B}$, their Jaccard index is the size of their intersection divided by the size of their union:

$$J(\mathcal{A}, \mathcal{B}) = \frac{|\mathcal{A} \cap \mathcal{B}|}{|\mathcal{A} \cup \mathcal{B}|}.$$

In fact, we can consider the pixel area of a bounding box as a collection of pixels. In this way, we can measure the similarity of the two bounding boxes by the Jaccard index of their pixel sets. When we measure the similarity of two bounding boxes, we usually refer the Jaccard index as Intersection over Union (IoU), which is the ratio of the intersecting area to the union area of the two bounding boxes, as shown in Figure 9.2. The value range of IoU is between 0 and 1: 0 means that there are no overlapping pixels between the two bounding boxes, while 1 indicates that the two bounding boxes are equal.
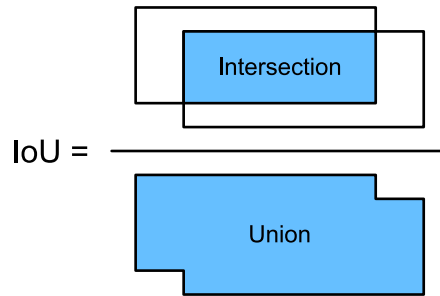

Fig. 10.2: IoU is the ratio of the intersecting area to the union area of two bounding boxes.

For the remainder of this section, we will use IoU to measure the similarity between anchor boxes and ground-truth bounding boxes, and between different anchor boxes.

10.4.3 Labeling Training Set Anchor Boxes

In the training set, we consider each anchor box as a training example. In order to train the object detection model, we need to mark two types of labels for each anchor box: first, the category of the target contained in the anchor box (category) and, second, the offset of the ground-truth bounding box relative to the anchor box (offset). In object detection, we first generate multiple anchor boxes, predict the categories and offsets for each anchor box, adjust the anchor box position according to the predicted offset to obtain the bounding boxes to be used for prediction, and finally filter out the prediction bounding boxes that need to be output.

We know that, in the object detection training set, each image is labelled with the location of the ground-truth bounding box and the category of the target contained. After the anchor boxes are generated, we primarily label anchor boxes based on the location and category information of the ground-truth bounding boxes similar to the anchor boxes. So how do we assign ground-truth bounding boxes to anchor boxes similar to them?

Assume the anchor boxes in the image are $A_1, A_2, \ldots, A_{n_a}$ and the ground-truth bounding boxes are $B_1, B_2, \ldots, B_{n_b}$ and $n_a \geq n_b$. Define matrix $\boldsymbol{X} \in \mathbb{R}^{n_a \times n_b}$, where element x_{ij} in the ith row and jth column is the IoU of the anchor box A_i to the ground-truth bounding box B_j. First, we find the largest element in the matrix $\boldsymbol{X}$ and record the row index and column index of the element as i_1, j_1. We assign the ground-truth bounding box B_{j_1} to the anchor box A_{i_1}. Obviously, anchor box A_{i_1} and ground-truth bounding box B_{j_1} have the highest similarity among all the anchor box - ground-truth bounding box

pairings. Next, discard all elements in the i_1th row and the j_1th column in the matrix $\boldsymbol{X}$. Find the largest remaining element in the matrix $\boldsymbol{X}$ and record the row index and column index of the element as i_2, j_2. We assign ground-truth bounding box B_{j_2} to anchor box A_{i_2} and then discard all elements in the i_2th row and the j_2th column in the matrix $\boldsymbol{X}$. At this point, elements in two rows and two columns in the matrix $\boldsymbol{X}$ have been discarded. We proceed until all elements in the n_b column in the matrix $\boldsymbol{X}$ are discarded. At this time, we have assigned a ground-truth bounding box to each of the n_b anchor boxes. Next, we only traverse the remaining $n_a - n_b$ anchor boxes. Given anchor box A_i, find the bounding box B_j with the largest IoU with A_i according to the ith row of the matrix $\boldsymbol{X}$, and only assign ground-truth bounding box B_j to anchor box A_i when the IoU is greater than the predetermined threshold.

As shown in Figure 9.3 (left), assuming that the maximum value in the matrix $\boldsymbol{X}$ is x_{23}, we will assign ground-truth bounding box B_3 to anchor box A_2. Then, we discard all the elements in row 2 and column 3 of the matrix, find the largest element x_{71} of the remaining shaded area, and assign ground-truth bounding box B_1 to anchor box A_7. Then, as shown in Figure 9.3 (middle), discard all the elements in row 7 and column 1 of the matrix, find the largest element x_{54} of the remaining shaded area, and assign ground-truth bounding box B_4 to anchor box A_5. Finally, as shown in Figure 9.3 (right), discard all the elements in row 5 and column 4 of the matrix, find the largest element x_{92} of the remaining shaded area, and assign ground-truth bounding box B_2 to anchor box A_9. After that, we only need to traverse the remaining anchor boxes of A_2, A_5, A_7, A_9 and determine whether to assign ground-truth bounding boxes to the remaining anchor boxes according to the threshold.
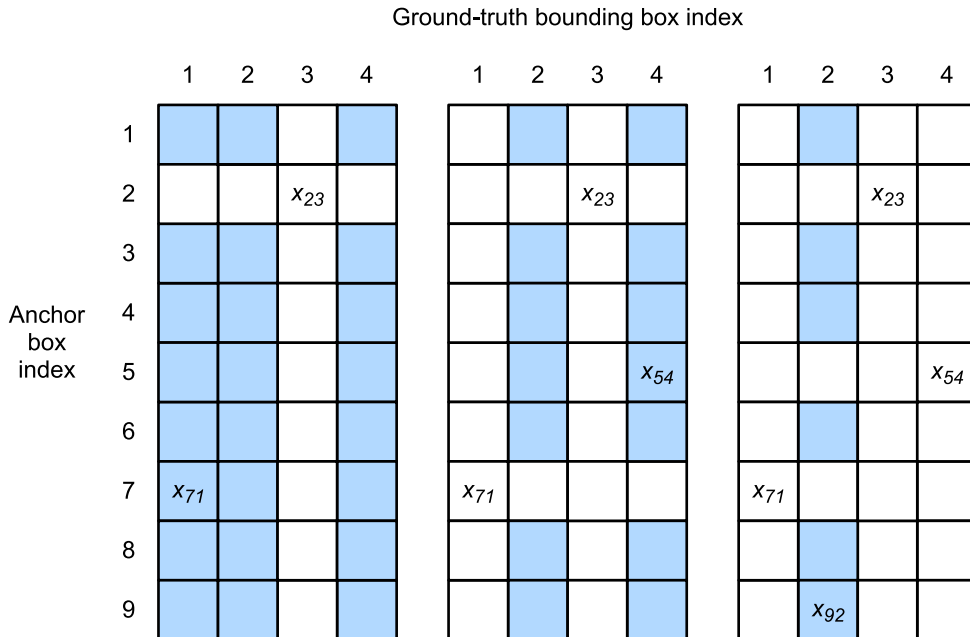

Fig. 10.3: Assign ground-truth bounding boxes to anchor boxes.

Now we can label the categories and offsets of the anchor boxes. If an anchor box A is assigned ground-truth bounding box B, the category of the anchor box A is set to the category of B and the offset of the anchor box A is set according to the relative position of the central coordinates of B and A and the relative sizes of the two boxes. Because the positions and sizes of various boxes in the data set may vary, these relative positions and relative sizes usually require some special transformations to make the offset distribution more uniform and easier to fit. Assume the center coordinates of anchor box A and its assigned ground-truth bounding box B are $(x_a, y_a), (x_b, y_b)$, the widths of A and B are w_a, W_b, and their heights are h_a, h_b, respectively. In this case, a common technique is to label the offset of A as

$$\left(\frac{\frac{x_b - x_a}{w_a} - \mu_x}{\sigma_x}, \frac{\frac{y_b - y_a}{h_a} - \mu_y}{\sigma_y}, \frac{\log \frac{w_b}{w_a} - \mu_w}{\sigma_w}, \frac{\log \frac{h_b}{h_a} - \mu_h}{\sigma_h} \right),$$

The default values of the constant are $\mu_x = \mu_y = \mu_w = \mu_h = 0, \sigma_x = \sigma_y = 0.1, and \sigma_w = \sigma_h = 0.2$. If an anchor box is not assigned a ground-truth bounding box, we only need to set the category of the anchor box to background. Anchor boxes whose category is background are often referred to as negative anchor boxes, and the rest are referred to as positive anchor boxes.

Below we demonstrate a detailed example. We define ground-truth bounding boxes for the cat and dog in the read image, where the first element is category (0 for dog, 1 for cat) and the remaining four elements are the x, y axis coordinates at top-left corner and x, y axis coordinates at lower-right corner (the value range is between 0 and 1). Here, we construct five anchor boxes to be labeled by the coordinates of the upper-left corner and the lower-right corner, which are recorded as $A_0, \ldots, A_4$, respectively (the index in the program starts from 0). First, draw the positions of these anchor boxes and the ground-truth bounding boxes in the image.

```
In [6]: ground_truth = nd.array([[0, 0.1, 0.08, 0.52, 0.92],
                                  [1, 0.55, 0.2, 0.9, 0.88]])
        anchors = nd.array([[0, 0.1, 0.2, 0.3], [0.15, 0.2, 0.4, 0.4],
                            [0.63, 0.05, 0.88, 0.98], [0.66, 0.45, 0.8, 0.8],
                            [0.57, 0.3, 0.92, 0.9]])

        fig = d2l.plt.imshow(img)
        show_bboxes(fig.axes, ground_truth[:, 1:] * bbox_scale, ['dog', 'cat'], 'k')
        show_bboxes(fig.axes, anchors * bbox_scale, ['0', '1', '2', '3', '4']);
```
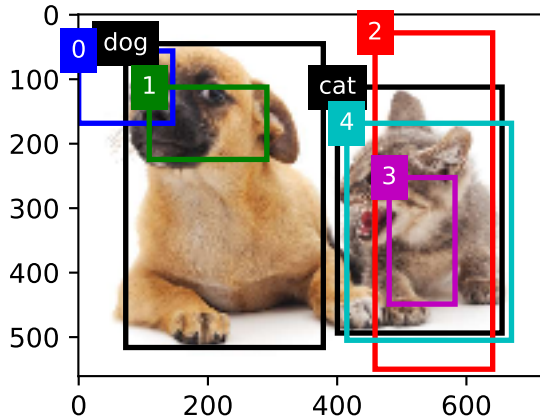
We can label categories and offsets for anchor boxes by using the `MultiBoxTarget` function in the `contrib.nd` module. This function sets the background category to 0 and increments the integer index of the target category from zero by 1 (1 for dog and 2 for cat). We add example dimensions to the anchor boxes and ground-truth bounding boxes and construct random predicted results with a shape of (batch size, number of categories including background, number of anchor boxes) by using the `expand_dims` function.

```
In [7]: labels = contrib.nd.MultiBoxTarget(anchors.expand_dims(axis=0),
                                           ground_truth.expand_dims(axis=0),
                                           nd.zeros((1, 3, 5)))
```

There are three items in the returned result, all of which are in NDArray format. The third item is represented by the category labelled for the anchor box.

```
In [8]: labels[2]
Out[8]:
        [[0. 1. 2. 0. 2.]]
        <NDArray 1x5 @cpu(0)>
```

We analyze these labelled categories based on positions of anchor boxes and ground-truth bounding boxes in the image. First, in all anchor box - ground-truth bounding box pairs, the IoU of anchor box A_4 to the ground-truth bounding box of the cat is the largest, so the category of anchor box A_4 is labeled as cat. Without considering anchor box A_4 or the ground-truth bounding box of the cat, in the remaining anchor box - ground-truth bounding box pairs, the pair with the largest IoU is anchor box A_1 and the ground-truth bounding box of the dog, so the category of anchor box A_1 is labeled as dog. Next, traverse the remaining three unlabeled anchor boxes. The category of the ground-truth bounding box with the largest IoU with anchor box A_0 is dog, but the IoU is smaller than the threshold (the default is 0.5), so the category is labeled as background; the category of the ground-truth bounding box with the largest IoU with anchor box A_2 is cat and the IoU is greater than the threshold, so the category is labeled as cat; the category of the ground-truth bounding box with the largest IoU with anchor box A_3 is cat, but the IoU is smaller than the threshold, so the category is labeled as background.

The second item of the return value is a mask variable, with the shape of (batch size, four times the

number of anchor boxes). The elements in the mask variable correspond one-to-one with the four offset values of each anchor box. Because we don't care about background detection, offsets of the negative class should not affect the target function. By multiplying by element, the 0 in the mask variable can filter out negative class offsets before calculating target function.

```
In [9]: labels[1]

Out[9]:
      [[0. 0. 0. 0. 1. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 1. 1. 1. 1.]]
      <NDArray 1x20 @cpu(0)>
```

The first item returned is the four offset values labeled for each anchor box, with the offsets of negative class anchor boxes labeled as 0.

```
In [10]: labels[0]

Out[10]:
      [[ 0.00e+00  0.00e+00  0.00e+00  0.00e+00  1.40e+00  1.00e+01  2.59e+00
         7.18e+00 -1.20e+00  2.69e-01  1.68e+00 -1.57e+00  0.00e+00  0.00e+00
         0.00e+00  0.00e+00 -5.71e-01 -1.00e+00 -8.94e-07  6.26e-01]]
      <NDArray 1x20 @cpu(0)>
```

10.4.4 Output Bounding Boxes for Prediction

During model prediction phase, we first generate multiple anchor boxes for the image and then predict categories and offsets for these anchor boxes one by one. Then, we obtain prediction bounding boxes based on anchor boxes and their predicted offsets. When there are many anchor boxes, many similar prediction bounding boxes may be output for the same target. To simplify the results, we can remove similar prediction bounding boxes. A commonly used method is called non-maximum suppression (NMS).

Let us take a look at how NMS works. For a prediction bounding box B, the model calculates the predicted probability for each category. Assume the largest predicted probability is p, the category corresponding to this probability is the predicted category of B. We also refer to p as the confidence level of prediction bounding box B. On the same image, we sort the prediction bounding boxes with predicted categories other than background by confidence level from high to low, and obtain the list L. Select the prediction bounding box B_1 with highest confidence level from L as a baseline and remove all non-benchmark prediction bounding boxes with an IoU with B_1 greater than a certain threshold from L. The threshold here is a preset hyper-parameter. At this point, L retains the prediction bounding box with the highest confidence level and removes other prediction bounding boxes similar to it. Next, select the prediction bounding box B_2 with the second highest confidence level from L as a baseline, and remove all non-benchmark prediction bounding boxes with an IoU with B_2 greater than a certain threshold from L. Repeat this process until all prediction bounding boxes in L have been used as a baseline. At this time, the IoU of any pair of prediction bounding boxes in L is less than the threshold. Finally, output all prediction bounding boxes in the list L.

Next, we will look at a detailed example. First, construct four anchor boxes. For the sake of simplicity, we assume that predicted offsets are all 0. This means that the prediction bounding boxes are anchor boxes. Finally, we construct a predicted probability for each category.

```
In [11]: anchors = nd.array([[0.1, 0.08, 0.52, 0.92], [0.08, 0.2, 0.56, 0.95],
                             [0.15, 0.3, 0.62, 0.91], [0.55, 0.2, 0.9, 0.88]])
         offset_preds = nd.array([0] * anchors.size)
         cls_probs = nd.array([[0] * 4,  # Predicted probability for background
                               [0.9, 0.8, 0.7, 0.1],  # Predicted probability for dog
                               [0.1, 0.2, 0.3, 0.9]])  # Predicted probability for cat
```

Print prediction bounding boxes and their confidence levels on the image.

```
In [12]: fig = d2l.plt.imshow(img)
         show_bboxes(fig.axes, anchors * bbox_scale,
                     ['dog=0.9', 'dog=0.8', 'dog=0.7', 'cat=0.9'])
```
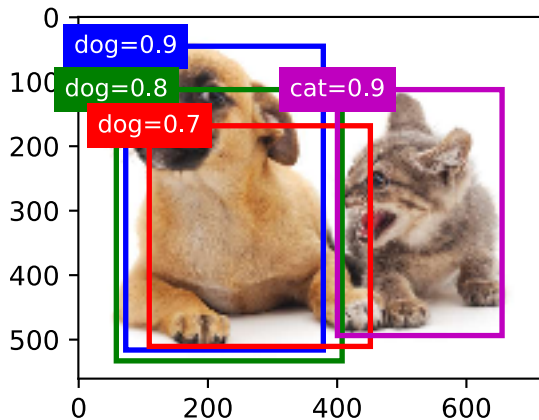

We use the `MultiBoxDetection` function of the `contrib.nd` module to perform NMS and set the threshold to 0.5. This adds an example dimension to the NDArray input. We can see that the shape of the returned result is (batch size, number of anchor boxes, 6). The 6 elements of each row represent the output information for the same prediction bounding box. The first element is the predicted category index, which starts from 0 (0 is dog, 1 is cat). The value -1 indicates background or removal in NMS. The second element is the confidence level of prediction bounding box. The remaining four elements are the x, y axis coordinates of the upper-left corner and the x, y axis coordinates of the lower-right corner of the prediction bounding box (the value range is between 0 and 1).

```
In [13]: output = contrib.ndarray.MultiBoxDetection(
             cls_probs.expand_dims(axis=0), offset_preds.expand_dims(axis=0),
             anchors.expand_dims(axis=0), nms_threshold=0.5)
         output

Out[13]:
         [[[ 0.    0.9    0.1   0.08  0.52  0.92]
           [ 1.    0.9    0.55  0.2   0.9   0.88]
           [-1.    0.8    0.08  0.2   0.56  0.95]
           [-1.    0.7    0.15  0.3   0.62  0.91]]]
         <NDArray 1x4x6 @cpu(0)>
```

We remove the prediction bounding boxes of category -1 and visualize the results retained by NMS.

```
In [14]: fig = d2l.plt.imshow(img)
```

```
for i in output[0].asnumpy():
    if i[0] == -1:
        continue
    label = ('dog=', 'cat=')[int(i[0])] + str(i[1])
    show_bboxes(fig.axes, [nd.array(i[2:]) * bbox_scale], label)
```
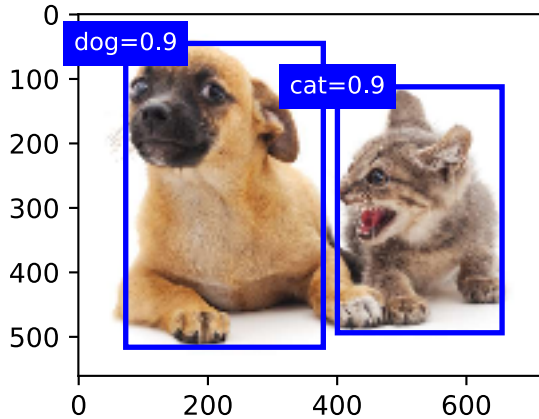

In practice, we can remove prediction bounding boxes with lower confidence levels before performing NMS, thereby reducing the amount of computation for NMS. We can also filter the output of NMS, for example, by only retaining results with higher confidence levels as the final output.

Summary

- We generate multiple anchor boxes with different sizes and aspect ratios, centered on each pixel.
- IoU is the ratio of the intersecting area to the union area of two bounding boxes.
- In the training set, we mark two types of labels for each anchor box: one is the category of the target contained in the anchor box and the other is the offset of the ground-truth bounding box relative to the anchor box.
- When predicting, we can use non-maximum suppression (NMS) to remove similar prediction bounding boxes, thereby simplifying the results.

Exercises

- Change the `sizes` and `ratios` values in `contrib.nd.MultiBoxPrior` and observe the changes to the generated anchor boxes.
- Construct two bounding boxes with and IoU of 0.5, and observe their coincidence.
- Verify the output of offset `labels[0]` by marking the anchor box offsets as defined in this section (the constant is the default value).

- Modify the variable `anchors` in the Labeling Training Set Anchor Boxes and Output Bounding Boxes for Prediction sections. How do the results change?

Scan the QR Code to Discuss

10.5 Multiscale Object Detection

In the *Anchor Box* section, we generated multiple anchor boxes centered on each pixel of the input image. These anchor boxes are used to sample different regions of the input image. However, if anchor boxes are generated centered on each pixel of the image, soon there will be too many anchor boxes for us to compute. For example, we assume that the input image has a height and a width of 561 and 728 pixels respectively. If five different shapes of anchor boxes are generated centered on each pixel, over two million anchor boxes ($561 \times 728 \times 5$) need to be predicted and labeled on the image.

It is not difficult to reduce the number of anchor boxes. An easy way is to apply uniform sampling on a small portion of pixels from the input image and generate anchor boxes centered on the sampled pixels. In addition, we can generate anchor boxes of varied numbers and sizes on multiple scales. Notice that smaller objects are more likely to be positioned on the image than larger ones. Here, we will use a simple example: Objects with shapes of 1×1, 1×2, and 2×2 may have 4, 2, and 1 possible position(s) on an image with the shape 2×2. Therefore, when using smaller anchor boxes to detect smaller objects, we can sample more regions; when using larger anchor boxes to detect larger objects, we can sample fewer regions.

To demonstrate how to generate anchor boxes on multiple scales, let us read an image first. It has a height and width of 561 * 728 pixels.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet import contrib, image, nd

        img = image.imread('../img/catdog.jpg')
        h, w = img.shape[0:2]
        h, w
Out[1]: (561, 728)
```

In the *Two-Dimensional Convolutional Layer* section, the 2D array output of the convolutional neural network (CNN) is called a feature map. We can determine the midpoints of anchor boxes uniformly sampled on any image by defining the shape of the feature map.

The function `display_anchors` is defined below. We are going to generate anchor boxes `anchors` centered on each unit (pixel) on the feature map `fmap`. Since the coordinates of axes x and y in anchor boxes `anchors` have been divided by the width and height of the feature map `fmap`, values between 0 and 1 can be used to represent relative positions of anchor boxes in the feature map. Since the midpoints of anchor boxes `anchors` overlap with all the units on feature map `fmap`, the relative spatial positions of the midpoints of the `anchors` on any image must have a uniform distribution. Specifically, when the width and height of the feature map are set to `fmap_w` and `fmap_h` respectively, the function will conduct uniform sampling for `fmap_h` rows and `fmap_w` columns of pixels and use them as midpoints to generate anchor boxes with size s (we assume that the length of list s is 1) and different aspect ratios (`ratios`).

```
In [2]: d2l.set_figsize()

        def display_anchors(fmap_w, fmap_h, s):
            # The values from the first two dimensions will not affect the output
            fmap = nd.zeros((1, 10, fmap_w, fmap_h))
            anchors = contrib.nd.MultiBoxPrior(fmap, sizes=s, ratios=[1, 2, 0.5])
            bbox_scale = nd.array((w, h, w, h))
            d2l.show_bboxes(d2l.plt.imshow(img.asnumpy()).axes,
                            anchors[0] * bbox_scale)
```

We will first focus on the detection of small objects. In order to make it easier to distinguish upon display, the anchor boxes with different midpoints here do not overlap. We assume that the size of the anchor boxes is 0.15 and the height and width of the feature map are 4. We can see that the midpoints of anchor boxes from the 4 rows and 4 columns on the image are uniformly distributed.

```
In [3]: display_anchors(fmap_w=4, fmap_h=4, s=[0.15])
```
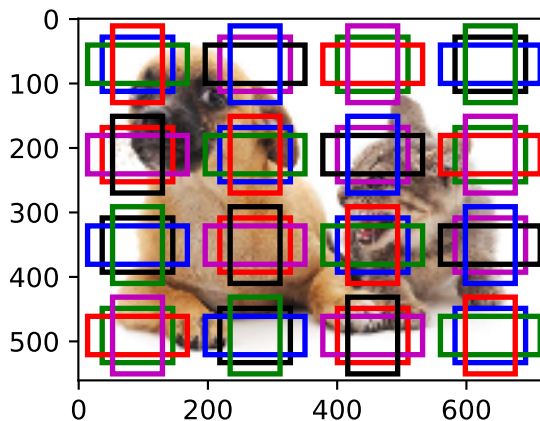

We are going to reduce the height and width of the feature map by half and use a larger anchor box to detect larger objects. When the size is set to 0.4, overlaps will occur between regions of some anchor

boxes.

```
In [4]: display_anchors(fmap_w=2, fmap_h=2, s=[0.4])
```
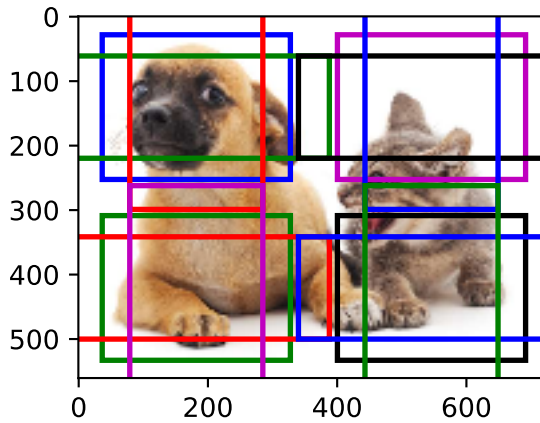

Finally, we are going to reduce the height and width of the feature map by half and increase the anchor box size to 0.8. Now the midpoint of the anchor box is the center of the image.

```
In [5]: display_anchors(fmap_w=1, fmap_h=1, s=[0.8])
```
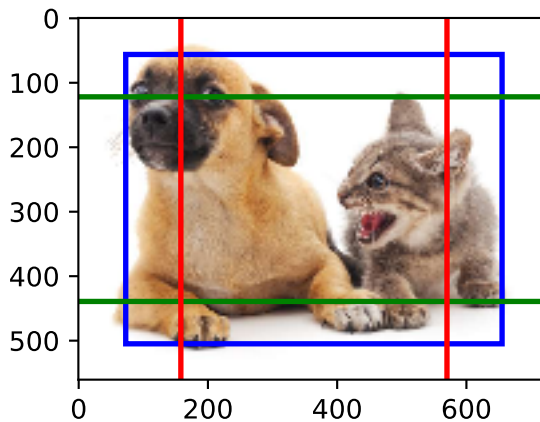

Since we have generated anchor boxes of different sizes on multiple scales, we will use them to detect objects of various sizes at different scales. Now we are going to introduce a method based on convolutional neural networks (CNNs).

At a certain scale, suppose we generate $h \times w$ sets of anchor boxes with different midpoints based on c_i feature maps with the shape $h \times w$ and the number of anchor boxes in each set is a. For example, for the first scale of the experiment, we generate 16 sets of anchor boxes with different midpoints based on 10 (number of channels) feature maps with a shape of 4×4, and each set contains 3 anchor boxes.

Next, each anchor box is labeled with a category and offset based on the classification and position of the ground-truth bounding box. At the current scale, the object detection model needs to predict the category and offset of $h \times w$ sets of anchor boxes with different midpoints based on the input image.

We assume that the c_i feature maps are the intermediate output of the CNN based on the input image. Since each feature map has $h \times w$ different spatial positions, the same position will have c_i units. According to the definition of receptive field in the *Two-Dimensional Convolutional Layer* section, the c_i units of the feature map at the same spatial position have the same receptive field on the input image. Thus, they represent the information of the input image in this same receptive field. Therefore, we can transform the c_i units of the feature map at the same spatial position into the categories and offsets of the a anchor boxes generated using that position as a midpoint. It is not hard to see that, in essence, we use the information of the input image in a certain receptive field to predict the category and offset of the anchor boxes close to the field on the input image.

When the feature maps of different layers have receptive fields of different sizes on the input image, they are used to detect objects of different sizes. For example, we can design a network to have a wider receptive field for each unit in the feature map that is closer to the output layer, to detect objects with larger sizes in the input image.

We will implement a multiscale object detection model in the following section.

Summary

- We can generate anchor boxes with different numbers and sizes on multiple scales to detect objects of different sizes on multiple scales.

- The shape of the feature map can be used to determine the midpoint of the anchor boxes that uniformly sample any image.

- We use the information for the input image from a certain receptive field to predict the category and offset of the anchor boxes close to that field on the image.

Exercises

- Given an input image, assume $1 \times c_i \times h \times w$ to be the shape of the feature map while c_i, h, w are the number, height, and width of the feature map. What methods can you think of to convert this variable into the anchor box's category and offset? What is the shape of the output?

10.6 Object Detection Data Set (Pikachu)

There are no small data sets, like MNIST or Fashion-MNIST, in the object detection field. In order to quickly test models, we are going to assemble a small data set. First, we generate 1000 Pikachu images of different angles and sizes using an open source 3D Pikachu model. Then, we collect a series of background images and place a Pikachu image at a random position on each image. We use the im2rec tool provided by MXNet to convert the images to binary RecordIO format[1]. This format can reduce the storage overhead of the data set on the disk and improve the reading efficiency. If you want to learn more about how to read images, refer to the documentation for the GluonCV Toolkit[2].

10.6.1 Download the Data Set

The Pikachu data set in RecordIO format can be downloaded directly from the Internet. The operation for downloading the data set is defined in the function `_download_pikachu`.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet import gluon, image
        from mxnet.gluon import utils as gutils
        import os

        def _download_pikachu(data_dir):
            root_url = ('https://apache-mxnet.s3-accelerate.amazonaws.com/'
                        'gluon/dataset/pikachu/')
            dataset = {'train.rec': 'e6bcb6ffba1ac04ff8a9b1115e650af56ee969c8',
                       'train.idx': 'dcf7318b2602c06428b9988470c731621716c393',
                       'val.rec': 'd6c33f799b4d058e82f2cb5bd9a976f69d72d520'}
            for k, v in dataset.items():
                gutils.download(root_url + k, os.path.join(data_dir, k), sha1_hash=v)
```

10.6.2 Read the Data Set

We are going to read the object detection data set by creating the instance `ImageDetIter`. The Det in the name refers to Detection. We will read the training data set in random order. Since the format

---

of the data set is RecordIO, we need the image index file `'train.idx'` to read random mini-batches. In addition, for each image of the training set, we will use random cropping and require the cropped image to cover at least 95% of each object. Since the cropping is random, this requirement is not always satisfied. We preset the maximum number of random cropping attempts to 200. If none of them meets the requirement, the image will not be cropped. To ensure the certainty of the output, we will not randomly crop the images in the test data set. We also do not need to read the test data set in random order.

```
In [2]:  # This function has been saved in the d2l package for future use
         # Edge_size: the width and height of the output image
         def load_data_pikachu(batch_size, edge_size=256):
             data_dir = '../data/pikachu'
             _download_pikachu(data_dir)
             train_iter = image.ImageDetIter(
                 path_imgrec=os.path.join(data_dir, 'train.rec'),
                 path_imgidx=os.path.join(data_dir, 'train.idx'),
                 batch_size=batch_size,
                 data_shape=(3, edge_size, edge_size),  # The shape of the output image
                 shuffle=True,  # Read the data set in random order
                 rand_crop=1,  # The probability of random cropping is 1
                 min_object_covered=0.95, max_attempts=200)
             val_iter = image.ImageDetIter(
                 path_imgrec=os.path.join(data_dir, 'val.rec'), batch_size=batch_size,
                 data_shape=(3, edge_size, edge_size), shuffle=False)
             return train_iter, val_iter
```

Below, we read a mini-batch and print the shape of the image and label. The shape of the image is the same as in the previous experiment (batch size, number of channels, height, width). The shape of the label is (batch size, m, 5), where m is equal to the maximum number of bounding boxes contained in a single image in the data set. Although computation for the mini-batch is very efficient, it requires each image to contain the same number of bounding boxes so that they can be placed in the same batch. Since each image may have a different number of bounding boxes, we can add illegal bounding boxes to images that have less than m bounding boxes until each image contains m bounding boxes. Thus, we can read a mini-batch of images each time. The label of each bounding box in the image is represented by an array of length 5 The first element in the array is the category of the object contained in the bounding box. When the value is -1, the bounding box is an illegal bounding box for filling purpose. The remaining four elements of the array represent the x, y axis coordinates of the upper-left corner of the bounding box and the x, y axis coordinates of the lower-right corner of the bounding box (the value range is between 0 and 1). The Pikachu data set here has only one bounding box per image, so $m = 1$.

```
In [3]:  batch_size, edge_size = 32, 256
         train_iter, _ = load_data_pikachu(batch_size, edge_size)
         batch = train_iter.next()
         batch.data[0].shape, batch.label[0].shape
Out[3]:  ((32, 3, 256, 256), (32, 1, 5))
```

10.6.3 Graphic Data

We have ten images with bounding boxes on them. We can see that the angle, size, and position of Pikachu are different in each image. Of course, this is a simple man-made data set. In actual practice, the

data is usually much more complicated.

```
In [4]: imgs = (batch.data[0][0:10].transpose((0, 2, 3, 1))) / 255
        axes = d2l.show_images(imgs, 2, 5).flatten()
        for ax, label in zip(axes, batch.label[0][0:10]):
            d2l.show_bboxes(ax, [label[0][1:5] * edge_size], colors=['w'])
```
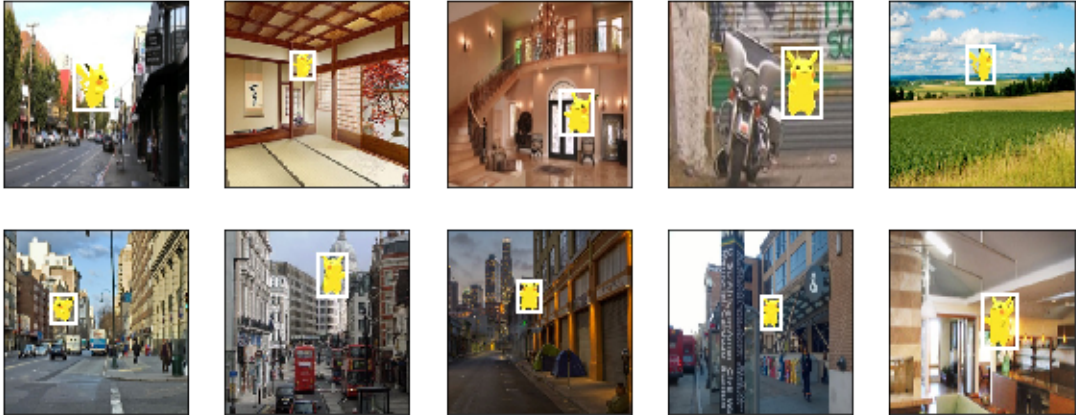

Summary

- The Pikachu data set we synthesized can be used to test object detection models.

- The data reading for object detection is similar to that for image classification. However, after we introduce bounding boxes, the label shape and image augmentation (e.g., random cropping) are changed.

Exercises

- Referring to the MXNet documentation, what are the parameters for the constructors of the `image.ImageDetIter` and `image.CreateDetAugmenter` classes? What is their significance?

References

[1] im2rec Tool. https://github.com/apache/incubator-mxnet/blob/master/tools/im2rec.py

[2] GluonCV Toolkit. https://gluon-cv.mxnet.io/

10.7 Single Shot Multibox Detection (SSD)

In the previous few sections, we have introduced bounding boxes, anchor boxes, multiscale object detection, and data sets. Now, we will use this background knowledge to construct an object detection model: single shot multibox detection (SSD)[1]. This quick and easy model is already widely used. Some of the design concepts and implementation details of this model are also applicable to other object detection models.

10.7.1 Model

Figure 9.4 shows the design of an SSD model. The model's main components are a base network block and several multiscale feature blocks connected in a series. Here, the base network block is used to extract features of original images, and it generally takes the form of a deep convolutional neural network. The paper on SSDs chooses to place a truncated VGG before the classification layer[1], but this is now commonly replaced by ResNet. We can design the base network so that it outputs larger heights and widths. In this way, more anchor boxes are generated based on this feature map, allowing us to detect smaller objects. Next, each multiscale feature block reduces the height and width of the feature map provided by the previous layer (for example, it may reduce the sizes by half). The blocks then use each element in the feature map to expand the receptive field on the input image. In this way, the closer a multiscale feature block is to the top of Figure 9.4 the smaller its output feature map, and the fewer the anchor boxes that are generated based on the feature map. In addition, the closer a feature block is to the top, the larger the receptive field of each element in the feature map and the better suited it is to detect larger objects. As the SSD generates different numbers of anchor boxes of different sizes based on the base network block and each multiscale feature block and then predicts the categories and offsets (i.e., predicted bounding boxes) of the anchor boxes in order to detect objects of different sizes, SSD is a multiscale object detection model.
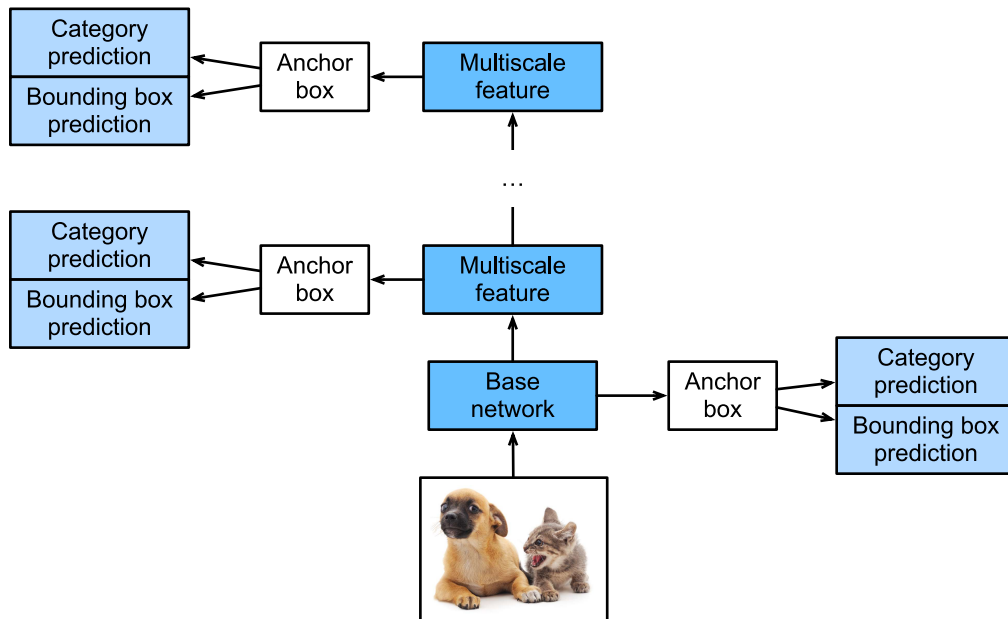
Fig. 10.4: The SSD is composed of a base network block and several multiscale feature blocks connected in a series.

Next, we will describe the implementation of the modules in the figure. First, we need to discuss the implementation of category prediction and bounding box prediction.

Category Prediction Layer

Set the number of object categories to q. In this case, the number of anchor box categories is $q + 1$, with 0 indicating an anchor box that only contains background. For a certain scale, set the height and width of the feature map to h and w, respectively. If we use each element as the center to generate a anchor boxes, we need to classify a total of hwa anchor boxes. If we use a fully connected layer (FCN) for the output, this will likely result in an excessive number of model parameters. Recall how we used convolutional layer channels to output category predictions in the *Network in Network (NiN)* section. SSD uses the same method to reduce the model complexity.

Specifically, the category prediction layer uses a convolutional layer that maintains the input height and width. Thus, the output and input have a one-to-one correspondence to the spatial coordinates along the width and height of the feature map. Assuming that the output and input have the same spatial coordinates (x, y), the channel for the coordinates (x, y) on the output feature map contains the category predictions for all anchor boxes generated using the input feature map coordinates (x, y) as the center. Therefore, there are $a(q + 1)$ output channels, with the output channels indexed as $i(q + 1) + j$ $(0 \leq j \leq q)$ representing the predictions of the category index j for the anchor box index i.

Now, we will define a category prediction layer of this type. After we specify the parameters a and q, it uses a 3×3 convolutional layer with a padding of 1. The heights and widths of the input and output of this convolutional layer remain unchanged.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet import autograd, contrib, gluon, image, init, nd
        from mxnet.gluon import loss as gloss, nn
        import time

        def cls_predictor(num_anchors, num_classes):
            return nn.Conv2D(num_anchors * (num_classes + 1), kernel_size=3,
                             padding=1)
```

Bounding Box Prediction Layer

The design of the bounding box prediction layer is similar to that of the category prediction layer. The only difference is that, here, we need to predict 4 offsets for each anchor box, rather than $q + 1$ categories.

```
In [2]: def bbox_predictor(num_anchors):
            return nn.Conv2D(num_anchors * 4, kernel_size=3, padding=1)
```

Concatenating Predictions for Multiple Scales

As we mentioned, SSD uses feature maps based on multiple scales to generate anchor boxes and predict their categories and offsets. Because the shapes and number of anchor boxes centered on the same element differ for the feature maps of different scales, the prediction outputs at different scales may have different shapes.

In the following example, we use the same batch of data to construct feature maps of two different scales, Y1 and Y2. Here, Y2 has half the height and half the width of Y1. Using category prediction as an example, we assume that each element in the Y1 and Y2 feature maps generates five (Y1) or three (Y2) anchor boxes. When there are 10 object categories, the number of category prediction output channels is either $5 \times (10 + 1) = 55$ or $3 \times (10 + 1) = 33$. The format of the prediction output is (batch size, number of channels, height, width). As you can see, except for the batch size, the sizes of the other dimensions are different. Therefore, we must transform them into a consistent format and concatenate the predictions of the multiple scales to facilitate subsequent computation.

```
In [3]: def forward(x, block):
            block.initialize()
            return block(x)

        Y1 = forward(nd.zeros((2, 8, 20, 20)), cls_predictor(5, 10))
        Y2 = forward(nd.zeros((2, 16, 10, 10)), cls_predictor(3, 10))
        (Y1.shape, Y2.shape)
Out[3]: ((2, 55, 20, 20), (2, 33, 10, 10))
```

The channel dimension contains the predictions for all anchor boxes with the same center. We first move the channel dimension to the final dimension. Because the batch size is the same for all scales, we can convert the prediction results to binary format (batch size, height × width × number of channels) to facilitate subsequent concatenation on the 1st dimension.

```
In [4]: def flatten_pred(pred):
            return pred.transpose((0, 2, 3, 1)).flatten()

        def concat_preds(preds):
            return nd.concat(*[flatten_pred(p) for p in preds], dim=1)
```

Thus, regardless of the different shapes of Y1 and Y2, we can still concatenate the prediction results for the two different scales of the same batch.

```
In [5]: concat_preds([Y1, Y2]).shape
Out[5]: (2, 25300)
```

Height and Width Downsample Block

For multiscale object detection, we define the following down_sample_blk block, which reduces the height and width by 50%. This block consists of two 3×3 convolutional layers with a padding of 1 and a 2×2 maximum pooling layer with a stride of 2 connected in a series. As we know, 3×3 convolutional layers with a padding of 1 do not change the shape of feature maps. However, the subsequent pooling layer directly reduces the size of the feature map by half. Because $1 \times 2 + (3 - 1) + (3 - 1) = 6$, each element in the output feature map has a receptive field on the input feature map of the shape 6×6. As you can see, the height and width downsample block enlarges the receptive field of each element in the output feature map.

```
In [6]: def down_sample_blk(num_channels):
            blk = nn.Sequential()
            for _ in range(2):
                blk.add(nn.Conv2D(num_channels, kernel_size=3, padding=1),
                        nn.BatchNorm(in_channels=num_channels),
                        nn.Activation('relu'))
            blk.add(nn.MaxPool2D(2))
            return blk
```

By testing forward computation in the height and width downsample block, we can see that it changes the number of input channels and halves the height and width.

```
In [7]: forward(nd.zeros((2, 3, 20, 20)), down_sample_blk(10)).shape
Out[7]: (2, 10, 10, 10)
```

Base Network Block

The base network block is used to extract features from original images. To simplify the computation, we will construct a small base network. This network consists of three height and width downsample blocks connected in a series, so it doubles the number of channels at each step. When we input an original image with the shape 256×256, the base network block outputs a feature map with the shape 32×32.

10.7. Single Shot Multibox Detection (SSD) 471

```
In [8]: def base_net():
            blk = nn.Sequential()
            for num_filters in [16, 32, 64]:
                blk.add(down_sample_blk(num_filters))
            return blk

        forward(nd.zeros((2, 3, 256, 256)), base_net()).shape
Out[8]: (2, 64, 32, 32)
```

The Complete Model

The SSD model contains a total of five modules. Each module outputs a feature map used to generate anchor boxes and predict the categories and offsets of these anchor boxes. The first module is the base network block, modules two to four are height and width downsample blocks, and the fifth module is a global maximum pooling layer that reduces the height and width to 1. Therefore, modules two to five are all multiscale feature blocks shown in Figure 9.4.

```
In [9]: def get_blk(i):
            if i == 0:
                blk = base_net()
            elif i == 4:
                blk = nn.GlobalMaxPool2D()
            else:
                blk = down_sample_blk(128)
            return blk
```

Now, we will define the forward computation process for each module. In contrast to the previously-described convolutional neural networks, this module not only returns feature map `Y` output by convolutional computation, but also the anchor boxes of the current scale generated from `Y` and their predicted categories and offsets.

```
In [10]: def blk_forward(X, blk, size, ratio, cls_predictor, bbox_predictor):
             Y = blk(X)
             anchors = contrib.ndarray.MultiBoxPrior(Y, sizes=size, ratios=ratio)
             cls_preds = cls_predictor(Y)
             bbox_preds = bbox_predictor(Y)
             return (Y, anchors, cls_preds, bbox_preds)
```

As we mentioned, the closer a multiscale feature block is to the top in Figure 9.4, the larger the objects it detects and the larger the anchor boxes it must generate. Here, we first divide the interval from 0.2 to 1.05 into five equal parts to determine the sizes of smaller anchor boxes at different scales: 0.2, 0.37, 0.54, etc. Then, according to $\sqrt{0.2 \times 0.37} = 0.272$, $\sqrt{0.37 \times 0.54} = 0.447$, and similar formulas, we determine the sizes of larger anchor boxes at the different scales.

```
In [11]: sizes = [[0.2, 0.272], [0.37, 0.447], [0.54, 0.619], [0.71, 0.79],
                  [0.88, 0.961]]
         ratios = [[1, 2, 0.5]] * 5
         num_anchors = len(sizes[0]) + len(ratios[0]) - 1
```

Now, we can define the complete model, `TinySSD`.

```
In [12]: class TinySSD(nn.Block):
             def __init__(self, num_classes, **kwargs):
```

```
                super(TinySSD, self).__init__(**kwargs)
                self.num_classes = num_classes
                for i in range(5):
                    # The assignment statement is self.blk_i = get_blk(i)
                    setattr(self, 'blk_%d' % i, get_blk(i))
                    setattr(self, 'cls_%d' % i, cls_predictor(num_anchors,
                                                              num_classes))
                    setattr(self, 'bbox_%d' % i, bbox_predictor(num_anchors))

            def forward(self, X):
                anchors, cls_preds, bbox_preds = [None] * 5, [None] * 5, [None] * 5
                for i in range(5):
                    # getattr(self, 'blk_%d' % i) accesses self.blk_i
                    X, anchors[i], cls_preds[i], bbox_preds[i] = blk_forward(
                        X, getattr(self, 'blk_%d' % i), sizes[i], ratios[i],
                        getattr(self, 'cls_%d' % i), getattr(self, 'bbox_%d' % i))
                # In the reshape function, 0 indicates that the batch size remains
                # unchanged
                return (nd.concat(*anchors, dim=1),
                        concat_preds(cls_preds).reshape(
                            (0, -1, self.num_classes + 1)), concat_preds(bbox_preds))
```

We now create an SSD model instance and use it to perform forward computation on image mini-batch X, which has a height and width of 256 pixels. As we verified previously, the first module outputs a feature map with the shape 32×32. Because modules two to four are height and width downsample blocks, module five is a global pooling layer, and each element in the feature map is used as the center for 4 anchor boxes, a total of $(32^2 + 16^2 + 8^2 + 4^2 + 1) \times 4 = 5444$ anchor boxes are generated for each image at the five scales.

```
In [13]: net = TinySSD(num_classes=1)
         net.initialize()
         X = nd.zeros((32, 3, 256, 256))
         anchors, cls_preds, bbox_preds = net(X)

         print('output anchors:', anchors.shape)
         print('output class preds:', cls_preds.shape)
         print('output bbox preds:', bbox_preds.shape)

output anchors: (1, 5444, 4)
output class preds: (32, 5444, 2)
output bbox preds: (32, 21776)
```

10.7.2 Training

Now, we will explain, step by step, how to train the SSD model for object detection.

Data Reading and Initialization

We read the Pikachu data set we created in the previous section.

```
In [14]: batch_size = 32
         train_iter, _ = d2l.load_data_pikachu(batch_size)
```

There is 1 category in the Pikachu data set. After defining the module, we need to initialize the model parameters and define the optimization algorithm.

```
In [15]: ctx, net = d2l.try_gpu(), TinySSD(num_classes=1)
         net.initialize(init=init.Xavier(), ctx=ctx)
         trainer = gluon.Trainer(net.collect_params(), 'sgd',
                                 {'learning_rate': 0.2, 'wd': 5e-4})
```

Define Loss and Evaluation Functions

Object detection is subject to two types of losses. The first is anchor box category loss. For this, we can simply reuse the cross-entropy loss function we used in image classification. The second loss is positive anchor box offset loss. Offset prediction is a normalization problem. However, here, we do not use the squared loss introduced previously. Rather, we use the L_1 norm loss, which is the absolute value of the difference between the predicted value and the ground-truth value. The mask variable bbox_masks removes negative anchor boxes and padding anchor boxes from the loss calculation. Finally, we add the anchor box category and offset losses to find the final loss function for the model.

```
In [16]: cls_loss = gloss.SoftmaxCrossEntropyLoss()
         bbox_loss = gloss.L1Loss()

         def calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels, bbox_masks):
             cls = cls_loss(cls_preds, cls_labels)
             bbox = bbox_loss(bbox_preds * bbox_masks, bbox_labels * bbox_masks)
             return cls + bbox
```

We can use the accuracy rate to evaluate the classification results. As we use the L_1 norm loss, we will use the average absolute error to evaluate the bounding box prediction results.

```
In [17]: def cls_eval(cls_preds, cls_labels):
             # Because the category prediction results are placed in the final
             # dimension, argmax must specify this dimension
             return (cls_preds.argmax(axis=-1) == cls_labels).sum().asscalar()

         def bbox_eval(bbox_preds, bbox_labels, bbox_masks):
             return ((bbox_labels - bbox_preds) * bbox_masks).abs().sum().asscalar()
```

Train the Model

During model training, we must generate multiscale anchor boxes (anchors) in the model's forward computation process and predict the category (cls_preds) and offset (bbox_preds) for each anchor box. Afterwards, we label the category (cls_labels) and offset (bbox_labels) of each generated anchor box based on the label information Y. Finally, we calculate the loss function using the predicted and labeled category and offset values. To simplify the code, we do not evaluate the training data set here.

```
In [18]: for epoch in range(20):
             acc_sum, mae_sum, n, m = 0.0, 0.0, 0, 0
             train_iter.reset()  # Read data from the start.
             start = time.time()
             for batch in train_iter:
                 X = batch.data[0].as_in_context(ctx)
```

```
            Y = batch.label[0].as_in_context(ctx)
            with autograd.record():
                # Generate multiscale anchor boxes and predict the category and
                # offset of each
                anchors, cls_preds, bbox_preds = net(X)
                # Label the category and offset of each anchor box
                bbox_labels, bbox_masks, cls_labels = contrib.nd.MultiBoxTarget(
                    anchors, Y, cls_preds.transpose((0, 2, 1)))
                # Calculate the loss function using the predicted and labeled
                # category and offset values
                l = calc_loss(cls_preds, cls_labels, bbox_preds, bbox_labels,
                              bbox_masks)
            l.backward()
            trainer.step(batch_size)
            acc_sum += cls_eval(cls_preds, cls_labels)
            n += cls_labels.size
            mae_sum += bbox_eval(bbox_preds, bbox_labels, bbox_masks)
            m += bbox_labels.size

        if (epoch + 1) % 5 == 0:
            print('epoch %2d, class err %.2e, bbox mae %.2e, time %.1f sec' % (
                epoch + 1, 1 - acc_sum / n, mae_sum / m, time.time() - start))

epoch  5, class err 2.98e-03, bbox mae 3.17e-03, time 13.7 sec
epoch 10, class err 2.61e-03, bbox mae 2.92e-03, time 13.7 sec
epoch 15, class err 2.45e-03, bbox mae 2.71e-03, time 13.6 sec
epoch 20, class err 2.49e-03, bbox mae 2.63e-03, time 13.7 sec
```

10.7.3 Prediction

In the prediction stage, we want to detect all objects of interest in the image. Below, we read the test image and transform its size. Then, we convert it to the four-dimensional format required by the convolutional layer.

```
In [19]: img = image.imread('../img/pikachu.jpg')
         feature = image.imresize(img, 256, 256).astype('float32')
         X = feature.transpose((2, 0, 1)).expand_dims(axis=0)
```

Using the `MultiBoxDetection` function, we predict the bounding boxes based on the anchor boxes and their predicted offsets. Then, we use non-maximum suppression to remove similar bounding boxes.

```
In [20]: def predict(X):
             anchors, cls_preds, bbox_preds = net(X.as_in_context(ctx))
             cls_probs = cls_preds.softmax().transpose((0, 2, 1))
             output = contrib.nd.MultiBoxDetection(cls_probs, bbox_preds, anchors)
             idx = [i for i, row in enumerate(output[0]) if row[0].asscalar() != -1]
             return output[0, idx]

         output = predict(X)
```

Finally, we take all the bounding boxes with a confidence level of at least 0.3 and display them as the final output.

```
In [21]: d2l.set_figsize((5, 5))
```

```
def display(img, output, threshold):
    fig = d2l.plt.imshow(img.asnumpy())
    for row in output:
        score = row[1].asscalar()
        if score < threshold:
            continue
        h, w = img.shape[0:2]
        bbox = [row[2:6] * nd.array((w, h, w, h), ctx=row.context)]
        d2l.show_bboxes(fig.axes, bbox, '%.2f' % score, 'w')

display(img, output, threshold=0.3)
```
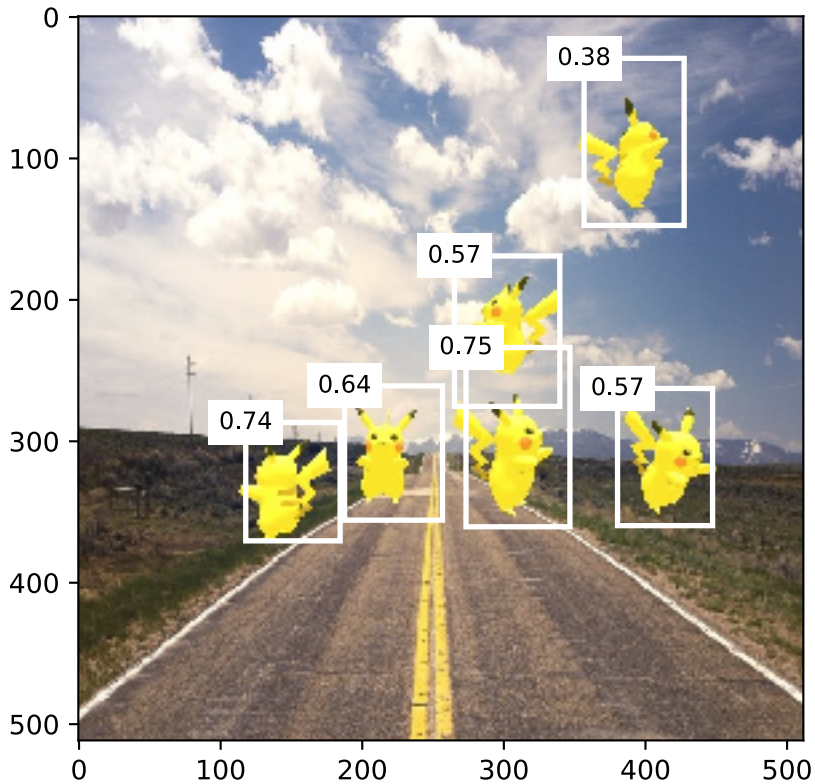

Summary

- SSD is a multiscale object detection model. This model generates different numbers of anchor boxes of different sizes based on the base network block and each multiscale feature block and predicts the categories and offsets of the anchor boxes to detect objects of different sizes.

- During SSD model training, the loss function is calculated using the predicted and labeled category and offset values.

Exercises

- Due to space limitations, we have ignored some of the implementation details of SSD models in this experiment. Can you further improve the model in the following areas?

Loss Function

For the predicted offsets, replace L_1 norm loss with L_1 regularization loss. This loss function uses a square function around zero for greater smoothness. This is the regularized area controlled by the hyper-parameter σ:

$$f(x) = \begin{cases} (\sigma x)^2/2, & \text{if } |x| < 1/\sigma^2 \\ |x| - 0.5/\sigma^2, & \text{otherwise} \end{cases}$$

When σ is large, this loss is similar to the L_1 norm loss. When the value is small, the loss function is smoother.

```
In [22]: sigmas = [10, 1, 0.5]
         lines = ['-', '--', '-.']
         x = nd.arange(-2, 2, 0.1)
         d2l.set_figsize()

         for l, s in zip(lines, sigmas):
             y = nd.smooth_l1(x, scalar=s)
             d2l.plt.plot(x.asnumpy(), y.asnumpy(), l, label='sigma=%.1f' % s)
         d2l.plt.legend();
```
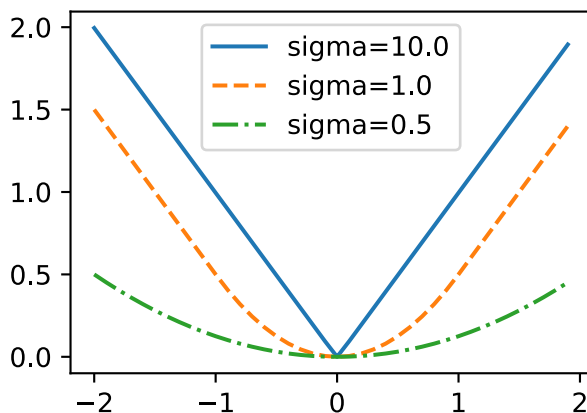

In the experiment, we used cross-entropy loss for category prediction. Now, assume that the prediction probability of the actual category j is p_j and the cross-entropy loss is $-\log p_j$. We can also use the focal loss[2]. Given the positive hyper-parameters γ and α, this loss is defined as:

$$-\alpha(1 - p_j)^\gamma \log p_j.$$

10.7. Single Shot Multibox Detection (SSD)

As you can see, by increasing γ, we can effectively reduce the loss when the probability of predicting the correct category is high.
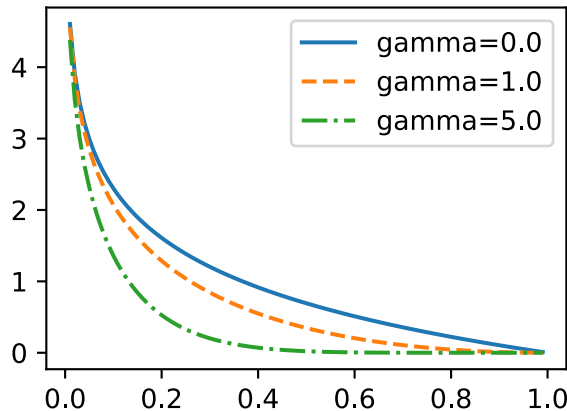
```
In [23]: def focal_loss(gamma, x):
             return -(1 - x) ** gamma * x.log()

         x = nd.arange(0.01, 1, 0.01)
         for l, gamma in zip(lines, [0, 1, 5]):
             y = d2l.plt.plot(x.asnumpy(), focal_loss(gamma, x).asnumpy(), l,
                              label='gamma=%.1f' % gamma)
         d2l.plt.legend();
```


Training and Prediction

- When an object is relatively large compared to the image, the model normally adopts a larger input image size.

- This generally produces a large number of negative anchor boxes when labeling anchor box categories. We can sample the negative anchor boxes to better balance the data categories. To do this, we can set the `MultiBoxTarget` function's `negative_mining_ratio` parameter.

- Assign hyper-parameters with different weights to the anchor box category loss and positive anchor box offset loss in the loss function.

- Refer to the SSD paper. What methods can be used to evaluate the precision of object detection models[1]?

References

[1] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C. Y., & Berg, A. C. (2016, October). Ssd: Single shot multibox detector. In European conference on computer vision (pp. 21-37). Springer, Cham.

[2] Lin, T. Y., Goyal, P., Girshick, R., He, K., & Dollár, P. (2018). Focal loss for dense object detection. IEEE transactions on pattern analysis and machine intelligence.

Scan the QR Code to Discuss

10.8 Region-based CNNs (R-CNNs)

Region-based convolutional neural networks or regions with CNN features (R-CNNs) are a pioneering approach that applies deep models to object detection[1]. In this section, we will discuss R-CNNs and a series of improvements made to them: Fast R-CNN[3], Faster R-CNN[4], and Mask R-CNN[5]. Due to space limitations, we will confine our discussion to the designs of these models.

10.8.1 R-CNNs

R-CNN models first select several proposed regions from an image (for example, anchor boxes are one type of selection method) and then label their categories and bounding boxes (e.g., offsets). Then, they use a CNN to perform forward computation to extract features from each proposed area. Afterwards, we use the features of each proposed region to predict their categories and bounding boxes. Figure 9.5 shows an R-CNN model.
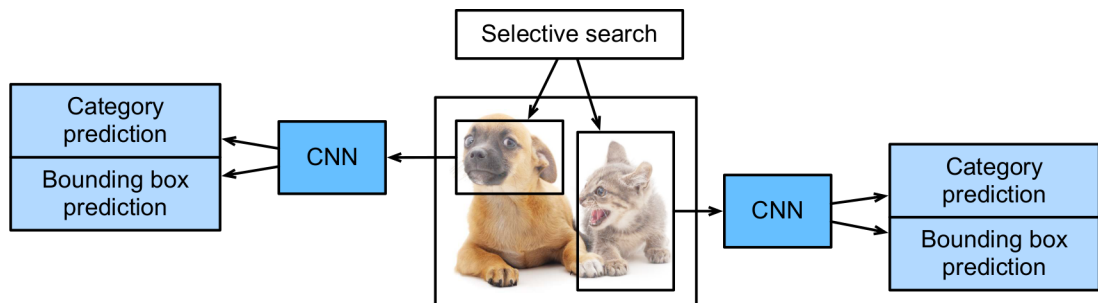

Fig. 10.5: R-CNN model.

Specifically, R-CNNs are composed of four main parts:

1. Selective search is performed on the input image to select multiple high-quality proposed regions[2]. These proposed regions are generally selected on multiple scales and have different shapes and sizes. The category and ground-truth bounding box of each proposed region is labeled.

2. A pre-trained CNN is selected and placed, in truncated form, before the output layer. It transforms each proposed region into the input dimensions required by the network and uses forward computation to output the features extracted from the proposed regions.

3. The features and labeled category of each proposed region are combined as an example to train multiple support vector machines for object classification. Here, each support vector machine is used to determine whether an example belongs to a certain category.

4. The features and labeled bounding box of each proposed region are combined as an example to train a linear regression model for ground-truth bounding box prediction.

Although R-CNN models use pre-trained CNNs to effectively extract image features, the main downside is the slow speed. As you can imagine, we can select thousands of proposed regions from a single image, requiring thousands of forward computations from the CNN to perform object detection. This massive computing load means that R-CNNs are not widely used in actual applications.

10.8.2 Fast R-CNN

The main performance bottleneck of an R-CNN model is the need to independently extract features for each proposed region. As these regions have a high degree of overlap, independent feature extraction results in a high volume of repetitive computations. Fast R-CNN improves on the R-CNN by only performing CNN forward computation on the image as a whole.
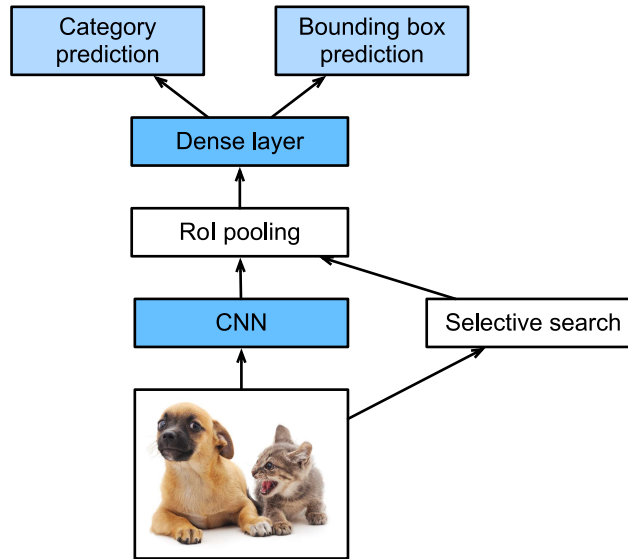
Fig. 10.6: Fast R-CNN model.

Figure 9.6 shows a Fast R-CNN model. It's primary computation steps are described below:

1. Compared to an R-CNN model, a Fast R-CNN model uses the entire image as the CNN input for feature extraction, rather than each proposed region. Moreover, this network is generally trained to update the model parameters. As the input is an entire image, the CNN output shape is $1 \times c \times h_1 \times w_1$.

2. Assuming selective search generates n proposed regions, their different shapes indicate regions of interests (RoIs) of different shapes on the CNN output. Features of the same shapes must be extracted from these RoIs (here we assume that the height is h_2 and the width is w_2). Fast R-CNN introduces RoI pooling, which uses the CNN output and RoIs as input to output a concatenation of the features extracted from each proposed region with the shape $n \times c \times h_2 \times w_2$.

3. A fully connected layer is used to transform the output shape to $n \times d$, where d is determined by the model design.

4. During category prediction, the shape of the fully connected layer output is again transformed to $n \times q$ and we use software regression (q is the number of categories). During bounding box prediction, the shape of the fully connected layer output is again transformed to $n \times 4$. This means that we predict the category and bounding box for each proposed region.

The RoI pooling layer in Fast R-CNN is somewhat different from the pooling layers we have discussed before. In a normal pooling layer, we set the pooling window, padding, and stride to control the output shape. In an RoI pooling layer, we can directly specify the output shape of each region, such as specifying the height and width of each region as h_2, w_2. Assuming that the height and width of the RoI window are h and w, this window is divided into a grid of sub-windows with the shape $h_2 \times w_2$. The size of each

sub-window is about $(h/h_2) \times (w/w_2)$. The sub-window height and width must always be integers and the largest element is used as the output for a given sub-window. This allows the RoI pooling layer to extract features of the same shape from RoIs of different shapes.

In Figure 9.7, we select an 3×3 region as an RoI of the 4×4 input. For this RoI, we use a 2×2 RoI pooling layer to obtain a single 2×2 output. When we divide the region into four sub-windows, they respectively contain the elements 0, 1, 4, and 5 (5 is the largest); 2 and 6 (6 is the largest); 8 and 9 (9 is the largest); and 10.
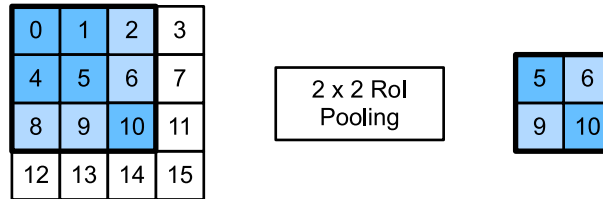
Fig. 10.7: 2×2 RoI pooling layer.

We use the `ROIPooling` function to demonstrate the RoI pooling layer computation. Assume that the CNN extracts the feature X with both a height and width of 4 and only a single channel.

```
In [1]: from mxnet import nd

        X = nd.arange(16).reshape((1, 1, 4, 4))
        X
Out[1]:
        [[[[ 0.  1.  2.  3.]
          [ 4.  5.  6.  7.]
          [ 8.  9. 10. 11.]
          [12. 13. 14. 15.]]]]
        <NDArray 1x1x4x4 @cpu(0)>
```

Assume that the height and width of the image are both 40 pixels and that selective search generates two proposed regions on the image. Each region is expressed as five elements: the region's object category and the x, y coordinates of its upper-left and bottom-right corners.

```
In [2]: rois = nd.array([[0, 0, 0, 20, 20], [0, 0, 10, 30, 30]])
```

Because the height and width of X are $1/10$ of the height and width of the image, the coordinates of the two proposed regions are multiplied by 0.1 according to the `spatial_scale`, and then the RoIs are labeled on X as `X[:,:,0:3,0:3]` and `X[:,:,1:4,0:4]`, respectively. Finally, we divide the two RoIs into a sub-window grid and extract features with a height and width of 2.

```
In [3]: nd.ROIPooling(X, rois, pooled_size=(2, 2), spatial_scale=0.1)
Out[3]:
        [[[[ 5.  6.]
          [ 9. 10.]]]]
```

```
[[[ 9. 11.]
  [13. 15.]]]]
<NDArray 2x1x2x2 @cpu(0)>
```

10.8.3 Faster R-CNN

In order to obtain precise object detection results, Fast R-CNN generally requires that many proposed regions be generated in selective search. Faster R-CNN replaces selective search with a region proposal network. This reduces the number of proposed regions generated, while ensuring precise object detection.
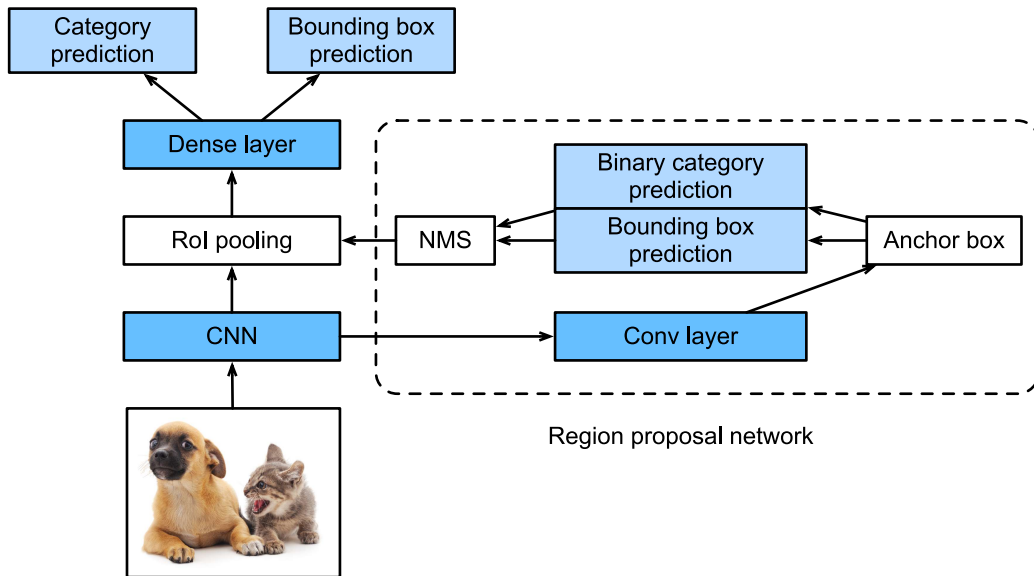
Fig. 10.8: Faster R-CNN model.

Figure 9.8 shows a Faster R-CNN model. Compared to Fast R-CNN, Faster R-CNN only changes the method for generating proposed regions from selective search to region proposal network. The other parts of the model remain unchanged. The detailed region proposal network computation process is described below:

1. We use a 3×3 convolutional layer with a padding of 1 to transform the CNN output and set the number of output channels to c. This way, each element in the feature map the CNN extracts from the image is a new feature with a length of c.

2. We use each element in the feature map as a center to generate multiple anchor boxes of different sizes and aspect ratios and then label them.

3. We use the features of the elements of length c at the center on the anchor boxes to predict the binary category (object or background) and bounding box for their respective anchor boxes.

4. Then, we use non-maximum suppression to remove similar bounding box results that correspond to category predictions of object. Finally, we output the predicted bounding boxes as the proposed regions required by the RoI pooling layer.

It is worth noting that, as a part of the Faster R-CNN model, the region proposal network is trained together with the rest of the model. In addition, the Faster R-CNN object functions include the category and bounding box predictions in object detection, as well as the binary category and bounding box predictions for the anchor boxes in the region proposal network. Finally, the region proposal network can learn how to generate high-quality proposed regions, which reduces the number of proposed regions while maintaining the precision of object detection.

10.8.4 Mask R-CNN

If training data is labeled with the pixel-level positions of each object in an image, a Mask R-CNN model can effectively use these detailed labels to further improve the precision of object detection.
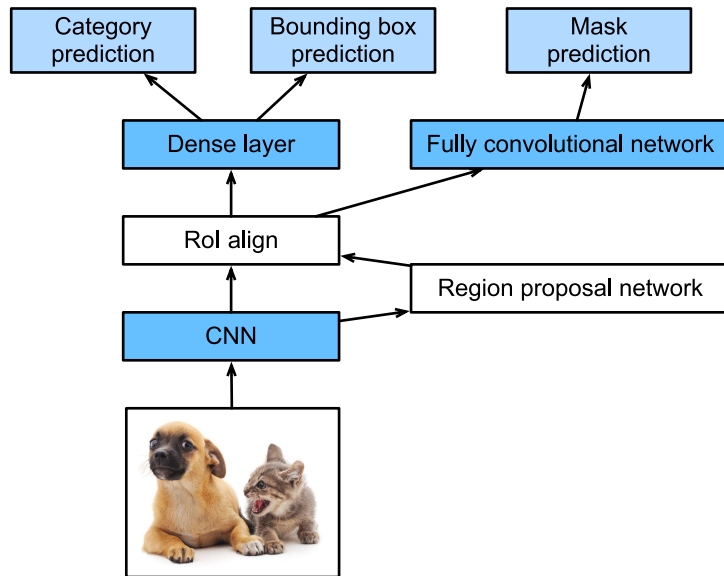
Fig. 10.9: Mask R-CNN model.

As shown in 9.9, Mask R-CNN is a modification to the Faster R-CNN model. Mask R-CNN models replace the RoI pooling layer with an RoI alignment layer. This allows the use of bilinear interpolation to retain spatial information on feature maps, making Mask R-CNN better suited for pixel-level predictions. The RoI alignment layer outputs feature maps of the same shape for all RoIs. This not only predicts the categories and bounding boxes of RoIs, but allows us to use an additional fully convolutional network to predict the pixel-level positions of objects. We will describe how to use fully convolutional networks to predict pixel-level semantics in images later in this chapter.

Summary

- An R-CNN model selects several proposed regions and uses a CNN to perform forward computation and extract the features from each proposed region. It then uses these features to predict the categories and bounding boxes of proposed regions.

- Fast R-CNN improves on the R-CNN by only performing CNN forward computation on the image as a whole. It introduces an RoI pooling layer to extract features of the same shape from RoIs of different shapes.

- Faster R-CNN replaces the selective search used in Fast R-CNN with a region proposal network. This reduces the number of proposed regions generated, while ensuring precise object detection.

- Mask R-CNN uses the same basic structure as Faster R-CNN, but adds a fully convolution layer to help locate objects at the pixel level and further improve the precision of object detection.

Exercises

- Study the implementation of each model in the GluonCV toolkit related to this section[6].

References

[1] Girshick, R., Donahue, J., Darrell, T., & Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 580-587).

[2] Uijlings, J. R., Van De Sande, K. E., Gevers, T., & Smeulders, A. W. (2013). Selective search for object recognition. International journal of computer vision, 104(2), 154-171.

[3] Girshick, R. (2015). Fast r-cnn. arXiv preprint arXiv:1504.08083.

[4] Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster r-cnn: Towards real-time object detection with region proposal networks. In Advances in neural information processing systems (pp. 91-99).

[5] He, K., Gkioxari, G., Dollár, P., & Girshick, R. (2017, October). Mask r-cnn. In Computer Vision (ICCV), 2017 IEEE International Conference on (pp. 2980-2988). IEEE.

[6] GluonCV Toolkit. https://gluon-cv.mxnet.io/

Scan the QR Code to Discuss

10.9 Semantic Segmentation and Data Sets

In our discussion of object detection issues in the previous sections, we only used rectangular bounding boxes to label and predict objects in images. In this section, we will look at semantic segmentation, which attempts to segment images into regions with different semantic categories. These semantic regions label and predict objects at the pixel level. Figure 9.10 shows a semantically-segmented image, with areas labeled dog, cat, and background. As you can see, compared to object detection, semantic segmentation labels areas with pixel-level borders, for significantly greater precision.
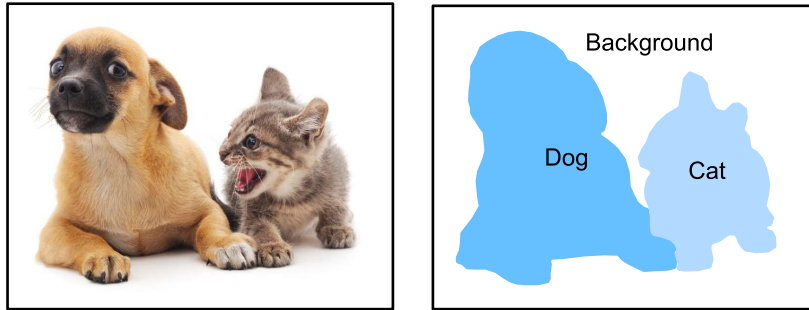
Fig. 10.10: Semantically-segmented image, with areas labeled dog, cat, and background.

10.9.1 Image Segmentation and Instance Segmentation

In the computer vision field, there are two important methods related to semantic segmentation: image segmentation and instance segmentation. Here, we will distinguish these concepts from semantic segmentation as follows:

- Image segmentation divides an image into several constituent regions. This method generally uses the correlations between pixels in an image. During training, labels are not needed for image pixels. However, during prediction, this method cannot ensure that the segmented regions have the semantics we want. If we input the image in 9.10, image segmentation might divide the dog into two regions, one covering the dog's mouth and eyes where black is the prominent color and the other covering the rest of the dog where yellow is the prominent color.

- Instance segmentation is also called simultaneous detection and segmentation. This method attempts to identify the pixel-level regions of each object instance in an image. In contrast to semantic segmentation, instance segmentation not only distinguishes semantics, but also different object instances. If an image contains two dogs, instance segmentation will distinguish which pixels belong to which dog.

10.9.2 Pascal VOC2012 Semantic Segmentation Data Set

In the semantic segmentation field, one important data set is Pascal VOC2012[1]. To better understand this data set, we must first import the package or module needed for the experiment.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet import gluon, image, nd
        from mxnet.gluon import data as gdata, utils as gutils
        import os
        import sys
        import tarfile
```

We download the archive containing this data set to the `../data` path. The archive is about 2GB, so it will take some time to download. After you decompress the archive, the data set is located in the `../data/VOCdevkit/VOC2012` path.

```
In [2]: # This function has been saved in the d2l package for future use
        def download_voc_pascal(data_dir='../data'):
            voc_dir = os.path.join(data_dir, 'VOCdevkit/VOC2012')
            url = ('http://host.robots.ox.ac.uk/pascal/VOC/voc2012'
                   '/VOCtrainval_11-May-2012.tar')
            sha1 = '4e443f8a2eca6b1dac8a6c57641b67dd40621a49'
            fname = gutils.download(url, data_dir, sha1_hash=sha1)
            with tarfile.open(fname, 'r') as f:
                f.extractall(data_dir)
            return voc_dir

        voc_dir = download_voc_pascal()
```

Go to `../data/VOCdevkit/VOC2012` to see the different parts of the data set. The `ImageSets/Segmentation` path contains text files that specify the training and testing examples. The `JPEGImages` and `SegmentationClass` paths contain the example input images and labels, respectively. These labels are also in image format, with the same dimensions as the input images to which they correspond. In the labels, pixels with the same color belong to the same semantic category. The `read_voc_images` function defined below reads all input images and labels to the memory.

```
In [3]: # This function has been saved in the d2l package for future use
        def read_voc_images(root=voc_dir, is_train=True):
            txt_fname = '%s/ImageSets/Segmentation/%s' % (
                root, 'train.txt' if is_train else 'val.txt')
            with open(txt_fname, 'r') as f:
                images = f.read().split()
            features, labels = [None] * len(images), [None] * len(images)
            for i, fname in enumerate(images):
                features[i] = image.imread('%s/JPEGImages/%s.jpg' % (root, fname))
                labels[i] = image.imread(
                    '%s/SegmentationClass/%s.png' % (root, fname))
            return features, labels

        train_features, train_labels = read_voc_images()
```

We draw the first five input images and their labels. In the label images, white represents borders and black represents the background. Other colors correspond to different categories.

```
In [4]: n = 5
        imgs = train_features[0:n] + train_labels[0:n]
        d2l.show_images(imgs, 2, n);
```
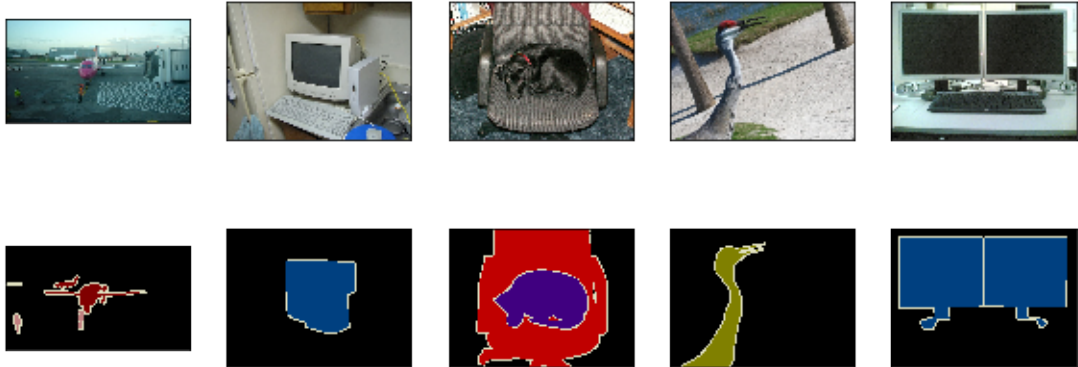

Next, we list each RGB color value in the labels and the categories they label.

```
In [5]: # This constant has been saved in the d2l package for future use
        VOC_COLORMAP = [[0, 0, 0], [128, 0, 0], [0, 128, 0], [128, 128, 0],
                        [0, 0, 128], [128, 0, 128], [0, 128, 128], [128, 128, 128],
                        [64, 0, 0], [192, 0, 0], [64, 128, 0], [192, 128, 0],
                        [64, 0, 128], [192, 0, 128], [64, 128, 128], [192, 128, 128],
                        [0, 64, 0], [128, 64, 0], [0, 192, 0], [128, 192, 0],
                        [0, 64, 128]]
        # This constant has been saved in the d2l package for future use
        VOC_CLASSES = ['background', 'aeroplane', 'bicycle', 'bird', 'boat',
                       'bottle', 'bus', 'car', 'cat', 'chair', 'cow',
                       'diningtable', 'dog', 'horse', 'motorbike', 'person',
                       'potted plant', 'sheep', 'sofa', 'train', 'tv/monitor']
```

After defining the two constants above, we can easily find the category index for each pixel in the labels.

```
In [6]: colormap2label = nd.zeros(256 ** 3)
        for i, colormap in enumerate(VOC_COLORMAP):
            colormap2label[(colormap[0] * 256 + colormap[1]) * 256 + colormap[2]] = i

        # This function has been saved in the d2l package for future use
        def voc_label_indices(colormap, colormap2label):
            colormap = colormap.astype('int32')
            idx = ((colormap[:, :, 0] * 256 + colormap[:, :, 1]) * 256
                   + colormap[:, :, 2])
            return colormap2label[idx]
```

For example, in the first example image, the category index for the front part of the airplane is 1 and the index for the background is 0.

```
In [7]: y = voc_label_indices(train_labels[0], colormap2label)
        y[105:115, 130:140], VOC_CLASSES[1]
```

```
Out[7]: (
        [[0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
         [0. 0. 0. 0. 0. 0. 0. 1. 1. 1.]
         [0. 0. 0. 0. 0. 0. 1. 1. 1. 1.]
         [0. 0. 0. 0. 0. 1. 1. 1. 1. 1.]
         [0. 0. 0. 0. 0. 1. 1. 1. 1. 1.]
         [0. 0. 0. 0. 1. 1. 1. 1. 1. 1.]
         [0. 0. 0. 0. 0. 1. 1. 1. 1. 1.]
         [0. 0. 0. 0. 0. 1. 1. 1. 1. 1.]
         [0. 0. 0. 0. 0. 0. 1. 1. 1. 1.]
         [0. 0. 0. 0. 0. 0. 0. 0. 1. 1.]]
        <NDArray 10x10 @cpu(0)>, 'aeroplane')
```

Data Preprocessing

In the preceding chapters, we scaled images to make them fit the input shape of the model. In semantic segmentation, this method would require us to re-map the predicted pixel categories back to the original-size input image. It would be very difficult to do this precisely, especially in segmented regions with different semantics. To avoid this problem, we crop the images to set dimensions and do not scale them. Specifically, we use the random cropping method used in image augmentation to crop the same region from input images and their labels.

```
In [8]: # This function has been saved in the d2l package for future use
        def voc_rand_crop(feature, label, height, width):
            feature, rect = image.random_crop(feature, (width, height))
            label = image.fixed_crop(label, *rect)
            return feature, label

        imgs = []
        for _ in range(n):
            imgs += voc_rand_crop(train_features[0], train_labels[0], 200, 300)
        d2l.show_images(imgs[::2] + imgs[1::2], 2, n);
```

Data Set Classes for Custom Semantic Segmentation

We use the inherited `Dataset` class provided by Gluon to customize the semantic segmentation data set class `VOCSegDataset`. By implementing the `__getitem__` function, we can arbitrarily access the input image with the index `idx` and the category indexes for each of its pixels from the data set. As some images in the data set may be smaller than the output dimensions specified for random cropping, we must remove these example by using a custom `filter` function. In addition, we define the `normalize_image` function to normalize each of the three RGB channels of the input images.

```
In [9]:  # This class has been saved in the d2l package for future use
         class VOCSegDataset(gdata.Dataset):
             def __init__(self, is_train, crop_size, voc_dir, colormap2label):
                 self.rgb_mean = nd.array([0.485, 0.456, 0.406])
                 self.rgb_std = nd.array([0.229, 0.224, 0.225])
                 self.crop_size = crop_size
                 features, labels = read_voc_images(root=voc_dir, is_train=is_train)
                 self.features = [self.normalize_image(feature)
                                  for feature in self.filter(features)]
                 self.labels = self.filter(labels)
                 self.colormap2label = colormap2label
                 print('read ' + str(len(self.features)) + ' examples')

             def normalize_image(self, img):
                 return (img.astype('float32') / 255 - self.rgb_mean) / self.rgb_std

             def filter(self, imgs):
                 return [img for img in imgs if (
                     img.shape[0] >= self.crop_size[0] and
                     img.shape[1] >= self.crop_size[1])]

             def __getitem__(self, idx):
                 feature, label = voc_rand_crop(self.features[idx], self.labels[idx],
                                                *self.crop_size)
                 return (feature.transpose((2, 0, 1)),
                         voc_label_indices(label, self.colormap2label))

             def __len__(self):
                 return len(self.features)
```

Read the Data Set

Using the custom `VOCSegDataset` class, we create the training set and testing set instances. We assume the random cropping operation output images in the shape 320×480. Below, we can see the number of examples retained in the training and testing sets.

```
In [10]: crop_size = (320, 480)
         voc_train = VOCSegDataset(True, crop_size, voc_dir, colormap2label)
         voc_test = VOCSegDataset(False, crop_size, voc_dir, colormap2label)

read 1114 examples
read 1078 examples
```

We set the batch size to 64 and define the iterators for the training and testing sets.

```
In [11]: batch_size = 64
         num_workers = 0 if sys.platform.startswith('win32') else 4
         train_iter = gdata.DataLoader(voc_train, batch_size, shuffle=True,
                                       last_batch='discard', num_workers=num_workers)
         test_iter = gdata.DataLoader(voc_test, batch_size, last_batch='discard',
                                      num_workers=num_workers)
```

Print the shape of the first mini-batch. In contrast to image classification and object recognition, labels here are three-dimensional arrays.

```
In [12]: for X, Y in train_iter:
             print(X.shape)
             print(Y.shape)
             break

(64, 3, 320, 480)
(64, 320, 480)
```

Summary

- Semantic segmentation looks at how images can be segmented into regions with different semantic categories.

- In the semantic segmentation field, one important data set is Pascal VOC2012.

- Because the input images and labels in semantic segmentation have a one-to-one correspondence at the pixel level, we randomly crop them to a fixed size, rather than scaling them.

Exercises

- Recall the content we covered in the *Image Augmentation* section. Which of the image augmentation methods used in image classification would be hard to use in semantic segmentation?

Reference

[1] Pascal VOC2012 data set. http://host.robots.ox.ac.uk/pascal/VOC/voc2012/

Scan the QR Code to Discuss

10.10 Fully Convolutional Networks (FCN)

We previously discussed semantic segmentation using each pixel in an image for category prediction. A fully convolutional network (FCN) uses a convolutional neural network to transform image pixels to pixel categories. Unlike the convolutional neural networks previously introduced, an FCN transforms the height and width of the intermediate layer feature map back to the size of input image through the transposed convolution layer, so that the predictions have a one-to-one correspondence with input image in spatial dimension (height and width). Given a position on the spatial dimension, the output of the channel dimension will be a category prediction of the pixel corresponding to the location.

We will first import the package or module needed for the experiment and then explain the transposed convolution layer.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet import gluon, image, init, nd
        from mxnet.gluon import data as gdata, loss as gloss, model_zoo, nn
        import numpy as np
        import sys
```

10.10.1 Transposed Convolution Layer

The transposed convolution layer takes its name from the matrix transposition operation. In fact, convolution operations can also be achieved by matrix multiplication. In the example below, we define input X with a height and width of 4 respectively, and a convolution kernel K with a height and width of 3 respectively. Print the output of the 2D convolution operation and the convolution kernel. As you can see, the output has a height and a width of 2.

```
In [2]: X = nd.arange(1, 17).reshape((1, 1, 4, 4))
        K = nd.arange(1, 10).reshape((1, 1, 3, 3))
        conv = nn.Conv2D(channels=1, kernel_size=3)
        conv.initialize(init.Constant(K))
        conv(X), K

Out[2]: (
         [[[[348. 393.]
           [528. 573.]]]]
         <NDArray 1x1x2x2 @cpu(0)>,
         [[[[1. 2. 3.]
           [4. 5. 6.]
           [7. 8. 9.]]]]
         <NDArray 1x1x3x3 @cpu(0)>)
```

Next, we rewrite convolution kernel K as a sparse matrix W with a large number of zero elements, i.e. a weight matrix. The shape of the weight matrix is (4,16), where the non-zero elements are taken from the elements in convolution kernel K. Enter X and concatenate line by line to get a vector of length 16. Then, perform matrix multiplication for W and the X vector to get a vector of length 4. After the transformation,

we can get the same result as the convolution operation above. As you can see, in this example, we implement the convolution operation using matrix multiplication.

```
In [3]: W, k = nd.zeros((4, 16)), nd.zeros(11)
        k[:3], k[4:7], k[8:] = K[0, 0, 0, :], K[0, 0, 1, :], K[0, 0, 2, :]
        W[0, 0:11], W[1, 1:12], W[2, 4:15], W[3, 5:16] = k, k, k, k
        nd.dot(W, X.reshape(16)).reshape((1, 1, 2, 2)), W

Out[3]: (
          [[[[348. 393.]
            [528. 573.]]]]
          <NDArray 1x1x2x2 @cpu(0)>,
          [[1. 2. 3. 0. 4. 5. 6. 0. 7. 8. 9. 0. 0. 0. 0. 0.]
           [0. 1. 2. 3. 0. 4. 5. 6. 0. 7. 8. 9. 0. 0. 0. 0.]
           [0. 0. 0. 0. 1. 2. 3. 0. 4. 5. 6. 0. 7. 8. 9. 0.]
           [0. 0. 0. 0. 0. 1. 2. 3. 0. 4. 5. 6. 0. 7. 8. 9.]]
          <NDArray 4x16 @cpu(0)>)
```

Now we will describe the convolution operation from the perspective of matrix multiplication. Let the input vector be x and weight matrix be W. The implementation of the convolutional forward computation function can be considered as the multiplication of the function input by the weight matrix to output the vector $y = Wx$. We know that back propagation needs to be based on chain rules. Because $\nabla_x y = W^\top$, the implementation of the convolutional back propagation function can be considered as the multiplication of the function input by the transposed weight matrix $W^\top$. The transposed convolution layer exchanges the forward computation function and the back propagation function of the convolution layer. These two functions can be regarded as the multiplication of the function input vectors by $W^\top$ and W, respectively.

It is not difficult to see that the transposed convolution layer can be used to exchange the shape of input and output of the convolution layer. Let us continue to describe convolution using matrix multiplication. Let the weight matrix be a matrix with a shape of 4×16. For an input vector of length 16, the convolution forward computation outputs a vector with a length of 4. If the length of the input vector is 4 and the shape of the transpose weight matrix is 16×4, then the transposed convolution layer outputs a vector of length 16. In model design, transposed convolution layers are often used to transform smaller feature maps into larger ones. In a full convolutional network, when the input is a feature map with a high height and a wide width, the transposed convolution layer can be used to magnify the height and width to the size of the input image.

Now we will look at an example. Construct a convolution layer `conv` and let shape of input `X` be (1,3,64,64). The number of channels for convolution output `Y` is increased to 10, but the height and width are reduced by half.

```
In [4]: conv = nn.Conv2D(10, kernel_size=4, padding=1, strides=2)
        conv.initialize()

        X = nd.random.uniform(shape=(1, 3, 64, 64))
        Y = conv(X)
        Y.shape

Out[4]: (1, 10, 32, 32)
```

Next, we construct transposed convolution layer `conv_trans` by creating a `Conv2DTranspose` instance. Here, we assume the convolution kernel shape, padding, and stride of `conv_trans` are the same

with those in `conv`, and the number of output channels is 3. When the input is output `Y` of the convolution layer `conv`, the transposed convolution layer output has the same height and width as convolution layer input. The transposed convolution layer magnifies the height and width of the feature map by a factor of 2.

```
In [5]: conv_trans = nn.Conv2DTranspose(3, kernel_size=4, padding=1, strides=2)
        conv_trans.initialize()
        conv_trans(Y).shape

Out[5]: (1, 3, 64, 64)
```

In the literature, transposed convolution is also sometimes referred to as fractionally-strided convolution[2].

10.10.2 Construct a Model

Here, we demonstrate the most basic design of a fully convolutional network model. As shown in Figure 9.11, the fully convolutional network first uses the convolutional neural network to extract image features, then transforms the number of channels into the number of categories through the 1×1 convolution layer, and finally transforms the height and width of the feature map to the size of the input image by using the transposed convolution layer. The model output has the same height and width as the input image and has a one-to-one correspondence in spatial positions. The final output channel contains the category prediction of the pixel of the corresponding spatial position.
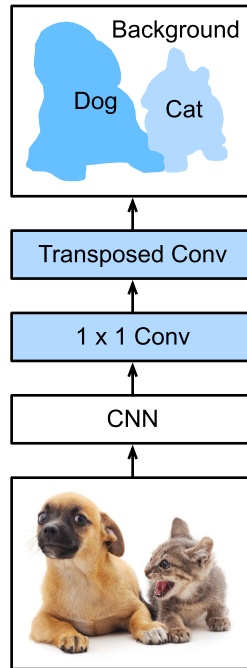
Fig. 10.11: Fully convolutional network.

Below, we use a ResNet-18 model pre-trained on the ImageNet data set to extract image features and record the network instance as `pretrained_net`. As you can see, the last two layers of the model member variable `features` are the global maximum pooling layer `GlobalAvgPool2D` and example flattening layer `Flatten`. The `output` module contains the fully connected layer used for output. These layers are not required for a fully convolutional network.

```
In [6]: pretrained_net = model_zoo.vision.resnet18_v2(pretrained=True)
        pretrained_net.features[-4:], pretrained_net.output

Out[6]: (HybridSequential(
           (0): BatchNorm(axis=1, eps=1e-05, momentum=0.9, fix_gamma=False,
↪  use_global_stats=False, in_channels=512)
           (1): Activation(relu)
           (2): GlobalAvgPool2D(size=(1, 1), stride=(1, 1), padding=(0, 0),
↪  ceil_mode=True)
           (3): Flatten
         ), Dense(512 -> 1000, linear))
```

Next, we create the fully convolutional network instance `net`. It duplicates all the neural layers except the last two layers of the instance member variable `features` of `pretrained_net` and the model parameters obtained after pre-training.

```
In [7]: net = nn.HybridSequential()
        for layer in pretrained_net.features[:-2]:
            net.add(layer)
```

Given an input of a height and width of 320 and 480 respectively, the forward computation of `net` will reduce the height and width of the input to $1/32$ of the original, i.e. 10 and 15.

```
In [8]: X = nd.random.uniform(shape=(1, 3, 320, 480))
        net(X).shape

Out[8]: (1, 512, 10, 15)
```

Next, we transform the number of output channels to the number of categories of Pascal VOC2012 (21) through the 1×1 convolution layer. Finally, we need to magnify the height and width of the feature map by a factor of 32 to change them back to the height and width of the input image. Recall the calculation method for the convolution layer output shape described in the section *Padding and Stride*. Because $(320 - 64 + 16 \times 2 + 32)/32 = 10$ and $(480 - 64 + 16 \times 2 + 32)/32 = 15$, we construct a transposed convolution layer with a stride of 32 and set the height and width of the convolution kernel to 64 and the padding to 16. It is not difficult to see that, if the stride is s, the padding is $s/2$ (assuming $s/2$ is an integer), and the height and width of the convolution kernel are $2s$, the transposed convolution kernel will magnify both the height and width of the input by a factor of s.

```
In [9]: num_classes = 21
        net.add(nn.Conv2D(num_classes, kernel_size=1),
                nn.Conv2DTranspose(num_classes, kernel_size=64, padding=16,
                                   strides=32))
```

10.10.3 Initialize the Transposed Convolution Layer

We already know that the transposed convolution layer can magnify a feature map. In image processing, sometimes we need to magnify the image, i.e. upsampling. There are many methods for upsampling, and one common method is bilinear interpolation. Simply speaking, in order to get the pixel of the output image at the coordinates (x, y), the coordinates are first mapped to the coordinates of the input image (x', y'). This can be done based on the ratio of the size of thee input to the size of the output. The mapped values x' and y' are usually real numbers. Then, we find the four pixels closest to the coordinate (x', y') on the input image. Finally, the pixels of the output image at coordinates (x, y) are calculated based on these four pixels on the input image and their relative distances to (x', y'). Upsampling by bilinear interpolation can be implemented by transposed convolution layer of the convolution kernel constructed using the following `bilinear_kernel` function. Due to space limitations, we only give the implementation of the `bilinear_kernel` function and will not discuss the principles of the algorithm.

```
In [10]: def bilinear_kernel(in_channels, out_channels, kernel_size):
             factor = (kernel_size + 1) // 2
             if kernel_size % 2 == 1:
                 center = factor - 1
             else:
                 center = factor - 0.5
             og = np.ogrid[:kernel_size, :kernel_size]
             filt = (1 - abs(og[0] - center) / factor) * \
                    (1 - abs(og[1] - center) / factor)
             weight = np.zeros((in_channels, out_channels, kernel_size, kernel_size),
                               dtype='float32')
             weight[range(in_channels), range(out_channels), :, :] = filt
             return nd.array(weight)
```

Now, we will experiment with bilinear interpolation upsampling implemented by transposed convolution layers. Construct a transposed convolution layer that magnifies height and width of input by a factor of 2 and initialize its convolution kernel with the `bilinear_kernel` function.

```
In [11]: conv_trans = nn.Conv2DTranspose(3, kernel_size=4, padding=1, strides=2)
         conv_trans.initialize(init.Constant(bilinear_kernel(3, 3, 4)))
```

Read the image `X` and record the result of upsampling as `Y`. In order to print the image, we need to adjust the position of the channel dimension.

```
In [12]: img = image.imread('../img/catdog.jpg')
         X = img.astype('float32').transpose((2, 0, 1)).expand_dims(axis=0) / 255
         Y = conv_trans(X)
         out_img = Y[0].transpose((1, 2, 0))
```

As you can see, the transposed convolution layer magnifies both the height and width of the image by a factor of 2. It is worth mentioning that, besides to the difference in coordinate scale, the image magnified by bilinear interpolation and original image printed in the *Object Detection and Bounding Box* section look the same.

```
In [13]: d2l.set_figsize()
         print('input image shape:', img.shape)
         d2l.plt.imshow(img.asnumpy());
         print('output image shape:', out_img.shape)
         d2l.plt.imshow(out_img.asnumpy());

input image shape: (561, 728, 3)
output image shape: (1122, 1456, 3)
```
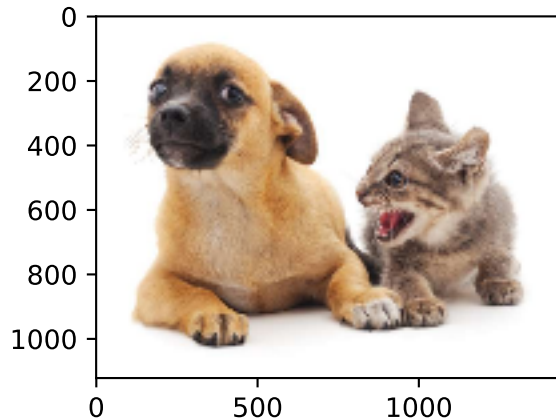

In a fully convolutional network, we initialize the transposed convolution layer for upsampled bilinear interpolation. For a 1×1 convolution layer, we use Xavier for randomly initialization.

```
In [14]: net[-1].initialize(init.Constant(bilinear_kernel(num_classes, num_classes,
                                                            64)))
         net[-2].initialize(init=init.Xavier())
```

10.10.4 Read the Data Set

We read the data set using the method described in the previous section. Here, we specify shape of the randomly cropped output image as 320×480, so both the height and width are divisible by 32.

```
In [15]: crop_size, batch_size, colormap2label = (320, 480), 32, nd.zeros(256**3)
         for i, cm in enumerate(d2l.VOC_COLORMAP):
             colormap2label[(cm[0] * 256 + cm[1]) * 256 + cm[2]] = i
         voc_dir = d2l.download_voc_pascal(data_dir='../data')

         num_workers = 0 if sys.platform.startswith('win32') else 4
         train_iter = gdata.DataLoader(
             d2l.VOCSegDataset(True, crop_size, voc_dir, colormap2label), batch_size,
             shuffle=True, last_batch='discard', num_workers=num_workers)
         test_iter = gdata.DataLoader(
             d2l.VOCSegDataset(False, crop_size, voc_dir, colormap2label), batch_size,
             last_batch='discard', num_workers=num_workers)

read 1114 examples
read 1078 examples
```

10.10.5 Training

Now we can start training the model. The loss function and accuracy calculation here are not substantially different from those used in image classification. Because we use the channel of the transposed convolution layer to predict pixel categories, the `axis=1` (channel dimension) option is specified in `SoftmaxCrossEntropyLoss`. In addition, the model calculates the accuracy based on whether the prediction category of each pixel is correct.

```
In [16]: ctx = d2l.try_all_gpus()
         loss = gloss.SoftmaxCrossEntropyLoss(axis=1)
         net.collect_params().reset_ctx(ctx)
         trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.1,
                                                               'wd': 1e-3})
         d2l.train(train_iter, test_iter, net, loss, trainer, ctx, num_epochs=5)

training on [gpu(0), gpu(1)]
epoch 1, loss 1.3434, train acc 0.732, test acc 0.789, time 27.8 sec
epoch 2, loss 0.5883, train acc 0.825, test acc 0.839, time 24.0 sec
epoch 3, loss 0.4579, train acc 0.855, test acc 0.843, time 23.9 sec
epoch 4, loss 0.4095, train acc 0.868, test acc 0.824, time 24.0 sec
epoch 5, loss 0.3396, train acc 0.888, test acc 0.843, time 23.9 sec
```

10.10.6 Prediction

During predicting, we need to standardize the input image in each channel and transform them into the four-dimensional input format required by the convolutional neural network.

```
In [17]: def predict(img):
             X = test_iter._dataset.normalize_image(img)
             X = X.transpose((2, 0, 1)).expand_dims(axis=0)
```

```
            pred = nd.argmax(net(X.as_in_context(ctx[0])), axis=1)
            return pred.reshape((pred.shape[1], pred.shape[2]))
```

To visualize the predicted categories for each pixel, we map the predicted categories back to their labeled colors in the data set.

```
In [18]: def label2image(pred):
             colormap = nd.array(d2l.VOC_COLORMAP, ctx=ctx[0], dtype='uint8')
             X = pred.astype('int32')
             return colormap[X, :]
```

The size and shape of the images in the test data set vary. Because the model uses a transposed convolution layer with a stride of 32, when the height or width of the input image is not divisible by 32, the height or width of the transposed convolution layer output deviates from the size of the input image. In order to solve this problem, we can crop multiple rectangular areas in the image with heights and widths as integer multiples of 32, and then perform forward computation on the pixels in these areas. When combined, these areas must completely cover the input image. When a pixel is covered by multiple areas, the average of the transposed convolution layer output in the forward computation of the different areas can be used as an input for the softmax operation to predict the category.

For the sake of simplicity, we only read a few large test images and crop an area with a shape of 320×480 from the top-left corner of the image. Only this area is used for prediction. For the input image, we print the cropped area first, then print the predicted result, and finally print the labeled category.

```
In [19]: test_images, test_labels = d2l.read_voc_images(is_train=False)
         n, imgs = 4, []
         for i in range(n):
             crop_rect = (0, 0, 480, 320)
             X = image.fixed_crop(test_images[i], *crop_rect)
             pred = label2image(predict(X))
             imgs += [X, pred, image.fixed_crop(test_labels[i], *crop_rect)]
         d2l.show_images(imgs[::3] + imgs[1::3] + imgs[2::3], 3, n);
```

10.10. Fully Convolutional Networks (FCN)   **499**
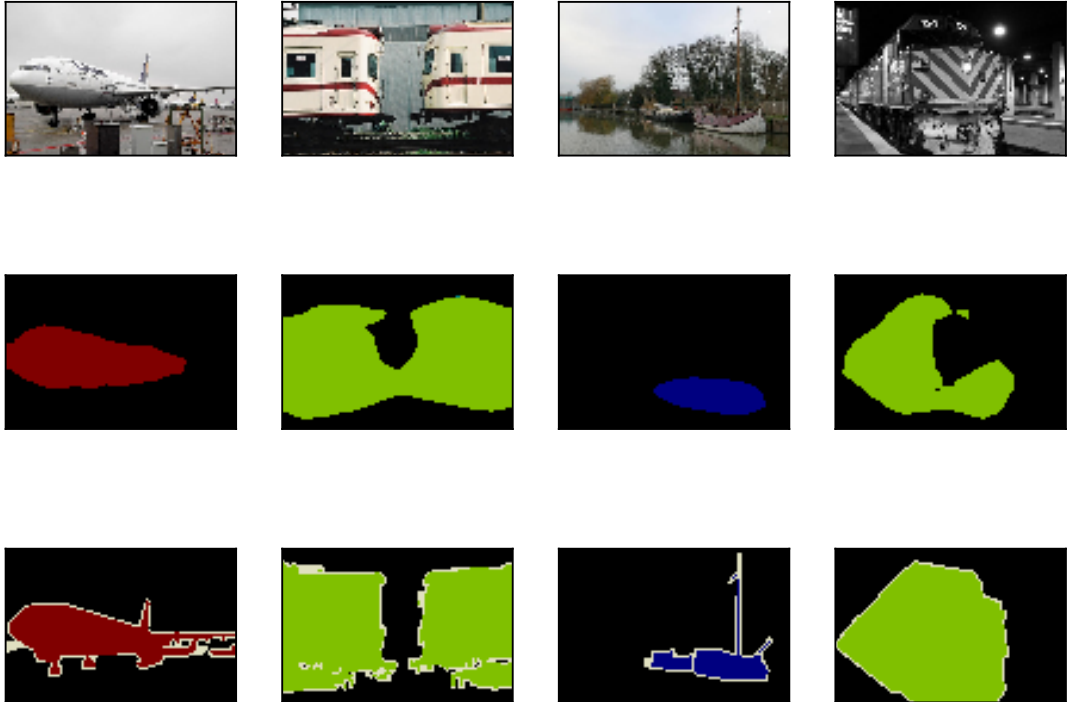
Summary

- We can implement convolution operations by matrix multiplication.

- The fully convolutional network first uses the convolutional neural network to extract image features, then transforms the number of channels into the number of categories through the 1×1 convolution layer, and finally transforms the height and width of the feature map to the size of the input image by using the transposed convolution layer to output the category of each pixel.

- In a fully convolutional network, we initialize the transposed convolution layer for upsampled bilinear interpolation.

Exercises

- Is it efficient to use matrix multiplication to implement convolution operations? Why?

- If we use Xavier to randomly initialize the transposed convolution layer, what will happen to the result?

- Can you further improve the accuracy of the model by tuning the hyper-parameters?

- Predict the categories of all pixels in the test image.

- The outputs of some intermediate layers of the convolutional neural network are also used in the paper on fully convolutional networks[1]. Try to implement this idea.

References

[1] Long, J., Shelhamer, E., & Darrell, T. (2015). Fully convolutional networks for semantic segmentation. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 3431-3440).

[2] Dumoulin, V., & Visin, F. (2016). A guide to convolution arithmetic for deep learning. arXiv preprint arXiv:1603.07285.

Scan the QR Code to Discuss

10.11 Neural Style Transfer

If you use social sharing apps or happen to be an amateur photographer, you are familiar with filters. Filters can alter the color styles of photos to make the background sharper or people's faces whiter. However, a filter generally can only change one aspect of a photo. To create the ideal photo, you often need to try many different filter combinations. This process is as complex as tuning the hyper-parameters of a model.

In this section, we will discuss how we can use convolution neural networks (CNNs) to automatically apply the style of one image to another image, an operation known as style transfer[1]. Here, we need two input images, one content image and one style image. We use a neural network to alter the content image so that its style mirrors that of the style image. In Figure 9.12, the content image is a landscape photo the author took in Mount Rainier National Part near Seattle. The style image is an oil painting of oak trees in autumn. The output composite image retains the overall shapes of the objects in the content image, but applies the oil painting brushwork of the style image and makes the overall color more vivid.
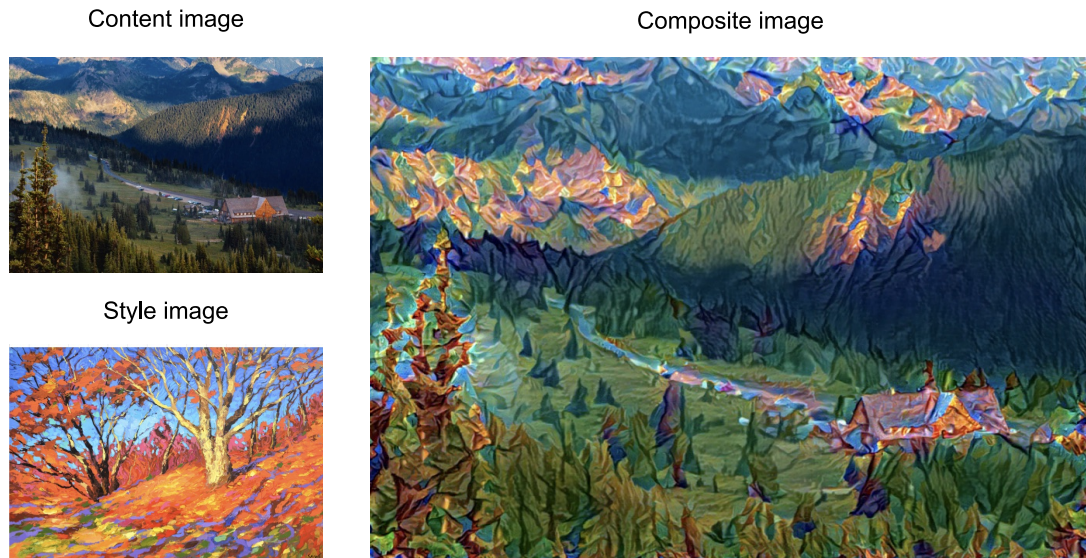
Content image   Composite image

Style image

Fig. 10.12: Content and style input images and composite image produced by style transfer.

10.11.1 Technique

Figure 9.13 shows an output of the CNN-based style transfer method. First, we initialize the composite image. For example, we can initialize it as the content image. This composite image is the only variable that needs to be updated in the style transfer process, i.e. the model parameter to be updated in style transfer. Then, we select a pre-trained CNN to extract image features. These model parameters do not need to be updated during training. The deep CNN uses multiple neural layers that successively extract image features. We can select the output of certain layers to use as content features or style features. If we use the structure in Figure 9.13, the pretrained neural network contains three convolutional layers. The second layer outputs the image content features, while the outputs of the first and third layers are used as style features. Next, we use forward propagation (in the direction of the solid lines) to compute the style transfer loss function and backward propagation (in the direction of the dotted lines) to update the model parameter, constantly updating the composite image. The loss functions used in style transfer generally have three parts: 1. Content loss is used to make the composite image approximate the content image as regards content features. 2. Style loss is used to make the composite image approximate the style image in terms of style features. 3. Total variation loss helps reduce the noise in the composite image. Finally, after we finish training the model, we output the style transfer model parameters to obtain the final composite image.
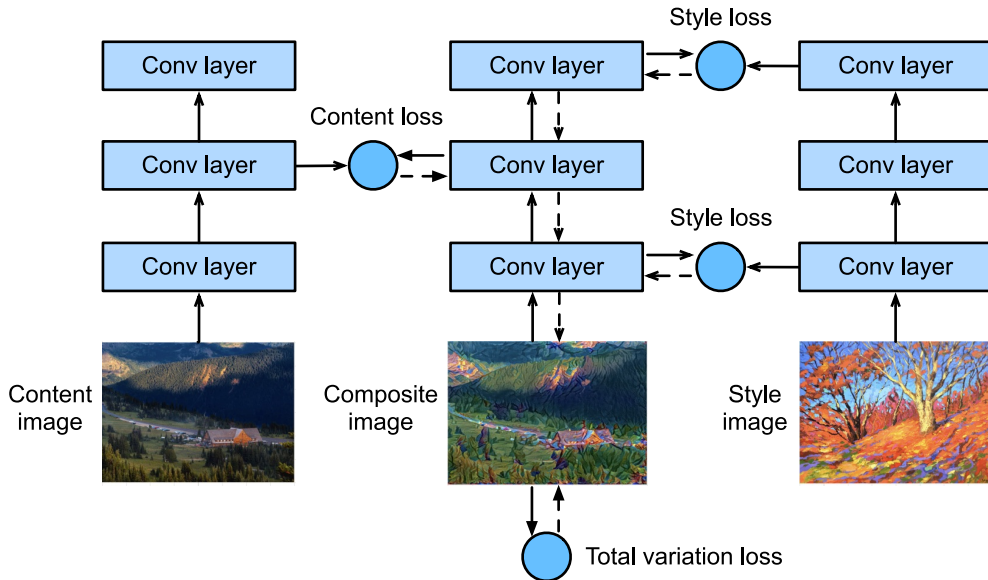
Fig. 10.13: CNN-based style transfer process. Solid lines show the direction of forward propagation and dotted lines show backward propagation.

Next, we will perform an experiment to help us better understand the technical details of style transfer.

10.11.2 Read the Content and Style Images

First, we read the content and style images. By printing out the image coordinate axes, we can see that they have different dimensions.

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
        import d2l
        from mxnet import autograd, gluon, image, init, nd
        from mxnet.gluon import model_zoo, nn
        import time

        d2l.set_figsize()
        content_img = image.imread('../img/rainier.jpg')
        d2l.plt.imshow(content_img.asnumpy());
```
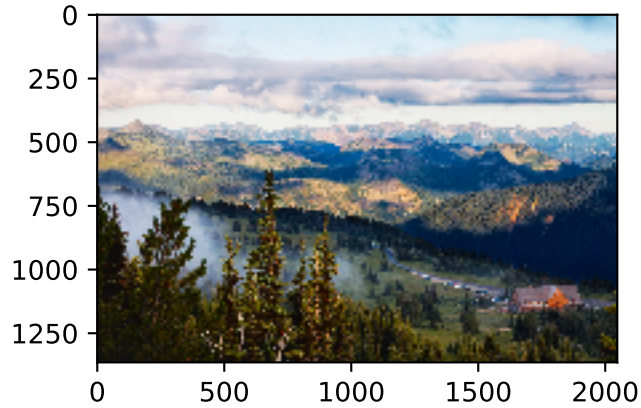
```
In [2]: style_img = image.imread('../img/autumn_oak.jpg')
        d2l.plt.imshow(style_img.asnumpy());
```
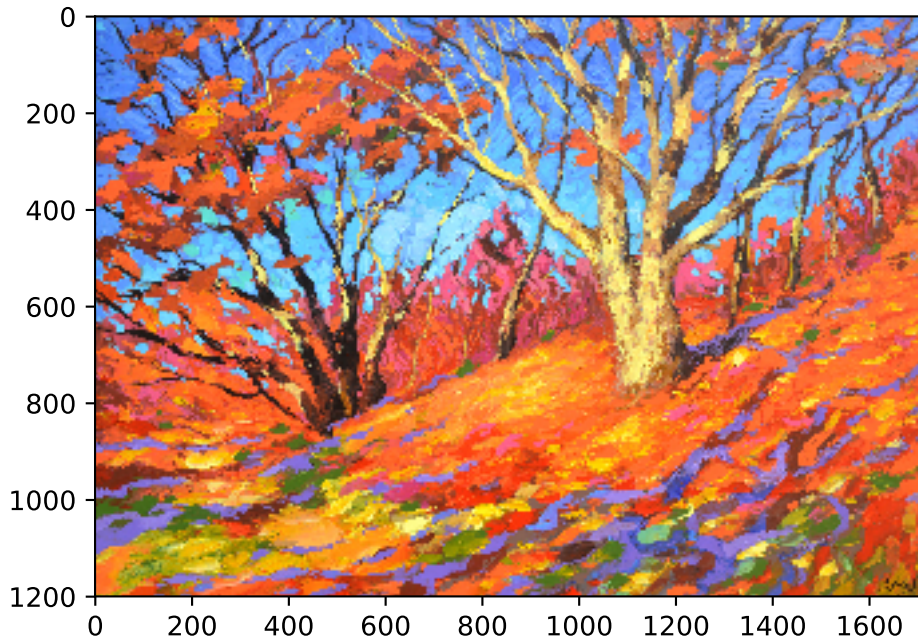

10.11.3 Preprocessing and Postprocessing

Below, we define the functions for image preprocessing and postprocessing. The `preprocess` function normalizes each of the three RGB channels of the input images and transforms the results to a format that can be input to the CNN. The `postprocess` function restores the pixel values in the output image to

their original values before normalization. Because the image printing function requires that each pixel has a floating point value from 0 to 1, we use the `clip` function to replace values smaller than 0 or greater than 1 with 0 or 1, respectively.

```
In [3]: rgb_mean = nd.array([0.485, 0.456, 0.406])
        rgb_std = nd.array([0.229, 0.224, 0.225])

        def preprocess(img, image_shape):
            img = image.imresize(img, *image_shape)
            img = (img.astype('float32') / 255 - rgb_mean) / rgb_std
            return img.transpose((2, 0, 1)).expand_dims(axis=0)

        def postprocess(img):
            img = img[0].as_in_context(rgb_std.context)
            return (img.transpose((1, 2, 0)) * rgb_std + rgb_mean).clip(0, 1)
```

10.11.4 Extract Features

We use the VGG-19 model pre-trained on the ImageNet data set to extract image features[1].

```
In [4]: pretrained_net = model_zoo.vision.vgg19(pretrained=True)
```

To extract image content and style features, we can select the outputs of certain layers in the VGG network. In general, the closer an output is to the input layer, the easier it is to extract image detail information. The farther away an output is, the easier it is to extract global information. To prevent the composite image from retaining too many details from the content image, we select a VGG network layer near the output layer to output the image content features. This layer is called the content layer. We also select the outputs of different layers from the VGG network for matching local and global styles. These are called the style layers. As we mentioned in the *Networks Using Duplicates (VGG)* section, VGG networks have five convolutional blocks. In this experiment, we select the last convolutional layer of the fourth convolutional block as the content layer and the first layer of each block as style layers. We can obtain the indexes for these layers by printing the `pretrained_net` instance.

```
In [5]: style_layers, content_layers = [0, 5, 10, 19, 28], [25]
```

During feature extraction, we only need to use all the VGG layers from the input layer to the content or style layer nearest the output layer. Below, we build a new network, `net`, which only retains the layers in the VGG network we need to use. We then use `net` to extract features.

```
In [6]: net = nn.Sequential()
        for i in range(max(content_layers + style_layers) + 1):
            net.add(pretrained_net.features[i])
```

Given input `X`, if we simply call the forward computation `net(X)`, we can only obtain the output of the last layer. Because we also need the outputs of the intermediate layers, we need to perform layer-by-layer computation and retain the content and style layer outputs.

```
In [7]: def extract_features(X, content_layers, style_layers):
            contents = []
            styles = []
            for i in range(len(net)):
                X = net[i](X)
```

```
            if i in style_layers:
                styles.append(X)
            if i in content_layers:
                contents.append(X)
        return contents, styles
```

Next, we define two functions: The `get_contents` function obtains the content features extracted from the content image, while the `get_styles` function obtains the style features extracted from the style image. Because we do not need to change the parameters of the pre-trained VGG model during training, we can extract the content features from the content image and style features from the style image before the start of training. As the composite image is the model parameter that must be updated during style transfer, we can only call the `extract_features` function during training to extract the content and style features of the composite image.

```
In [8]: def get_contents(image_shape, ctx):
            content_X = preprocess(content_img, image_shape).copyto(ctx)
            contents_Y, _ = extract_features(content_X, content_layers, style_layers)
            return content_X, contents_Y

        def get_styles(image_shape, ctx):
            style_X = preprocess(style_img, image_shape).copyto(ctx)
            _, styles_Y = extract_features(style_X, content_layers, style_layers)
            return style_X, styles_Y
```

10.11.5 Define the Loss Function

Next, we will look at the loss function used for style transfer. The loss function includes the content loss, style loss, and total variation loss.

Content Loss

Similar to the loss function used in linear regression, content loss uses a square error function to measure the difference in content features between the composite image and content image. The two inputs of the square error function are both content layer outputs obtained from the `extract_features` function.

```
In [9]: def content_loss(Y_hat, Y):
            return (Y_hat - Y).square().mean()
```

Style Loss

Style loss, similar to content loss, uses a square error function to measure the difference in style between the composite image and style image. To express the styles output by the style layers, we first use the `extract_features` function to compute the style layer output. Assuming that the output has 1 example, c channels, and a height and width of h and w, we can transform the output into the matrix $\boldsymbol{X}$, which has c rows and $h \cdot w$ columns. You can think of matrix $\boldsymbol{X}$ as the combination of the c vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_c$, which have a length of hw. Here, the vector $\boldsymbol{x}_i$ represents the style feature of channel i. In the Gram matrix of these vectors $\boldsymbol{X}\boldsymbol{X}^\top \in \mathbb{R}^{c \times c}$, element x_{ij} in row i column j is the inner product of

vectors x_i and x_j. It represents the correlation of the style features of channels i and j. We use this type of Gram matrix to represent the style output by the style layers. You must note that, when the $h \cdot w$ value is large, this often leads to large values in the Gram matrix. In addition, the height and width of the Gram matrix are both the number of channels c. To ensure that the style loss is not affected by the size of these values, we define the `gram` function below to divide the Gram matrix by the number of its elements, i.e. $c \cdot h \cdot w$.

```
In [10]: def gram(X):
             num_channels, n = X.shape[1], X.size // X.shape[1]
             X = X.reshape((num_channels, n))
             return nd.dot(X, X.T) / (num_channels * n)
```

Naturally, the two Gram matrix inputs of the square error function for style loss are taken from the composite image and style image style layer outputs. Here, we assume that the Gram matrix of the style image, `gram_Y`, has been computed in advance.

```
In [11]: def style_loss(Y_hat, gram_Y):
             return (gram(Y_hat) - gram_Y).square().mean()
```

Total Variance Loss

Sometimes, the composite images we learn have a lot of high-frequency noise, particularly bright or dark pixels. One common noise reduction method is total variation denoising. We assume that $x_{i,j}$ represents the pixel value at the coordinate (i, j), so the total variance loss is:

$$\sum_{i,j} |x_{i,j} - x_{i+1,j}| + |x_{i,j} - x_{i,j+1}|$$

We try to make the values of neighboring pixels as similar as possible.

```
In [12]: def tv_loss(Y_hat):
             return 0.5 * ((Y_hat[:, :, 1:, :] - Y_hat[:, :, :-1, :]).abs().mean() +
                           (Y_hat[:, :, :, 1:] - Y_hat[:, :, :, :-1]).abs().mean())
```

Loss Function

The loss function for style transfer is the weighted sum of the content loss, style loss, and total variance loss. By adjusting these weight hyper-parameters, we can balance the retained content, transferred style, and noise reduction in the composite image according to their relative importance.

```
In [13]: content_weight, style_weight, tv_weight = 1, 1e3, 10

         def compute_loss(X, contents_Y_hat, styles_Y_hat, contents_Y, styles_Y_gram):
             # Calculate the content, style, and total variance losses respectively
             contents_l = [content_loss(Y_hat, Y) * content_weight for Y_hat, Y in zip(
                 contents_Y_hat, contents_Y)]
             styles_l = [style_loss(Y_hat, Y) * style_weight for Y_hat, Y in zip(
                 styles_Y_hat, styles_Y_gram)]
             tv_l = tv_loss(X) * tv_weight
             # Add up all the losses
```

```
        l = nd.add_n(*styles_l) + nd.add_n(*contents_l) + tv_l
        return contents_l, styles_l, tv_l, l
```

10.11.6 Create and Initialize the Composite Image

In style transfer, the composite image is the only variable that needs to be updated. Therefore, we can define a simple model, `GeneratedImage`, and treat the composite image as a model parameter. In the model, forward computation only returns the model parameter.

```
In [14]: class GeneratedImage(nn.Block):
             def __init__(self, img_shape, **kwargs):
                 super(GeneratedImage, self).__init__(**kwargs)
                 self.weight = self.params.get('weight', shape=img_shape)

             def forward(self):
                 return self.weight.data()
```

Next, we define the `get_inits` function. This function creates a composite image model instance and initializes it to the image X. The Gram matrix for the various style layers of the style image, `styles_Y_gram`, is computed prior to training.

```
In [15]: def get_inits(X, ctx, lr, styles_Y):
             gen_img = GeneratedImage(X.shape)
             gen_img.initialize(init.Constant(X), ctx=ctx, force_reinit=True)
             trainer = gluon.Trainer(gen_img.collect_params(), 'adam',
                                     {'learning_rate': lr})
             styles_Y_gram = [gram(Y) for Y in styles_Y]
             return gen_img(), styles_Y_gram, trainer
```

10.11.7 Training

During model training, we constantly extract the content and style features of the composite image and calculate the loss function. Recall our discussion of how synchronization functions force the front end to wait for computation results in the *Asynchronous Computation* section. Because we only call the `asscalar` synchronization function every 50 epochs, the process may occupy a great deal of memory. Therefore, we call the `waitall` synchronization function during every epoch.

```
In [16]: def train(X, contents_Y, styles_Y, ctx, lr, max_epochs, lr_decay_epoch):
             X, styles_Y_gram, trainer = get_inits(X, ctx, lr, styles_Y)
             for i in range(max_epochs):
                 start = time.time()
                 with autograd.record():
                     contents_Y_hat, styles_Y_hat = extract_features(
                         X, content_layers, style_layers)
                     contents_l, styles_l, tv_l, l = compute_loss(
                         X, contents_Y_hat, styles_Y_hat, contents_Y, styles_Y_gram)
                 l.backward()
                 trainer.step(1)
                 nd.waitall()
                 if i % 50 == 0 and i != 0:
                     print('epoch %3d, content loss %.2f, style loss %.2f, '
```

```
                      'TV loss %.2f, %.2f sec'
                      % (i, nd.add_n(*contents_l).asscalar(),
                          nd.add_n(*styles_l).asscalar(), tv_l.asscalar(),
                          time.time() - start))
              if i % lr_decay_epoch == 0 and i != 0:
                  trainer.set_learning_rate(trainer.learning_rate * 0.1)
                  print('change lr to %.1e' % trainer.learning_rate)
          return X
```

Next, we start to train the model. First, we set the height and width of the content and style images to 150 by 225 pixels. We use the content image to initialize the composite image.

```
In [17]: ctx, image_shape = d2l.try_gpu(), (225, 150)
         net.collect_params().reset_ctx(ctx)
         content_X, contents_Y = get_contents(image_shape, ctx)
         _, styles_Y = get_styles(image_shape, ctx)
         output = train(content_X, contents_Y, styles_Y, ctx, 0.01, 500, 200)

epoch  50, content loss 10.10, style loss 29.39, TV loss 3.46, 0.03 sec
epoch 100, content loss 7.51, style loss 15.44, TV loss 3.89, 0.03 sec
epoch 150, content loss 6.31, style loss 10.37, TV loss 4.15, 0.03 sec
epoch 200, content loss 5.63, style loss 8.12, TV loss 4.29, 0.03 sec
change lr to 1.0e-03
epoch 250, content loss 5.56, style loss 7.94, TV loss 4.30, 0.03 sec
epoch 300, content loss 5.50, style loss 7.80, TV loss 4.31, 0.03 sec
epoch 350, content loss 5.45, style loss 7.65, TV loss 4.31, 0.03 sec
epoch 400, content loss 5.39, style loss 7.50, TV loss 4.32, 0.03 sec
change lr to 1.0e-04
epoch 450, content loss 5.39, style loss 7.48, TV loss 4.32, 0.03 sec
```

Next, we save the trained composite image. As you can see, the composite image in Figure 9.14 retains the scenery and objects of the content image, while introducing the color of the style image. Because the image is relatively small, the details are a bit fuzzy.

```
In [18]: d2l.plt.imsave('../img/neural-style-1.png', postprocess(output).asnumpy())
```


Fig. 10.14: 150×225 composite image.

To obtain a clearer composite image, we train the model using a larger image size: 300×450. We increase the height and width of the image in Figure 9.14 by a factor of two and initialize a larger composite image.

```
In [19]: image_shape = (450, 300)
         _, content_Y = get_contents(image_shape, ctx)
         _, style_Y = get_styles(image_shape, ctx)
         X = preprocess(postprocess(output) * 255, image_shape)
         output = train(X, content_Y, style_Y, ctx, 0.01, 300, 100)
         d2l.plt.imsave('../img/neural-style-2.png', postprocess(output).asnumpy())

epoch  50, content loss 13.84, style loss 13.73, TV loss 2.38, 0.10 sec
epoch 100, content loss 9.49, style loss 8.77, TV loss 2.65, 0.10 sec
change lr to 1.0e-03
epoch 150, content loss 9.19, style loss 8.45, TV loss 2.68, 0.10 sec
epoch 200, content loss 8.93, style loss 8.16, TV loss 2.70, 0.10 sec
change lr to 1.0e-04
epoch 250, content loss 8.89, style loss 8.13, TV loss 2.70, 0.10 sec
```

As you can see, each epoch takes more time due to the larger image size. As shown in Figure 9.15, the composite image produced retains more detail due to its larger size. The composite image not only has large blocks of color like the style image, but these blocks even have the subtle texture of brush strokes.
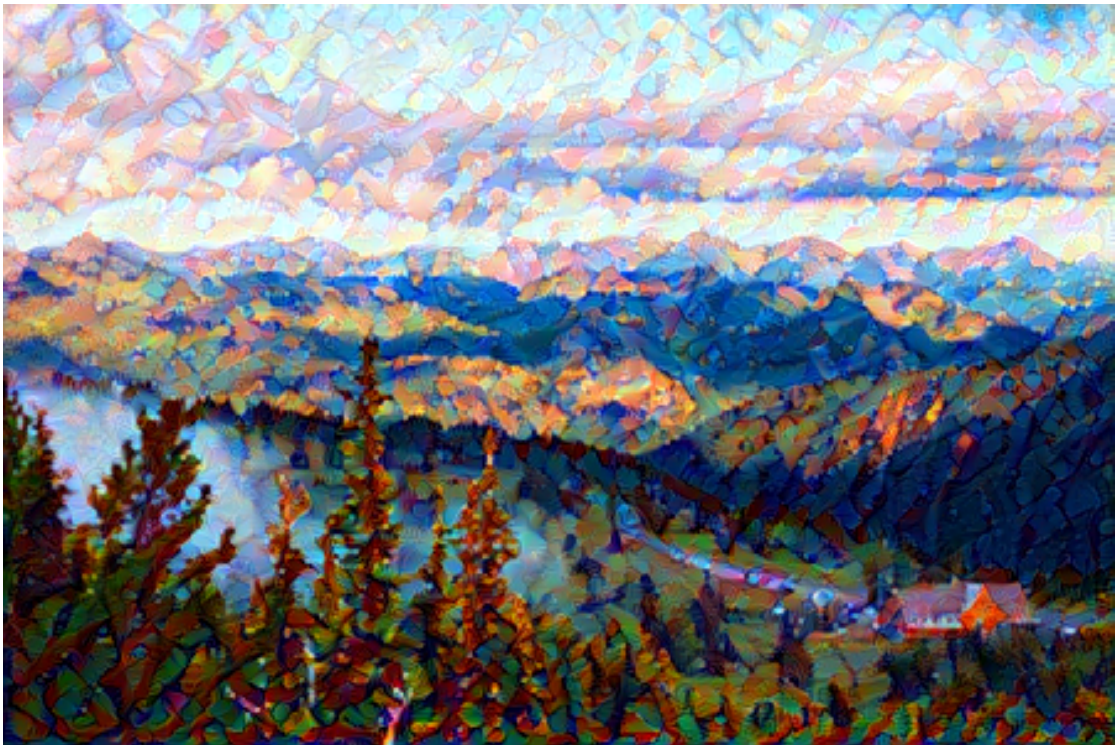

Fig. 10.15: 300×450 composite image.

Summary

- The loss functions used in style transfer generally have three parts: 1. Content loss is used to make the composite image approximate the content image as regards content features. 2. Style loss is used to make the composite image approximate the style image in terms of style features. 3. Total variation loss helps reduce the noise in the composite image.

- We can use a pre-trained CNN to extract image features and minimize the loss function to continuously update the composite image.

- We use a Gram matrix to represent the style output by the style layers.

Exercises

- How does the output change when you select different content and style layers?

- Adjust the weight hyper-parameters in the loss function. Does the output retain more content or have less noise?

- Use different content and style images. Can you create more interesting composite images?

Reference

[1] Gatys, L. A., Ecker, A. S., & Bethge, M. (2016). Image style transfer using convolutional neural networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 2414-2423).

Scan the QR Code to Discuss

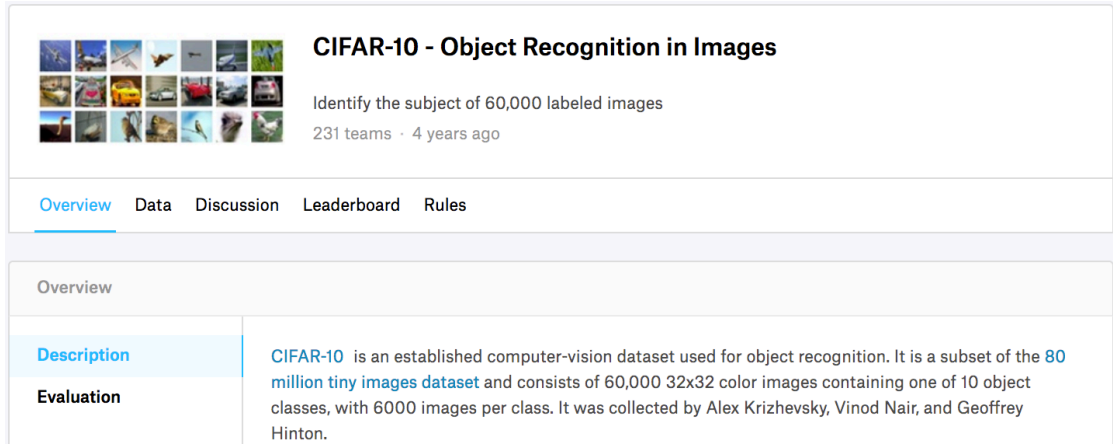# 10.12 Image Classification (CIFAR-10) on Kaggle

So far, we have been using Gluon's `data` package to directly obtain image data sets in NDArray format. In practice, however, image data sets often exist in the format of image files. In this section, we will start with the original image files and organize, read, and convert the files to NDArray format step by step.

We performed an experiment on the CIFAR-10 data set in the *Image Augmentation* section. This is an important data set in the computer vision field. Now, we will apply the knowledge we learned in the previous sections in order to participate in the Kaggle competition, which addresses CIFAR-10 image classification problems. The competition's web address is

> https://www.kaggle.com/c/cifar-10

Figure 9.16 shows the information on the competition's webpage. In order to submit the results, please register an account on the Kaggle website first.

Fig. 10.16: CIFAR-10 image classification competition webpage information. The data set for the competition can be accessed by clicking the Data tab. (Source: www.

First, import the packages or modules required for the competition.

```
In [1]: import sys
        sys.path.insert(0, '..')

        import d2l
        from mxnet import autograd, gluon, init
        from mxnet.gluon import data as gdata, loss as gloss, nn
        import os
        import pandas as pd
        import shutil
        import time
```

10.12.1 Obtain and Organize the Data Sets

The competition data is divided into a training set and testing set. The training set contains 50,000 images. The testing set contains 300,000 images, of which 10,000 images are used for scoring, while the other 290,000 non-scoring images are included to prevent the manual labeling of the testing set and the submission of labeling results. The image formats in both data sets are PNG, with heights and widths of

32 pixels and three color channels (RGB). The images cover 10 categories: planes, cars, birds, cats, deer, dogs, frogs, horses, boats, and trucks. The upper-left corner of Figure 9.16 shows some images of planes, cars, and birds in the data set.

Download the Data Set

After logging in to Kaggle, we can click on the Data tab on the CIFAR-10 image classification competition webpage shown in Figure 9.16 and download the training data set train.7z, the testing data set test.7z, and the training data set labels trainlabels.csv.

Unzip the Data Set

The training data set train.7z and the test data set test.7z need to be unzipped after downloading. After unzipping the data sets, store the training data set, test data set, and training data set labels in the following respective paths:

- ../data/kaggle_cifar10/train/[1-50000].png

- ../data/kaggle_cifar10/test/[1-300000].png

- ../data/kaggle_cifar10/trainLabels.csv

To make it easier to get started, we provide a small-scale sample of the data set mentioned above. train_tiny.zip contains 100 training examples, while test_tiny.zip contains only one test example. Their unzipped folder names are train_tiny and test_tiny, respectively. In addition, unzip the zip file of the training data set labels to obtain the file trainlabels.csv. If you are going to use the full data set of the Kaggle competition, you will also need to change the following `demo` variable to `False`.

```
In [2]: # If you use the full data set downloaded for the Kaggle competition, change
        # the demo variable to False
        demo = True
        if demo:
            import zipfile
            for f in ['train_tiny.zip', 'test_tiny.zip', 'trainLabels.csv.zip']:
                with zipfile.ZipFile('../data/kaggle_cifar10/' + f, 'r') as z:
                    z.extractall('../data/kaggle_cifar10/')
```

Organize the Data Set

We need to organize data sets to facilitate model training and testing. The following `read_label_file` function will be used to read the label file for the training data set. The parameter `valid_ratio` in this function is the ratio of the number of examples in the validation set to the number of examples in the original training set.

```
In [3]: def read_label_file(data_dir, label_file, train_dir, valid_ratio):
            with open(os.path.join(data_dir, label_file), 'r') as f:
                # Skip the file header line (column name)
                lines = f.readlines()[1:]
```

```
            tokens = [l.rstrip().split(',') for l in lines]
            idx_label = dict(((int(idx), label) for idx, label in tokens))
    labels = set(idx_label.values())
    n_train_valid = len(os.listdir(os.path.join(data_dir, train_dir)))
    n_train = int(n_train_valid * (1 - valid_ratio))
    assert 0 < n_train < n_train_valid
    return n_train // len(labels), idx_label
```

Below we define a helper function to create a path only if the path does not already exist.

```
In [4]:   # This function has been saved in the d2l package for future use
      def mkdir_if_not_exist(path):
          if not os.path.exists(os.path.join(*path)):
              os.makedirs(os.path.join(*path))
```

Next, we define the reorg_train_valid function to segment the validation set from the original training set. Here, we use valid_ratio=0.1 as an example. Since the original training set has 50,000 images, there will be 45,000 images used for training and stored in the path input_dir/train when tuning hyper-parameters, while the other 5,000 images will be stored as validation set in the path input_dir/valid. After organizing the data, images of the same type will be placed under the same folder so that we can read them later.

```
In [5]: def reorg_train_valid(data_dir, train_dir, input_dir, n_train_per_label,
                               idx_label):
            label_count = {}
            for train_file in os.listdir(os.path.join(data_dir, train_dir)):
                idx = int(train_file.split('.')[0])
                label = idx_label[idx]
                mkdir_if_not_exist([data_dir, input_dir, 'train_valid', label])
                shutil.copy(os.path.join(data_dir, train_dir, train_file),
                            os.path.join(data_dir, input_dir, 'train_valid', label))
                if label not in label_count or label_count[label] < n_train_per_label:
                    mkdir_if_not_exist([data_dir, input_dir, 'train', label])
                    shutil.copy(os.path.join(data_dir, train_dir, train_file),
                                os.path.join(data_dir, input_dir, 'train', label))
                    label_count[label] = label_count.get(label, 0) + 1
                else:
                    mkdir_if_not_exist([data_dir, input_dir, 'valid', label])
                    shutil.copy(os.path.join(data_dir, train_dir, train_file),
                                os.path.join(data_dir, input_dir, 'valid', label))
```

The reorg_test function below is used to organize the testing set to facilitate the reading during prediction.

```
In [6]: def reorg_test(data_dir, test_dir, input_dir):
            mkdir_if_not_exist([data_dir, input_dir, 'test', 'unknown'])
            for test_file in os.listdir(os.path.join(data_dir, test_dir)):
                shutil.copy(os.path.join(data_dir, test_dir, test_file),
                            os.path.join(data_dir, input_dir, 'test', 'unknown'))
```

Finally, we use a function to call the previously defined reorg_test, reorg_train_valid, and reorg_test functions.

```
In [7]: def reorg_cifar10_data(data_dir, label_file, train_dir, test_dir, input_dir,
                                valid_ratio):
            n_train_per_label, idx_label = read_label_file(data_dir, label_file,
```

```
                                                       train_dir, valid_ratio)
        reorg_train_valid(data_dir, train_dir, input_dir, n_train_per_label,
                          idx_label)
        reorg_test(data_dir, test_dir, input_dir)
```

We use only 100 training example and one test example here. The folder names for the training and testing data sets are train_tiny and test_tiny, respectively. Accordingly, we only set the batch size to 1. During actual training and testing, the complete data set of the Kaggle competition should be used and `batch_size` should be set to a larger integer, such as 128. We use 10% of the training examples as the validation set for tuning hyper-parameters.

```
In [8]: if demo:
            # Note: Here, we use small training sets and small testing sets and the
            # batch size should be set smaller. When using the complete data set for
            # the Kaggle competition, the batch size can be set to a large integer
            train_dir, test_dir, batch_size = 'train_tiny', 'test_tiny', 1
        else:
            train_dir, test_dir, batch_size = 'train', 'test', 128
        data_dir, label_file = '../data/kaggle_cifar10', 'trainLabels.csv'
        input_dir, valid_ratio = 'train_valid_test', 0.1
        reorg_cifar10_data(data_dir, label_file, train_dir, test_dir, input_dir,
                           valid_ratio)
```

10.12.2 Image Augmentation

To cope with overfitting, we use image augmentation. For example, by adding `transforms.RandomFlipLeftRight()`, the images can be flipped at random. We can also perform normalization for the three RGB channels of color images using `transforms.Normalize()`. Below, we list some of these operations that you can choose to use or modify depending on requirements.

```
In [9]: transform_train = gdata.vision.transforms.Compose([
            # Magnify the image to a square of 40 pixels in both height and width
            gdata.vision.transforms.Resize(40),
            # Randomly crop a square image of 40 pixels in both height and width to
            # produce a small square of 0.64 to 1 times the area of the original
            # image, and then shrink it to a square of 32 pixels in both height and
            # width
            gdata.vision.transforms.RandomResizedCrop(32, scale=(0.64, 1.0),
                                                      ratio=(1.0, 1.0)),
            gdata.vision.transforms.RandomFlipLeftRight(),
            gdata.vision.transforms.ToTensor(),
            # Normalize each channel of the image
            gdata.vision.transforms.Normalize([0.4914, 0.4822, 0.4465],
                                              [0.2023, 0.1994, 0.2010])])
```

In order to ensure the certainty of the output during testing, we only perform normalization on the image.

```
In [10]: transform_test = gdata.vision.transforms.Compose([
             gdata.vision.transforms.ToTensor(),
             gdata.vision.transforms.Normalize([0.4914, 0.4822, 0.4465],
                                               [0.2023, 0.1994, 0.2010])])
```

10.12.3 Read the Data Set

Next, we can create the `ImageFolderDataset` instance to read the organized data set containing the original image files, where each data instance includes the image and label.

```
In [11]: # Read the original image file. Flag=1 indicates that the input image has
         # three channels (color)
         train_ds = gdata.vision.ImageFolderDataset(
             os.path.join(data_dir, input_dir, 'train'), flag=1)
         valid_ds = gdata.vision.ImageFolderDataset(
             os.path.join(data_dir, input_dir, 'valid'), flag=1)
         train_valid_ds = gdata.vision.ImageFolderDataset(
             os.path.join(data_dir, input_dir, 'train_valid'), flag=1)
         test_ds = gdata.vision.ImageFolderDataset(
             os.path.join(data_dir, input_dir, 'test'), flag=1)
```

We specify the defined image augmentation operation in `DataLoader`. During training, we only use the validation set to evaluate the model, so we need to ensure the certainty of the output. During prediction, we will train the model on the combined training set and validation set to make full use of all labelled data.

```
In [12]: train_iter = gdata.DataLoader(train_ds.transform_first(transform_train),
                                       batch_size, shuffle=True, last_batch='keep')
         valid_iter = gdata.DataLoader(valid_ds.transform_first(transform_test),
                                       batch_size, shuffle=True, last_batch='keep')
         train_valid_iter = gdata.DataLoader(train_valid_ds.transform_first(
             transform_train), batch_size, shuffle=True, last_batch='keep')
         test_iter = gdata.DataLoader(test_ds.transform_first(transform_test),
                                      batch_size, shuffle=False, last_batch='keep')
```

10.12.4 Define the Model

Here, we build the residual blocks based on the HybridBlock class, which is slightly different than the implementation described in the *Residual networks (ResNet)* section. This is done to improve execution efficiency.

```
In [13]: class Residual(nn.HybridBlock):
             def __init__(self, num_channels, use_1x1conv=False, strides=1, **kwargs):
                 super(Residual, self).__init__(**kwargs)
                 self.conv1 = nn.Conv2D(num_channels, kernel_size=3, padding=1,
                                        strides=strides)
                 self.conv2 = nn.Conv2D(num_channels, kernel_size=3, padding=1)
                 if use_1x1conv:
                     self.conv3 = nn.Conv2D(num_channels, kernel_size=1,
                                            strides=strides)
                 else:
                     self.conv3 = None
                 self.bn1 = nn.BatchNorm()
                 self.bn2 = nn.BatchNorm()

             def hybrid_forward(self, F, X):
                 Y = F.relu(self.bn1(self.conv1(X)))
                 Y = self.bn2(self.conv2(Y))
```

```
            if self.conv3:
                X = self.conv3(X)
            return F.relu(Y + X)
```

Next, we define the ResNet-18 model.

```
In [14]: def resnet18(num_classes):
             net = nn.HybridSequential()
             net.add(nn.Conv2D(64, kernel_size=3, strides=1, padding=1),
                     nn.BatchNorm(), nn.Activation('relu'))

             def resnet_block(num_channels, num_residuals, first_block=False):
                 blk = nn.HybridSequential()
                 for i in range(num_residuals):
                     if i == 0 and not first_block:
                         blk.add(Residual(num_channels, use_1x1conv=True, strides=2))
                     else:
                         blk.add(Residual(num_channels))
                 return blk

             net.add(resnet_block(64, 2, first_block=True),
                     resnet_block(128, 2),
                     resnet_block(256, 2),
                     resnet_block(512, 2))
             net.add(nn.GlobalAvgPool2D(), nn.Dense(num_classes))
             return net
```

The CIFAR-10 image classification challenge uses 10 categories. We will perform Xavier random initialization on the model before training begins.

```
In [15]: def get_net(ctx):
             num_classes = 10
             net = resnet18(num_classes)
             net.initialize(ctx=ctx, init=init.Xavier())
             return net

         loss = gloss.SoftmaxCrossEntropyLoss()
```

10.12.5 Define the Training Functions

We will select the model and tune hyper-parameters according to the model's performance on the validation set. Next, we define the model training function train. We record the training time of each epoch, which helps us compare the time costs of different models.

```
In [16]: def train(net, train_iter, valid_iter, num_epochs, lr, wd, ctx, lr_period,
                   lr_decay):
             trainer = gluon.Trainer(net.collect_params(), 'sgd',
                                     {'learning_rate': lr, 'momentum': 0.9, 'wd': wd})
             for epoch in range(num_epochs):
                 train_l_sum, train_acc_sum, n, start = 0.0, 0.0, 0, time.time()
                 if epoch > 0 and epoch % lr_period == 0:
                     trainer.set_learning_rate(trainer.learning_rate * lr_decay)
                 for X, y in train_iter:
                     y = y.astype('float32').as_in_context(ctx)
                     with autograd.record():
```

```
                y_hat = net(X.as_in_context(ctx))
                l = loss(y_hat, y).sum()
            l.backward()
            trainer.step(batch_size)
            train_l_sum += l.asscalar()
            train_acc_sum += (y_hat.argmax(axis=1) == y).sum().asscalar()
            n += y.size
        time_s = "time %.2f sec" % (time.time() - start)
        if valid_iter is not None:
            valid_acc = d2l.evaluate_accuracy(valid_iter, net, ctx)
            epoch_s = ("epoch %d, loss %f, train acc %f, valid acc %f, "
                       % (epoch + 1, train_l_sum / n, train_acc_sum / n,
                          valid_acc))
        else:
            epoch_s = ("epoch %d, loss %f, train acc %f, " %
                       (epoch + 1, train_l_sum / n, train_acc_sum / n))
        print(epoch_s + time_s + ', lr ' + str(trainer.learning_rate))
```

10.12.6 Train and Validate the Model

Now, we can train and validate the model. The following hyper-parameters can be tuned. For example, we can increase the number of epochs. Because `lr_period` and `lr_decay` are set to 80 and 0.1 respectively, the learning rate of the optimization algorithm will be multiplied by 0.1 after every 80 epochs. For simplicity, we only train one epoch here.

```
In [17]: ctx, num_epochs, lr, wd = d2l.try_gpu(), 1, 0.1, 5e-4
         lr_period, lr_decay, net = 80, 0.1, get_net(ctx)
         net.hybridize()
         train(net, train_iter, valid_iter, num_epochs, lr, wd, ctx, lr_period,
               lr_decay)

epoch 1, loss 6.958322, train acc 0.122222, valid acc 0.100000, time 2.14 sec, lr 0.1
```

10.12.7 Classify the Testing Set and Submit Results on Kaggle

After obtaining a satisfactory model design and hyper-parameters, we use all training data sets (including validation sets) to retrain the model and classify the testing set.

```
In [18]: net, preds = get_net(ctx), []
         net.hybridize()
         train(net, train_valid_iter, None, num_epochs, lr, wd, ctx, lr_period,
               lr_decay)

         for X, _ in test_iter:
             y_hat = net(X.as_in_context(ctx))
             preds.extend(y_hat.argmax(axis=1).astype(int).asnumpy())
         sorted_ids = list(range(1, len(test_ds) + 1))
         sorted_ids.sort(key=lambda x: str(x))
         df = pd.DataFrame({'id': sorted_ids, 'label': preds})
         df['label'] = df['label'].apply(lambda x: train_valid_ds.synsets[x])
         df.to_csv('submission.csv', index=False)
```

```
epoch 1, loss 7.463331, train acc 0.090000, time 1.61 sec, lr 0.1
```

After executing the above code, we will get a submission.csv file. The format of this file is consistent with the Kaggle competition requirements. The method for submitting results is similar to method in the Get Started with Kaggle Competition: Predicting House Prices section.

Summary

- We can create an `ImageFolderDataset` instance to read the data set containing the original image files.
- We can use convolutional neural networks, image augmentation, and hybrid programming to take part in an image classification competition.

Exercises

- Use the complete CIFAF-10 data set for the Kaggle competition. Change the `batch_size` and number of epochs `num_epochs` to 128 and 100, respectively. See what accuracy and ranking you can achieve in this competition.
- What accuracy can you achieve when not using image augmentation?
- Scan the QR code to access the relevant discussions and exchange ideas about the methods used and the results obtained with the community. Can you come up with any better techniques?

Scan the QR Code to Discuss

10.13 Dog Breed Identification (ImageNet Dogs) on Kaggle

In this section, we will tackle the dog breed identification challenge in the Kaggle Competition. The competition's web address is

> https://www.kaggle.com/c/dog-breed-identification

In this competition, we attempt to identify 120 different breeds of dogs. The data set used in this competition is actually a subset of the famous ImageNet data set. Different from the images in the CIFAR-10

data set used in the previous section, the images in the ImageNet data set are higher and wider and their dimensions are inconsistent.

Figure 9.17 shows the information on the competition's webpage. In order to submit the results, please register an account on the Kaggle website first.

Fig. 10.17: Dog breed identification competition website. The data set for the competition can be accessed by clicking the Data tab. (Source: www.

First, import the packages or modules required for the competition.

```
In [1]: import sys
        sys.path.insert(0, '..')

        import collections
        import d2l
```

```
import math
from mxnet import autograd, gluon, init, nd
from mxnet.gluon import data as gdata, loss as gloss, model_zoo, nn
import os
import shutil
import time
import zipfile
```

10.13.1 Obtain and Organize the Data Sets

The competition data is divided into a training set and testing set. The training set contains 10,222 images and the testing set contains 10,357 images. The images in both sets are in JPEG format. These images contain three RGB channels (color) and they have different heights and widths. There are 120 breeds of dogs in the training set, including Labradors, Poodles, Dachshunds, Samoyeds, Huskies, Chihuahuas, and Yorkshire Terriers.

Download the Data Set

After logging in to Kaggle, we can click on the Data tab on the dog breed identification competition webpage shown in Figure 9.17 and download the training data set train.zip, the testing data set test.zip, and the training data set labels label.csv.zip. After downloading the files, place them in the three paths below:

- ../data/kaggle_dog/train.zip

- ../data/kaggle_dog/test.zip

- ../data/kaggle_dog/labels.csv.zip

To make it easier to get started, we provide a small-scale sample of the data set mentioned above, train_valid_test_tiny.zip. If you are going to use the full data set for the Kaggle competition, you will also need to change the demo variable below to False.

```
In [2]: # If you use the full data set downloaded for the Kaggle competition, change
        # the variable below to False
        demo = True
        data_dir = '../data/kaggle_dog'
        if demo:
            zipfiles = ['train_valid_test_tiny.zip']
        else:
            zipfiles = ['train.zip', 'test.zip', 'labels.csv.zip']
        for f in zipfiles:
            with zipfile.ZipFile(data_dir + '/' + f, 'r') as z:
                z.extractall(data_dir)
```

Organize the Data Set

Next, we define the `reorg_train_valid` function to segment the validation set from the original Kaggle competition training set. The parameter `valid_ratio` in this function is the ratio of the number

---

of examples of each dog breed in the validation set to the number of examples of the breed with the least examples (66) in the original training set. After organizing the data, images of the same breed will be placed in the same folder so that we can read them later.

```
In [3]: def reorg_train_valid(data_dir, train_dir, input_dir, valid_ratio, idx_label):
            # The number of examples of the least represented breed in the training
            # set
            min_n_train_per_label = (
                collections.Counter(idx_label.values()).most_common()[:-2:-1][0][1])
            # The number of examples of each breed in the validation set
            n_valid_per_label = math.floor(min_n_train_per_label * valid_ratio)
            label_count = {}
            for train_file in os.listdir(os.path.join(data_dir, train_dir)):
                idx = train_file.split('.')[0]
                label = idx_label[idx]
                d2l.mkdir_if_not_exist([data_dir, input_dir, 'train_valid', label])
                shutil.copy(os.path.join(data_dir, train_dir, train_file),
                            os.path.join(data_dir, input_dir, 'train_valid', label))
                if label not in label_count or label_count[label] < n_valid_per_label:
                    d2l.mkdir_if_not_exist([data_dir, input_dir, 'valid', label])
                    shutil.copy(os.path.join(data_dir, train_dir, train_file),
                                os.path.join(data_dir, input_dir, 'valid', label))
                    label_count[label] = label_count.get(label, 0) + 1
                else:
                    d2l.mkdir_if_not_exist([data_dir, input_dir, 'train', label])
                    shutil.copy(os.path.join(data_dir, train_dir, train_file),
                                os.path.join(data_dir, input_dir, 'train', label))
```

The `reorg_dog_data` function below is used to read the training data labels, segment the validation set, and organize the training set.

```
In [4]: def reorg_dog_data(data_dir, label_file, train_dir, test_dir, input_dir,
                           valid_ratio):
            # Read the training data labels
            with open(os.path.join(data_dir, label_file), 'r') as f:
                # Skip the file header line (column name)
                lines = f.readlines()[1:]
                tokens = [l.rstrip().split(',') for l in lines]
                idx_label = dict(((idx, label) for idx, label in tokens))
            reorg_train_valid(data_dir, train_dir, input_dir, valid_ratio, idx_label)
            # Organize the training set
            d2l.mkdir_if_not_exist([data_dir, input_dir, 'test', 'unknown'])
            for test_file in os.listdir(os.path.join(data_dir, test_dir)):
                shutil.copy(os.path.join(data_dir, test_dir, test_file),
                            os.path.join(data_dir, input_dir, 'test', 'unknown'))
```

Because we are using a small data set, we set the batch size to 1. During actual training and testing, we would use the entire Kaggle Competition data set and call the `reorg_dog_data` function to organize the data set. Likewise, we would need to set the `batch_size` to a larger integer, such as 128.

```
In [5]: if demo:
            # Note: Here, we use a small data set and the batch size should be set
            # smaller. When using the complete data set for the Kaggle competition, we
            # can set the batch size to a larger integer
            input_dir, batch_size = 'train_valid_test_tiny', 1
        else:
```

```
                    label_file, train_dir, test_dir = 'labels.csv', 'train', 'test'
                    input_dir, batch_size, valid_ratio = 'train_valid_test', 128, 0.1
                    reorg_dog_data(data_dir, label_file, train_dir, test_dir, input_dir,
                                   valid_ratio)
```

10.13.2 Image Augmentation

The size of the images in this section are larger than the images in the previous section. Here are some more image augmentation operations that might be useful.

```
In [6]: transform_train = gdata.vision.transforms.Compose([
            # Randomly crop the image to obtain an image with an area of 0.08 to 1 of
            # the original area and height to width ratio between 3/4 and 4/3. Then,
            # scale the image to create a new image with a height and width of 224
            # pixels each
            gdata.vision.transforms.RandomResizedCrop(224, scale=(0.08, 1.0),
                                                      ratio=(3.0/4.0, 4.0/3.0)),
            gdata.vision.transforms.RandomFlipLeftRight(),
            # Randomly change the brightness, contrast, and saturation
            gdata.vision.transforms.RandomColorJitter(brightness=0.4, contrast=0.4,
                                                      saturation=0.4),
            # Add random noise
            gdata.vision.transforms.RandomLighting(0.1),
            gdata.vision.transforms.ToTensor(),
            # Standardize each channel of the image
            gdata.vision.transforms.Normalize([0.485, 0.456, 0.406],
                                              [0.229, 0.224, 0.225])])
```

During testing, we only use definite image preprocessing operations.

```
In [7]: transform_test = gdata.vision.transforms.Compose([
            gdata.vision.transforms.Resize(256),
            # Crop a square of 224 by 224 from the center of the image
            gdata.vision.transforms.CenterCrop(224),
            gdata.vision.transforms.ToTensor(),
            gdata.vision.transforms.Normalize([0.485, 0.456, 0.406],
                                              [0.229, 0.224, 0.225])])
```

10.13.3 Read the Data Set

As in the previous section, we can create an `ImageFolderDataset` instance to read the data set containing the original image files.

```
In [8]: train_ds = gdata.vision.ImageFolderDataset(
            os.path.join(data_dir, input_dir, 'train'), flag=1)
        valid_ds = gdata.vision.ImageFolderDataset(
            os.path.join(data_dir, input_dir, 'valid'), flag=1)
        train_valid_ds = gdata.vision.ImageFolderDataset(
            os.path.join(data_dir, input_dir, 'train_valid'), flag=1)
        test_ds = gdata.vision.ImageFolderDataset(
            os.path.join(data_dir, input_dir, 'test'), flag=1)
```

Here, we create a `DataLoader` instance, just like in the previous section.

10.13. Dog Breed Identification (ImageNet Dogs) on Kaggle   **523**

```
In [9]: train_iter = gdata.DataLoader(train_ds.transform_first(transform_train),
                                      batch_size, shuffle=True, last_batch='keep')
        valid_iter = gdata.DataLoader(valid_ds.transform_first(transform_test),
                                      batch_size, shuffle=True, last_batch='keep')
        train_valid_iter = gdata.DataLoader(train_valid_ds.transform_first(
            transform_train), batch_size, shuffle=True, last_batch='keep')
        test_iter = gdata.DataLoader(test_ds.transform_first(transform_test),
                                      batch_size, shuffle=False, last_batch='keep')
```

10.13.4 Define the Model

The data set for this competition is a subset of the ImageNet data set. Therefore, we can use the approach discussed in the *Fine Tuning* section to select a model pre-trained on the entire ImageNet data set and use it to extract image features to be input in the custom small-scale output network. Gluon provides a wide range of pre-trained models. Here, we will use the pre-trained ResNet-34 model. Because the competition data set is a subset of the pre-training data set, we simply reuse the input of the pre-trained model's output layer, i.e. the extracted features. Then, we can replace the original output layer with a small custom output network that can be trained, such as two fully connected layers in a series. Different from the experiment in the *Fine Tuning* section, here, we do not retrain the pre-trained model used for feature extraction. This reduces the training time and the memory required to store model parameter gradients.

You must note that, during image augmentation, we use the mean values and standard deviations of the three RGB channels for the entire ImageNet data set for normalization. This is consistent with the normalization of the pre-trained model.

```
In [10]: def get_net(ctx):
             finetune_net = model_zoo.vision.resnet34_v2(pretrained=True)
             # Define a new output network
             finetune_net.output_new = nn.HybridSequential(prefix='')
             finetune_net.output_new.add(nn.Dense(256, activation='relu'))
             # There are 120 output categories
             finetune_net.output_new.add(nn.Dense(120))
             # Initialize the output network
             finetune_net.output_new.initialize(init.Xavier(), ctx=ctx)
             # Distribute the model parameters to the CPUs or GPUs used for computation
             finetune_net.collect_params().reset_ctx(ctx)
             return finetune_net
```

When calculating the loss, we first use the member variable `features` to obtain the input of the pre-trained model's output layer, i.e. the extracted feature. Then, we use this feature as the input for our small custom output network and compute the output.

```
In [11]: loss = gloss.SoftmaxCrossEntropyLoss()

         def evaluate_loss(data_iter, net, ctx):
             l_sum, n = 0.0, 0
             for X, y in data_iter:
                 y = y.as_in_context(ctx)
                 output_features = net.features(X.as_in_context(ctx))
                 outputs = net.output_new(output_features)
                 l_sum += loss(outputs, y).sum().asscalar()
```

```
            n += y.size
        return l_sum / n
```

10.13.5 Define the Training Functions

We will select the model and tune hyper-parameters according to the model's performance on the validation set. The model training function `train` only trains the small custom output network.

```
In [12]: def train(net, train_iter, valid_iter, num_epochs, lr, wd, ctx, lr_period,
                   lr_decay):
             # Only train the small custom output network
             trainer = gluon.Trainer(net.output_new.collect_params(), 'sgd',
                                     {'learning_rate': lr, 'momentum': 0.9, 'wd': wd})
             for epoch in range(num_epochs):
                 train_l_sum, n, start = 0.0, 0, time.time()
                 if epoch > 0 and epoch % lr_period == 0:
                     trainer.set_learning_rate(trainer.learning_rate * lr_decay)
                 for X, y in train_iter:
                     y = y.as_in_context(ctx)
                     output_features = net.features(X.as_in_context(ctx))
                     with autograd.record():
                         outputs = net.output_new(output_features)
                         l = loss(outputs, y).sum()
                     l.backward()
                     trainer.step(batch_size)
                     train_l_sum += l.asscalar()
                     n += y.size
                 time_s = "time %.2f sec" % (time.time() - start)
                 if valid_iter is not None:
                     valid_loss = evaluate_loss(valid_iter, net, ctx)
                     epoch_s = ("epoch %d, train loss %f, valid loss %f, "
                                % (epoch + 1, train_l_sum / n, valid_loss))
                 else:
                     epoch_s = ("epoch %d, train loss %f, "
                                % (epoch + 1, train_l_sum / n))
                 print(epoch_s + time_s + ', lr ' + str(trainer.learning_rate))
```

10.13.6 Train and Validate the Model

Now, we can train and validate the model. The following hyper-parameters can be tuned. For example, we can increase the number of epochs. Because `lr_period` and `lr_decay` are set to 10 and 0.1 respectively, the learning rate of the optimization algorithm will be multiplied by 0.1 after every 10 epochs.

```
In [13]: ctx, num_epochs, lr, wd = d2l.try_gpu(), 1, 0.01, 1e-4
         lr_period, lr_decay, net = 10, 0.1, get_net(ctx)
         net.hybridize()
         train(net, train_iter, valid_iter, num_epochs, lr, wd, ctx, lr_period,
               lr_decay)

epoch 1, train loss 5.211201, valid loss 4.751868, time 2.83 sec, lr 0.01
```

10.13.7 Classify the Testing Set and Submit Results on Kaggle

After obtaining a satisfactory model design and hyper-parameters, we use all training data sets (including validation sets) to retrain the model and then classify the testing set. Note that predictions are made by the output network we just trained.

```
In [14]: net = get_net(ctx)
         net.hybridize()
         train(net, train_valid_iter, None, num_epochs, lr, wd, ctx, lr_period,
               lr_decay)

         preds = []
         for data, label in test_iter:
             output_features = net.features(data.as_in_context(ctx))
             output = nd.softmax(net.output_new(output_features))
             preds.extend(output.asnumpy())
         ids = sorted(os.listdir(os.path.join(data_dir, input_dir, 'test/unknown')))
         with open('submission.csv', 'w') as f:
             f.write('id,' + ','.join(train_valid_ds.synsets) + '\n')
             for i, output in zip(ids, preds):
                 f.write(i.split('.')[0] + ',' + ','.join(
                     [str(num) for num in output]) + '\n')

epoch 1, train loss 5.050728, time 3.68 sec, lr 0.01
```

After executing the above code, we will generate a submission.csv file. The format of this file is consistent with the Kaggle competition requirements. The method for submitting results is similar to method in the Get Started with Kaggle Competition: Predicting House Prices section.

Summary

- We can use a model pre-trained on the ImageNet data set to extract features and only train a small custom output network. This will allow us to classify a subset of the ImageNet data set with lower computing and storage overhead.

Exercises

- When using the entire Kaggle data set, what kind of results do you get when you increase the `batch_size` (batch size) and `num_epochs` (number of epochs)?
- Do you get better results if you use a deeper pre-trained model?
- Scan the QR code to access the relevant discussions and exchange ideas about the methods used and the results obtained with the community. Can you come up with any better techniques?

Reference

[1] Kaggle ImageNet Dog Breed Identification website. https://www.kaggle.com/c/

dog-breed-identification

Scan the QR Code to Discuss

11

Natural Language Processing

Natural language processing is concerned with interactions between computers and humans that use natural language. In practice, it is very common for us to use this technique to process and analyze large amounts of natural language data, like the language models from the Recurrent Neural Networks section.

In this chapter, we will discuss how to use vectors to represent words and train the word vectors on a corpus. We will also use word vectors pre-trained on a larger corpus to find synonyms and analogies. Then, in the text classification task, we will use word vectors to analyze the emotion of a text and explain the important ideas of timing data classification based on RNNs and the convolutional neural networks. In addition, many of the outputs of natural language processing tasks are not fixed, such as sentences of arbitrary length. We will introduce the encoder-decoder model, beam search, and attention mechanisms to address problems of this type and apply them to machine translation.

11.1 Word Embedding (word2vec)

A natural language is a complex system that we use to communicate. Words are commonly used as the unit of analysis in natural language processing. As its name implies, a word vector is a vector used to represent a word. It can also be thought of as the feature vector of a word. The technique of mapping words to vectors of real numbers is also known as word embedding. Over the last few years, word embedding has gradually become basic knowledge in natural language processing.

11.1.1 Why not Use One-hot Vectors?

We used one-hot vectors to represent words (characters are words) in the *Implementation of the Recurrent Neural Network from Scratch* section. Recall that when we assume the number of different words in a dictionary (the dictionary size) is N, each word can correspond one-to-one with consecutive integers from 0 to $N - 1$. These integers that correspond to words are called the indices of the words. We assume that the index of a word is i. In order to get the one-hot vector representation of the word, we create a vector of all 0s with a length of N and set element i to 1. In this way, each word is represented as a vector of length N that can be used directly by the neural network.

Although one-hot word vectors are easy to construct, they are usually not a good choice. One of the major reasons is that the one-hot word vectors cannot accurately express the similarity between different words, such as the cosine similarity that we commonly use. For the vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^d$, their cosine similarities are the cosines of the angles between them:

$$\frac{\boldsymbol{x}^\top \boldsymbol{y}}{\|\boldsymbol{x}\|\|\boldsymbol{y}\|} \in [-1, 1].$$

Since the cosine similarity between the one-hot vectors of any two different words is 0, it is difficult to use the one-hot vector to accurately represent the similarity between multiple different words.

Word2vec is a tool that we came up with to solve the problem above[1]. It represents each word with a fixed-length vector and uses these vectors to better indicate the similarity and analogy relationships between different words. The Word2vec tool contains two models: skip-gram[2] and continuous bag of words (CBOW)[3]. Next, we will take a look at the two models and their training methods.

11.1.2 The Skip-Gram Model

The skip-gram model assumes that a word can be used to generate the words that surround it in a text sequence. For example, we assume that the text sequence is the, man, loves, his, and son. We use loves as the central target word and set the context window size to 2. As shown in Figure 10.1, given the central target word loves, the skip-gram model is concerned with the conditional probability for generating the context words, the, man, his and son, that are within a distance of no more than 2 words, which is

$$\mathbb{P}(\text{``the''}, \text{``man''}, \text{``his''}, \text{``son''} \mid \text{``loves''}).$$

We assume that, given the central target word, the context words are generated independently of each other. In this case, the formula above can be rewritten as

$$\mathbb{P}(\text{``the''} \mid \text{``loves''}) \cdot \mathbb{P}(\text{``man''} \mid \text{``loves''}) \cdot \mathbb{P}(\text{``his''} \mid \text{``loves''}) \cdot \mathbb{P}(\text{``son''} \mid \text{``loves''}).$$
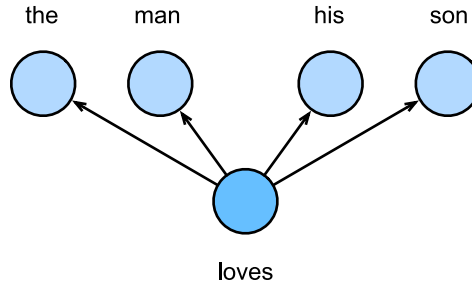
Fig. 11.1: The skip-gram model cares about the conditional probability of generating context words for a given central target word.

In the skip-gram model, each word is represented as two d-dimension vectors, which are used to compute the conditional probability. We assume that the word is indexed as i in the dictionary, its vector is represented as $\boldsymbol{v}_i \in \mathbb{R}^d$ when it is the central target word, and $\boldsymbol{u}_i \in \mathbb{R}^d$ when it is a context word. Let the central target word w_c and context word w_o be indexed as c and o respectively in the dictionary. The conditional probability of generating the context word for the given central target word can be obtained by performing a softmax operation on the vector inner product:

$$\mathbb{P}(w_o \mid w_c) = \frac{\exp(\boldsymbol{u}_o^\top \boldsymbol{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\boldsymbol{u}_i^\top \boldsymbol{v}_c)},$$

where vocabulary index set $\mathcal{V} = \{0, 1, \ldots, |\mathcal{V}| - 1\}$. Assume that a text sequence of length T is given, where the word at time step t is denoted as $w^{(t)}$. Assume that context words are independently generated given center words. When context window size is m, the likelihood function of the skip-gram model is the joint probability of generating all the context words given any center word

$$\prod_{t=1}^{T} \prod_{-m \leq j \leq m, \ j \neq 0} \mathbb{P}(w^{(t+j)} \mid w^{(t)}),$$

Here, any time step that is less than 1 or greater than T can be ignored.

Skip-Gram Model Training

The skip-gram model parameters are the central target word vector and context word vector for each individual word. In the training process, we are going to learn the model parameters by maximizing the likelihood function, which is also known as maximum likelihood estimation. This is equivalent to minimizing the following loss function:

$$-\sum_{t=1}^{T} \sum_{-m \leq j \leq m, \ j \neq 0} \log \mathbb{P}(w^{(t+j)} \mid w^{(t)}).$$

If we use the SGD, in each iteration we are going to pick a shorter subsequence through random sampling to compute the loss for that subsequence, and then compute the gradient to update the model parameters. The key of gradient computation is to compute the gradient of the logarithmic conditional probability for the central word vector and the context word vector. By definition, we first have

$$\log \mathbb{P}(w_o \mid w_c) = \boldsymbol{u}_o^\top \boldsymbol{v}_c - \log \left(\sum_{i \in \mathcal{V}} \exp(\boldsymbol{u}_i^\top \boldsymbol{v}_c) \right)$$

Through differentiation, we can get the gradient $\boldsymbol{v}_c$ from the formula above.

$$\begin{aligned} \frac{\partial \log \mathbb{P}(w_o \mid w_c)}{\partial \boldsymbol{v}_c} &= \boldsymbol{u}_o - \frac{\sum_{j \in \mathcal{V}} \exp(\boldsymbol{u}_j^\top \boldsymbol{v}_c) \boldsymbol{u}_j}{\sum_{i \in \mathcal{V}} \exp(\boldsymbol{u}_i^\top \boldsymbol{v}_c)} \\ &= \boldsymbol{u}_o - \sum_{j \in \mathcal{V}} \left(\frac{\exp(\boldsymbol{u}_j^\top \boldsymbol{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\boldsymbol{u}_i^\top \boldsymbol{v}_c)} \right) \boldsymbol{u}_j \\ &= \boldsymbol{u}_o - \sum_{j \in \mathcal{V}} \mathbb{P}(w_j \mid w_c) \boldsymbol{u}_j. \end{aligned}$$

Its computation obtains the conditional probability for all the words in the dictionary given the central target word w_c. We then use the same method to obtain the gradients for other word vectors.

After the training, for any word in the dictionary with index i, we are going to get its two word vector sets $\boldsymbol{v}_i$ and $\boldsymbol{u}_i$. In applications of natural language processing (NLP), the central target word vector in the skip-gram model is generally used as the representation vector of a word.

11.1.3 The Continuous Bag Of Words (CBOW) Model

The continuous bag of words (CBOW) model is similar to the skip-gram model. The biggest difference is that the CBOW model assumes that the central target word is generated based on the context words before and after it in the text sequence. With the same text sequence the, man, loves, his and son, in which loves is the central target word, given a context window size of 2, the CBOW model is concerned with the conditional probability of generating the target word loves based on the context words the, man, his and son(as shown in Figure 10.2), such as

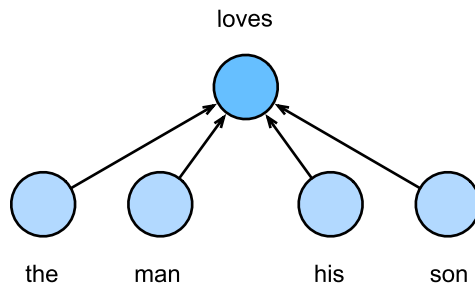$$\mathbb{P}(\text{``loves" } \mid \text{ ``the", ``man", ``his", ``son"}).$$

Fig. 11.2: The CBOW model cares about the conditional probability of generating the central target word from given context words.

Since there are multiple context words in the CBOW model, we will average their word vectors and then use the same method as the skip-gram model to compute the conditional probability. We assume that $\boldsymbol{v_i} \in \mathbb{R}^d$ and $\boldsymbol{u_i} \in \mathbb{R}^d$ are the context word vector and central target word vector of the word with index i in the dictionary (notice that the symbols are opposite to the ones in the skip-gram model). Let central target word w_c be indexed as c, and context words $w_{o_1}, \ldots, w_{o_{2m}}$ be indexed as $o_1, \ldots, o_{2m}$ in the dictionary. Thus, the conditional probability of generating a central target word from the given context word is

$$\mathbb{P}(w_c \mid w_{o_1}, \ldots, w_{o_{2m}}) = \frac{\exp\left(\frac{1}{2m}\boldsymbol{u}_c^\top (\boldsymbol{v}_{o_1} + \ldots + \boldsymbol{v}_{o_{2m}})\right)}{\sum_{i \in \mathcal{V}} \exp\left(\frac{1}{2m}\boldsymbol{u}_i^\top (\boldsymbol{v}_{o_1} + \ldots + \boldsymbol{v}_{o_{2m}})\right)}.$$

For brevity, denote $\mathcal{W}_o = \{w_{o_1}, \ldots, w_{o_{2m}}\}$, and $\bar{\boldsymbol{v}}_o = (\boldsymbol{v}_{o_1} + \ldots + \boldsymbol{v}_{o_{2m}})/(2m)$. The equation above can be simplified as

$$\mathbb{P}(w_c \mid \mathcal{W}_o) = \frac{\exp\left(\boldsymbol{u}_c^\top \bar{\boldsymbol{v}}_o\right)}{\sum_{i \in \mathcal{V}} \exp\left(\boldsymbol{u}_i^\top \bar{\boldsymbol{v}}_o\right)}.$$

Given a text sequence of length T, we assume that the word at time step t is $w^{(t)}$, and the context window size is m. The likelihood function of the CBOW model is the probability of generating any central target word from the context words.

$$\prod_{t=1}^{T} \mathbb{P}(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}).$$

CBOW Model Training

CBOW model training is quite similar to skip-gram model training. The maximum likelihood estimation of the CBOW model is equivalent to minimizing the loss function.

$$-\sum_{t=1}^{T} \log \mathbb{P}(w^{(t)} \mid w^{(t-m)}, \ldots, w^{(t-1)}, w^{(t+1)}, \ldots, w^{(t+m)}).$$

Notice that

$$\log \mathbb{P}(w_c \mid \mathcal{W}_o) = \boldsymbol{u}_c^\top \bar{\boldsymbol{v}}_o - \log \left(\sum_{i \in \mathcal{V}} \exp \left(\boldsymbol{u}_i^\top \bar{\boldsymbol{v}}_o \right) \right).$$

Through differentiation, we can compute the logarithm of the conditional probability of the gradient of any context word vector $\boldsymbol{v}_{o_i}(i = 1, \ldots, 2m)$ in the formula above.

$$\frac{\partial \log \mathbb{P}(w_c \mid \mathcal{W}_o)}{\partial \boldsymbol{v}_{o_i}} = \frac{1}{2m} \left(\boldsymbol{u}_c - \sum_{j \in \mathcal{V}} \frac{\exp(\boldsymbol{u}_j^\top \bar{\boldsymbol{v}}_o)\boldsymbol{u}_j}{\sum_{i \in \mathcal{V}} \exp(\boldsymbol{u}_i^\top \bar{\boldsymbol{v}}_o)} \right) = \frac{1}{2m} \left(\boldsymbol{u}_c - \sum_{j \in \mathcal{V}} \mathbb{P}(w_j \mid \mathcal{W}_o)\boldsymbol{u}_j \right).$$

We then use the same method to obtain the gradients for other word vectors. Unlike the skip-gram model, we usually use the context word vector as the representation vector for a word in the CBOW model.

Summary

- A word vector is a vector used to represent a word. The technique of mapping words to vectors of real numbers is also known as word embedding.

- Word2vec includes both the continuous bag of words (CBOW) and skip-gram models. The skip-gram model assumes that context words are generated based on the central target word. The CBOW model assumes that the central target word is generated based on the context words.

Exercises

- What is the computational complexity of each gradient? If the dictionary contains a large volume of words, what problems will this cause?

- There are some fixed phrases in the English language which consist of multiple words, such as new york. How can you train their word vectors? Hint: See section 4 in the Word2vec paper[2].

- Use the skip-gram model as an example to think about the design of a word2vec model. What is the relationship between the inner product of two word vectors and the cosine similarity in the skip-gram model? For a pair of words with close semantical meaning, why it is likely for their word vector cosine similarity to be high?

Reference

[1] Word2vec tool. https://code.google.com/archive/p/word2vec/

[2] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. In Advances in neural information processing systems (pp. 3111-3119).

[3] Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781.

Scan the QR Code to Discuss

11.2 Approximate Training

Recall content of the last section. The core feature of the skip-gram model is the use of softmax operations to compute the conditional probability of generating context word w_o based on the given central target word w_c.

$$\mathbb{P}(w_o \mid w_c) = \frac{\exp(\boldsymbol{u}_o^\top \boldsymbol{v}_c)}{\sum_{i \in \mathcal{V}} \exp(\boldsymbol{u}_i^\top \boldsymbol{v}_c)}.$$

The logarithmic loss corresponding to the conditional probability is given as

$$-\log \mathbb{P}(w_o \mid w_c) = -\boldsymbol{u}_o^\top \boldsymbol{v}_c + \log \left(\sum_{i \in \mathcal{V}} \exp(\boldsymbol{u}_i^\top \boldsymbol{v}_c) \right).$$

Because the softmax operation has considered that the context word could be any word in the dictionary $\mathcal{V}$, the loss mentioned above actually includes the sum of the number of items in the dictionary size. From the last section, we know that for both the skip-gram model and CBOW model, because they both get the conditional probability using a softmax operation, the gradient computation for each step contains the sum of the number of items in the dictionary size. For larger dictionaries with hundreds of thousands or even millions of words, the overhead for computing each gradient may be too high. In order to reduce such computational complexity, we will introduce two approximate training methods in this section: negative sampling and hierarchical softmax. Since there is no major difference between the skip-gram model and the CBOW model, we will only use the skip-gram model as an example to introduce these two training methods in this section.

11.2.1 Negative Sampling

Negative sampling modifies the original objective function. Given a context window for the central target word w_c, we will treat it as an event for context word w_o to appear in the context window and compute the probability of this event from

$$\mathbb{P}(D = 1 \mid w_c, w_o) = \sigma(\boldsymbol{u}_o^\top \boldsymbol{v}_c),$$

Here, the σ function has the same definition as the sigmoid activation function:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}.$$

We will first consider training the word vector by maximizing the joint probability of all events in the text sequence. Given a text sequence of length T, we assume that the word at time step t is $w^{(t)}$ and the context window size is m. Now we consider maximizing the joint probability

$$\prod_{t=1}^{T} \prod_{-m \leq j \leq m, \ j \neq 0} \mathbb{P}(D = 1 \mid w^{(t)}, w^{(t+j)}).$$

However, the events included in the model only consider positive examples. In this case, only when all the word vectors are equal and their values approach infinity can the joint probability above be maximized to 1. Obviously, such word vectors are meaningless. Negative sampling makes the objective function more meaningful by sampling with an addition of negative examples. Assume that event P occurs when context word w_o to appear in the context window of central target word w_c, and we sample K words that do not appear in the context window according to the distribution $\mathbb{P}(w)$ to act as noise words. We assume the event for noise word $w_k(k = 1, \ldots, K)$ to not appear in the context window of central target word w_c is N_k. Suppose that events P and $N_1, \ldots, N_K$ for both positive and negative examples are independent of each other. By considering negative sampling, we can rewrite the joint probability above, which only considers the positive examples, as

$$\prod_{t=1}^{T} \prod_{-m \leq j \leq m, \ j \neq 0} \mathbb{P}(w^{(t+j)} \mid w^{(t)}),$$

Here, the conditional probability is approximated to be

$$\mathbb{P}(w^{(t+j)} \mid w^{(t)}) = \mathbb{P}(D = 1 \mid w^{(t)}, w^{(t+j)}) \prod_{k=1, \ w_k \sim \mathbb{P}(w)}^{K} \mathbb{P}(D = 0 \mid w^{(t)}, w_k).$$

Let the text sequence index of word $w^{(t)}$ at time step t be i_t and h_k for noise word w_k in the dictionary. The logarithmic loss for the conditional probability above is

$$-\log \mathbb{P}(w^{(t+j)} \mid w^{(t)}) = -\log \mathbb{P}(D = 1 \mid w^{(t)}, w^{(t+j)}) - \sum_{k=1,\ w_k \sim \mathbb{P}(w)}^{K} \log \mathbb{P}(D = 0 \mid w^{(t)}, w_k)$$

$$= -\log \sigma \left(\boldsymbol{u}_{i_{t+j}}^{\top} \boldsymbol{v}_{i_t} \right) - \sum_{k=1,\ w_k \sim \mathbb{P}(w)}^{K} \log \left(1 - \sigma \left(\boldsymbol{u}_{h_k}^{\top} \boldsymbol{v}_{i_t} \right) \right)$$

$$= -\log \sigma \left(\boldsymbol{u}_{i_{t+j}}^{\top} \boldsymbol{v}_{i_t} \right) - \sum_{k=1,\ w_k \sim \mathbb{P}(w)}^{K} \log \sigma \left(-\boldsymbol{u}_{h_k}^{\top} \boldsymbol{v}_{i_t} \right).$$

Here, the gradient computation in each step of the training is no longer related to the dictionary size, but linearly related to K. When K takes a smaller constant, the negative sampling has a lower computational overhead for each step.

11.2.2 Hierarchical Softmax

Hierarchical softmax is another type of approximate training method. It uses a binary tree for data structure, with the leaf nodes of the tree representing every word in the dictionary $\mathcal{V}$.
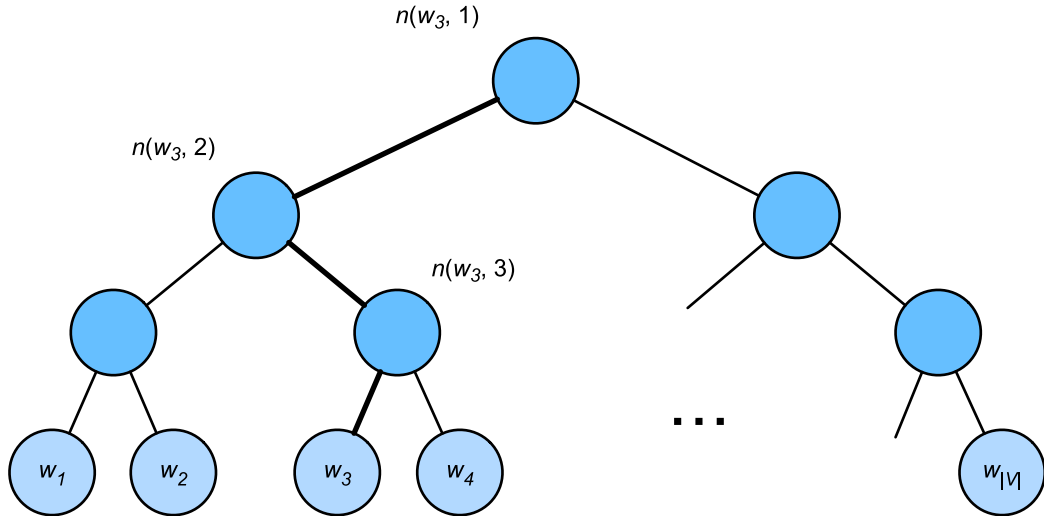

Fig. 11.3: Hierarchical Softmax. Each leaf node of the tree represents a word in the dictionary.

We assume that $L(w)$ is the number of nodes on the path (including the root and leaf nodes) from the root node of the binary tree to the leaf node of word w. Let $n(w, j)$ be the jth node on this path, with

the context word vector $\mathbf{u}_{n(w,j)}$. We use Figure 10.3 as an example, so $L(w_3) = 4$. Hierarchical softmax will approximate the conditional probability in the skip-gram model as

$$\mathbb{P}(w_o \mid w_c) = \prod_{j=1}^{L(w_o)-1} \sigma\left([\![n(w_o, j+1) = \text{leftChild}(n(w_o, j))]\!] \cdot \mathbf{u}_{n(w_o,j)}^\top \mathbf{v}_c \right),$$

Here the σ function has the same definition as the sigmoid activation function, and leftChild(n) is the left child node of node n. If x is true, $[\![x]\!] = 1$; otherwise $[\![x]\!] = -1$. Now, we will compute the conditional probability of generating word w_3 based on the given word w_c in Figure 10.3. We need to find the inner product of word vector $\mathbf{v}_c$ (for word w_c) and each non-leaf node vector on the path from the root node to w_3. Because, in the binary tree, the path from the root node to leaf node w_3 needs to be traversed left, right, and left again (the path with the bold line in Figure 10.3), we get

$$\mathbb{P}(w_3 \mid w_c) = \sigma(\mathbf{u}_{n(w_3,1)}^\top \mathbf{v}_c) \cdot \sigma(-\mathbf{u}_{n(w_3,2)}^\top \mathbf{v}_c) \cdot \sigma(\mathbf{u}_{n(w_3,3)}^\top \mathbf{v}_c).$$

Because $\sigma(x) + \sigma(-x) = 1$, the condition that the sum of the conditional probability of any word generated based on the given central target word w_c in dictionary $\mathcal{V}$ be 1 will also suffice:

$$\sum_{w \in \mathcal{V}} \mathbb{P}(w \mid w_c) = 1.$$

In addition, because the order of magnitude for $L(w_o) - 1$ is $\mathcal{O}(\log_2 |\mathcal{V}|)$, when the size of dictionary $\mathcal{V}$ is large, the computational overhead for each step in the hierarchical softmax training is greatly reduced compared to situations where we do not use approximate training.

Summary

- Negative sampling constructs the loss function by considering independent events that contain both positive and negative examples. The gradient computational overhead for each step in the training process is linearly related to the number of noise words we sample.

- Hierarchical softmax uses a binary tree and constructs the loss function based on the path from the root node to the leaf node. The gradient computational overhead for each step in the training process is related to the logarithm of the dictionary size.

Exercises

- Before reading the next section, think about how we should sample noise words in negative sampling.

- What makes the last formula in this section hold?

- How can we apply negative sampling and hierarchical softmax in the skip-gram model?

11.3 Implementation of Word2vec

This section is a practice exercise for the two previous sections. We use the skip-gram model from the *Word Embedding (word2vec)* section and negative sampling from the *Approximate Training* section as examples to introduce the implementation of word embedding model training on a corpus. We will also introduce some implementation tricks, such as subsampling and mask variables.

First, import the packages and modules required for the experiment.

```
In [1]: import sys
        sys.path.insert(0, '..')

        import collections
        import d2l
        import math
        from mxnet import autograd, gluon, nd
        from mxnet.gluon import data as gdata, loss as gloss, nn
        import random
        import sys
        import time
        import zipfile
```

11.3.1 Process the Data Set

Penn Tree Bank (PTB) is a small commonly-used corpus[1]. It takes samples from Wall Street Journal articles and includes training sets, validation sets, and test sets. We will train the word embedding model on the PTB training set. Each line of the data set acts as a sentence. All the words in a sentence are separated by spaces.

```
In [2]: with zipfile.ZipFile('../data/ptb.zip', 'r') as zin:
            zin.extractall('../data/')

        with open('../data/ptb/ptb.train.txt', 'r') as f:
            lines = f.readlines()
            # st is the abbreviation of "sentence" in the loop
            raw_dataset = [st.split() for st in lines]

        '# sentences: %d' % len(raw_dataset)

Out[2]: '# sentences: 42068'
```

For the first three sentences of the data set, print the number of words and the first five words of each sentence. The end character of this data set is <eos>, uncommon words are all represented by <unk>, and numbers are replaced with N.

```
In [3]: for st in raw_dataset[:3]:
            print('# tokens:', len(st), st[:5])

# tokens: 24 ['aer', 'banknote', 'berlitz', 'calloway', 'centrust']
# tokens: 15 ['pierre', '<unk>', 'N', 'years', 'old']
# tokens: 11 ['mr.', '<unk>', 'is', 'chairman', 'of']
```

Create Word Index

For the sake of simplicity, we only keep words that appear at least 5 times in the data set.

```
In [4]: # tk is an abbreviation for "token" in the loop
        counter = collections.Counter([tk for st in raw_dataset for tk in st])
        counter = dict(filter(lambda x: x[1] >= 5, counter.items()))
```

Then, map the words to the integer indexes.

```
In [5]: idx_to_token = [tk for tk, _ in counter.items()]
        token_to_idx = {tk: idx for idx, tk in enumerate(idx_to_token)}
        dataset = [[token_to_idx[tk] for tk in st if tk in token_to_idx]
                   for st in raw_dataset]
        num_tokens = sum([len(st) for st in dataset])
        '# tokens: %d' % num_tokens

Out[5]: '# tokens: 887100'
```

Subsampling

In text data, there are generally some words that appear at high frequencies, such the, a, and in in English. Generally speaking, in a context window, it is better to train the word embedding model when a word (such as chip) and a lower-frequency word (such as microprocessor) appear at the same time, rather than when a word appears with a higher-frequency word (such as the). Therefore, when training the word embedding model, we can perform subsampling[2] on the words. Specifically, each indexed word w_i in the data set will drop out at a certain probability. The dropout probability is given as:

$$\mathbb{P}(w_i) = \max\left(1 - \sqrt{\frac{t}{f(w_i)}}, 0\right),$$

Here, $f(w_i)$ is the ratio of the instances of word w_i to the total number of words in the data set, and the constant t is a hyper-parameter (set to 10^{-4} in this experiment). As we can see, it is only possible to drop out the word w_i in subsampling when $f(w_i) > t$. The higher the word's frequency, the higher its dropout probability.

```
In [6]: def discard(idx):
            return random.uniform(0, 1) < 1 - math.sqrt(
                1e-4 / counter[idx_to_token[idx]] * num_tokens)
```

```
          subsampled_dataset = [[tk for tk in st if not discard(tk)] for st in dataset]
          '# tokens: %d' % sum([len(st) for st in subsampled_dataset])

Out[6]: '# tokens: 376210'
```

As we can see, we have removed about half of the words after the second sampling. The following compares the number of times a word appears in the data set before and after subsampling. The sampling rate of the high-frequency word the is less than 1/20.

```
In [7]: def compare_counts(token):
            return '# %s: before=%d, after=%d' % (token, sum(
                [st.count(token_to_idx[token]) for st in dataset]), sum(
                [st.count(token_to_idx[token]) for st in subsampled_dataset]))

        compare_counts('the')

Out[7]: '# the: before=50770, after=2139'
```

But the low-frequency word join is completely preserved.

```
In [8]: compare_counts('join')

Out[8]: '# join: before=45, after=45'
```

Extract Central Target Words and Context Words

We use words with a distance from the central target word not exceeding the context window size as the context words of the given center target word. The following definition function extracts all the central target words and their context words. It uniformly and randomly samples an integer to be used as the context window size between integer 1 and the max_window_size (maximum context window).

```
In [9]: def get_centers_and_contexts(dataset, max_window_size):
            centers, contexts = [], []
            for st in dataset:
                # Each sentence needs at least 2 words to form a
                # "central target word - context word" pair
                if len(st) < 2:
                    continue
                centers += st
                for center_i in range(len(st)):
                    window_size = random.randint(1, max_window_size)
                    indices = list(range(max(0, center_i - window_size),
                                         min(len(st), center_i + 1 + window_size)))
                    # Exclude the central target word from the context words
                    indices.remove(center_i)
                    contexts.append([st[idx] for idx in indices])
            return centers, contexts
```

Next, we create an artificial data set containing two sentences of 7 and 3 words, respectively. Assume the maximum context window is 2 and print all the central target words and their context words.

```
In [10]: tiny_dataset = [list(range(7)), list(range(7, 10))]
         print('dataset', tiny_dataset)
         for center, context in zip(*get_centers_and_contexts(tiny_dataset, 2)):
             print('center', center, 'has contexts', context)
```

```
dataset [[0, 1, 2, 3, 4, 5, 6], [7, 8, 9]]
center 0 has contexts [1]
center 1 has contexts [0, 2]
center 2 has contexts [0, 1, 3, 4]
center 3 has contexts [1, 2, 4, 5]
center 4 has contexts [2, 3, 5, 6]
center 5 has contexts [3, 4, 6]
center 6 has contexts [4, 5]
center 7 has contexts [8, 9]
center 8 has contexts [7, 9]
center 9 has contexts [8]
```

In the experiment, we set the maximum context window size to 5. The following extracts all the central target words and their context words in the data set.

```
In [11]: all_centers, all_contexts = get_centers_and_contexts(subsampled_dataset, 5)
```

11.3.2 Negative Sampling

We use negative sampling for approximate training. For a central and context word pair, we randomly sample K noise words ($K = 5$ in the experiment). According to the suggestion in the Word2vec paper, the noise word sampling probability $\mathbb{P}(w)$ is the ratio of the word frequency of w to the total word frequency raised to the power of 0.75[2].

```
In [12]: def get_negatives(all_contexts, sampling_weights, K):
             all_negatives, neg_candidates, i = [], [], 0
             population = list(range(len(sampling_weights)))
             for contexts in all_contexts:
                 negatives = []
                 while len(negatives) < len(contexts) * K:
                     if i == len(neg_candidates):
                         # An index of k words is randomly generated as noise words
                         # based on the weight of each word (sampling_weights). For
                         # efficient calculation, k can be set slightly larger
                         i, neg_candidates = 0, random.choices(
                             population, sampling_weights, k=int(1e5))
                     neg, i = neg_candidates[i], i + 1
                     # Noise words cannot be context words
                     if neg not in set(contexts):
                         negatives.append(neg)
                 all_negatives.append(negatives)
             return all_negatives

         sampling_weights = [counter[w]**0.75 for w in idx_to_token]
         all_negatives = get_negatives(all_contexts, sampling_weights, 5)
```

11.3.3 Reading Data

We extract all central target words `all_centers`, and the context words `all_contexts` and noise words `all_negatives` of each central target word from the data set. We will read them in random mini-batches.

In a mini-batch of data, the i-th example includes a central word and its corresponding n_i context words and m_i noise words. Since the context window size of each example may be different, the sum of context words and noise words, $n_i + m_i$, will be different. When constructing a mini-batch, we concatenate the context words and noise words of each example, and add 0s for padding until the length of the concatenations are the same, that is, the length of all concatenations is $\max_i n_i + m_i$(max_len). In order to avoid the effect of padding on the loss function calculation, we construct the mask variable `masks`, each element of which corresponds to an element in the concatenation of context and noise words, `contexts_negatives`. When an element in the variable `contexts_negatives` is a padding, the element in the mask variable `masks` at the same position will be 0. Otherwise, it takes the value 1. In order to distinguish between positive and negative examples, we also need to distinguish the context words from the noise words in the `contexts_negatives` variable. Based on the construction of the mask variable, we only need to create a label variable `labels` with the same shape as the `contexts_negatives` variable and set the elements corresponding to context words (positive examples) to 1, and the rest to 0.

Next, we will implement the mini-batch reading function `batchify`. Its mini-batch input `data` is a list whose length is the batch size, each element of which contains central target words `center`, context words `context`, and noise words `negative`. The mini-batch data returned by this function conforms to the format we need, for example, it includes the mask variable.

```
In [13]: def batchify(data):
             max_len = max(len(c) + len(n) for _, c, n in data)
             centers, contexts_negatives, masks, labels = [], [], [], []
             for center, context, negative in data:
                 cur_len = len(context) + len(negative)
                 centers += [center]
                 contexts_negatives += [context + negative + [0] * (max_len - cur_len)]
                 masks += [[1] * cur_len + [0] * (max_len - cur_len)]
                 labels += [[1] * len(context) + [0] * (max_len - len(context))]
             return (nd.array(centers).reshape((-1, 1)), nd.array(contexts_negatives),
                     nd.array(masks), nd.array(labels))
```

We use the `batchify` function just defined to specify the mini-batch reading method in the `DataLoader` instance. Then, we print the shape of each variable in the first batch read.

```
In [14]: batch_size = 512
         num_workers = 0 if sys.platform.startswith('win32') else 4
         dataset = gdata.ArrayDataset(all_centers, all_contexts, all_negatives)
         data_iter = gdata.DataLoader(dataset, batch_size, shuffle=True,
                                      batchify_fn=batchify, num_workers=num_workers)
         for batch in data_iter:
             for name, data in zip(['centers', 'contexts_negatives', 'masks',
                                    'labels'], batch):
                 print(name, 'shape:', data.shape)
             break

centers shape: (512, 1)
contexts_negatives shape: (512, 60)
masks shape: (512, 60)
labels shape: (512, 60)
```

11.3.4 The Skip-Gram Model

We will implement the skip-gram model by using embedding layers and mini-batch multiplication. These methods are also often used to implement other natural language processing applications.

Embedding Layer

The layer in which the obtained word is embedded is called the embedding layer, which can be obtained by creating an `nn.Embedding` instance in Gluon. The weight of the embedding layer is a matrix whose number of rows is the dictionary size (`input_dim`) and whose number of columns is the dimension of each word vector (`output_dim`). We set the dictionary size to 20 and the word vector dimension to 4.

```
In [15]: embed = nn.Embedding(input_dim=20, output_dim=4)
         embed.initialize()
         embed.weight

Out[15]: Parameter embedding0_weight (shape=(20, 4), dtype=float32)
```

The input of the embedding layer is the index of the word. When we enter the index i of a word, the embedding layer returns the ith row of the weight matrix as its word vector. Below we enter an index of shape (2,3) into the embedding layer. Because the dimension of the word vector is 4, we obtain a word vector of shape (2,3,4).

```
In [16]: x = nd.array([[1, 2, 3], [4, 5, 6]])
         embed(x)

Out[16]:
         [[[ 0.01438687  0.05011239  0.00628365  0.04861524]
           [-0.01068833  0.01729892  0.02042518 -0.01618656]
           [-0.00873779 -0.02834515  0.05484822 -0.06206018]]

          [[ 0.06491279 -0.03182812 -0.01631819 -0.00312688]
           [ 0.0408415   0.04370362  0.00404529 -0.0028032 ]
           [ 0.00952624 -0.01501013  0.05958354  0.04705103]]]
         <NDArray 2x3x4 @cpu(0)>
```

Mini-batch Multiplication

We can multiply the matrices in two mini-batches one by one, by the mini-batch multiplication operation `batch_dot`. Suppose the first batch contains n matrices $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n$ with a shape of $a \times b$, and the second batch contains n matrices $\boldsymbol{Y}_1, \ldots, \boldsymbol{Y}_n$ with a shape of $b \times c$. The output of matrix multiplication on these two batches are n matrices $\boldsymbol{X}_1\boldsymbol{Y}_1, \ldots, \boldsymbol{X}_n\boldsymbol{Y}_n$ with a shape of $a \times c$. Therefore, given two NDArrays of shape (n, a, b) and (n, b, c), the shape of the mini-batch multiplication output is (n, a, c).

```
In [17]: X = nd.ones((2, 1, 4))
         Y = nd.ones((2, 4, 6))
         nd.batch_dot(X, Y).shape

Out[17]: (2, 1, 6)
```

Skip-gram Model Forward Calculation

In forward calculation, the input of the skip-gram model contains the central target word index `center` and the concatenated context and noise word index `contexts_and_negatives`. In which, the `center` variable has the shape (batch size, 1), while the `contexts_and_negatives` variable has the shape (batch size, `max_len`). These two variables are first transformed from word indexes to word vectors by the word embedding layer, and then the output of shape (batch size, 1, `max_len`) is obtained by mini-batch multiplication. Each element in the output is the inner product of the central target word vector and the context word vector or noise word vector.

```
In [18]: def skip_gram(center, contexts_and_negatives, embed_v, embed_u):
             v = embed_v(center)
             u = embed_u(contexts_and_negatives)
             pred = nd.batch_dot(v, u.swapaxes(1, 2))
             return pred
```

11.3.5 To train a model

Before training the word embedding model, we need to define the loss function of the model.

Binary Cross Entropy Loss Function

According to the definition of the loss function in negative sampling, we can directly use Gluon's binary cross entropy loss function `SigmoidBinaryCrossEntropyLoss`.

```
In [19]: loss = gloss.SigmoidBinaryCrossEntropyLoss()
```

It is worth mentioning that we can use the mask variable to specify the partial predicted value and label that participate in loss function calculation in the mini-batch: when the mask is 1, the predicted value and label of the corresponding position will participate in the calculation of the loss function; When the mask is 0, the predicted value and label of the corresponding position do not participate in the calculation of the loss function. As we mentioned earlier, mask variables can be used to avoid the effect of padding on loss function calculations.

```
In [20]: pred = nd.array([[1.5, 0.3, -1, 2], [1.1, -0.6, 2.2, 0.4]])
         # 1 and 0 in the label variables label represent context words and the noise
         # words, respectively
         label = nd.array([[1, 0, 0, 0], [1, 1, 0, 0]])
         mask = nd.array([[1, 1, 1, 1], [1, 1, 1, 0]])  # Mask variable
         loss(pred, label, mask) * mask.shape[1] / mask.sum(axis=1)
Out[20]:
         [0.8739896 1.2099689]
         <NDArray 2 @cpu(0)>
```

Next, as a comparison, we will implement binary cross-entropy loss function calculation from scratch and calculate the predicted value with a mask of 1 and the loss of the label based on the mask variable `mask`.

```
In [21]: def sigmd(x):
             return -math.log(1 / (1 + math.exp(-x)))

         print('%.7f' % ((sigmd(1.5) + sigmd(-0.3) + sigmd(1) + sigmd(-2)) / 4))
         print('%.7f' % ((sigmd(1.1) + sigmd(-0.6) + sigmd(-2.2)) / 3))

0.8739896
1.2099689
```

Initialize Model Parameters

We construct the embedding layers of the central and context words, respectively, and set the hyper-parameter word vector dimension embed_size to 100.

```
In [22]: embed_size = 100
         net = nn.Sequential()
         net.add(nn.Embedding(input_dim=len(idx_to_token), output_dim=embed_size),
                 nn.Embedding(input_dim=len(idx_to_token), output_dim=embed_size))
```

Training

The training function is defined below. Because of the existence of padding, the calculation of the loss function is slightly different compared to the previous training functions.

```
In [23]: def train(net, lr, num_epochs):
             ctx = d2l.try_gpu()
             net.initialize(ctx=ctx, force_reinit=True)
             trainer = gluon.Trainer(net.collect_params(), 'adam',
                                     {'learning_rate': lr})
             for epoch in range(num_epochs):
                 start, l_sum, n = time.time(), 0.0, 0
                 for batch in data_iter:
                     center, context_negative, mask, label = [
                         data.as_in_context(ctx) for data in batch]
                     with autograd.record():
                         pred = skip_gram(center, context_negative, net[0], net[1])
                         # Use the mask variable to avoid the effect of padding on loss
                         # function calculations
                         l = (loss(pred.reshape(label.shape), label, mask) *
                             mask.shape[1] / mask.sum(axis=1))
                     l.backward()
                     trainer.step(batch_size)
                     l_sum += l.sum().asscalar()
                     n += l.size
                 print('epoch %d, loss %.2f, time %.2fs'
                       % (epoch + 1, l_sum / n, time.time() - start))
```

Now, we can train a skip-gram model using negative sampling.

```
In [24]: train(net, 0.005, 3)

epoch 1, loss 0.46, time 30.94s
epoch 2, loss 0.39, time 31.82s
epoch 3, loss 0.35, time 30.66s
```

11.3.6 Applying the Word Embedding Model

After training the word embedding model, we can represent similarity in meaning between words based on the cosine similarity of two word vectors. As we can see, when using the trained word embedding model, the words closest in meaning to the word chip are mostly related to chips.

```
In [25]: def get_similar_tokens(query_token, k, embed):
             W = embed.weight.data()
             x = W[token_to_idx[query_token]]
             # The added 1e-9 is for numerical stability
             cos = nd.dot(W, x) / (nd.sum(W * W, axis=1) * nd.sum(x * x) + 1e-9).sqrt()
             topk = nd.topk(cos, k=k+1, ret_typ='indices').asnumpy().astype('int32')
             for i in topk[1:]:   # Remove the input words
                 print('cosine sim=%.3f: %s' % (cos[i].asscalar(), (idx_to_token[i])))

         get_similar_tokens('chip', 3, net[0])

cosine sim=0.580: microprocessors
cosine sim=0.532: intel
cosine sim=0.512: microprocessor
```

Summary

- We can use Gluon to train a skip-gram model through negative sampling.

- Subsampling attempts to minimize the impact of high-frequency words on the training of a word embedding model.

- We can pad examples of different lengths to create mini-batches with examples of all the same length and use mask variables to distinguish between padding and non-padding elements, so that only non-padding elements participate in the calculation of the loss function.

Exercises

- Set `sparse_grad=True` when creating an instance of `nn.Embedding`. Does it accelerate training? Look up MXNet documentation to learn the meaning of this argument.

- We use the `batchify` function to specify the mini-batch reading method in the `DataLoader` instance and print the shape of each variable in the first batch read. How should these shapes be calculated?

- Try to find synonyms for other words.

- Tune the hyper-parameters and observe and analyze the experimental results.

- When the data set is large, we usually sample the context words and the noise words for the central target word in the current mini-batch only when updating the model parameters. In other words, the same central target word may have different context words or noise words in different epochs. What are the benefits of this sort of training? Try to implement this training method.

Reference

[1] Penn Tree Bank. https://catalog.ldc.upenn.edu/LDC99T42

[2] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. In Advances in neural information processing systems (pp. 3111-3119).

Scan the QR Code to Discuss

11.4 Subword Embedding (fastText)

English words usually have internal structures and formation methods. For example, we can deduce the relationship between dog, dogs, and dogcatcher by their spelling. All these words have the same root, dog, but they use different suffixes to change the meaning of the word. Moreover, this association can be extended to other words. For example, the relationship between dog and dogs is just like the relationship between cat and cats. The relationship between boy and boyfriend is just like the relationship between girl and girlfriend. This characteristic is not unique to English. In French and Spanish, a lot of verbs can have more than 40 different forms depending on the context. In Finnish, a noun may have more than 15 forms. In fact, morphology, which is an important branch of linguistics, studies the internal structure and formation of words.

In word2vec, we did not directly use morphology information. In both the skip-gram model and continuous bag-of-words model, we use different vectors to represent words with different forms. For example, dog and dogs are represented by two different vectors, while the relationship between these two vectors is not directly represented in the model. In view of this, fastText proposes the method of subword embedding, thereby attempting to introduce morphological information in the skip-gram model in word2vec[1].

In fastText, each central word is represented as a collection of subwords. Below we use the word where as an example to understand how subwords are formed. First, we add the special characters < and > at the beginning and end of the word to distinguish the subwords used as prefixes and suffixes. Then, we treat the word as a sequence of characters to extract the n-grams. For example, when $n = 3$, we can get all subwords with a length of 3:

$$\text{"<wh", "whe", "her", "ere", "re>",}$$

and the special subword "<where>".

In fastText, for a word w, we record the union of all its subwords with length of 3 to 6 and special subwords as $\mathcal{G}_w$. Thus, the dictionary is the union of the collection of subwords of all words. Assume the vector of the subword g in the dictionary is $\boldsymbol{z}_g$. Then, the central word vector $\boldsymbol{v}_w$ for the word w in the skip-gram model can be expressed as

$$\boldsymbol{v}_w = \sum_{g \in \mathcal{G}_w} \boldsymbol{z}_g.$$

The rest of the fastText process is consistent with the skip-gram model, so it is not repeated here. As we can see, compared with the skip-gram model, the dictionary in fastText is larger, resulting in more model parameters. Also, the vector of one word requires the summation of all subword vectors, which results in higher computation complexity. However, we can obtain better vectors for more uncommon complex words, even words not existing in the dictionary, by looking at other words with similar structures.

Summary

- FastText proposes a subword embedding method. Based on the skip-gram model in word2vec, it represents the central word vector as the sum of the subword vectors of the word.

- Subword embedding utilizes the principles of morphology, which usually improves the quality of representations of uncommon words.

Exercises

- When there are too many subwords (for example, 6 words in English result in about 3×10^8 combinations), what problems arise? Can you think of any methods to solve them? Hint: Refer to the end of section 3.2 of the fastText paper[1].

- How can you design a subword embedding model based on the continuous bag-of-words model?

Reference

[1] Bojanowski, P., Grave, E., Joulin, A., & Mikolov, T. (2016). Enriching word vectors with subword information. arXiv preprint arXiv:1607.04606.

Scan the QR Code to Discuss

11.5 Word Embedding with Global Vectors (GloVe)

First, we should review the skip-gram model in word2vec. The conditional probability $\mathbb{P}(w_j \mid w_i)$ expressed in the skip-gram model using the softmax operation will be recorded as q_{ij}, that is:

$$q_{ij} = \frac{\exp(\mathbf{u}_j^\top \mathbf{v}_i)}{\sum_{k \in \mathcal{V}} \exp(\mathbf{u}_k^\top \mathbf{v}_i)},$$

where $\mathbf{v}_i$ and $\mathbf{u}_i$ are the vector representations of word w_i of index i as the center word and context word respectively, and $\mathcal{V} = \{0, 1, \ldots, |\mathcal{V}| - 1\}$ is the vocabulary index set.

For word w_i, it may appear in the data set for multiple times. We collect all the context words every time when w_i is a center word and keep duplicates, denoted as multiset $\mathcal{C}_i$. The number of an element in a multiset is called the multiplicity of the element. For instance, suppose that word w_i appears twice in the data set: the context windows when these two w_i become center words in the text sequence contain context word indices $2, 1, 5, 2$ and $2, 3, 2, 1$. Then, multiset $\mathcal{C}_i = \{1, 1, 2, 2, 2, 2, 3, 5\}$, where multiplicity of element 1 is 2, multiplicity of element 2 is 4, and multiplicities of elements 3 and 5 are both 1. Denote multiplicity of element j in multiset $\mathcal{C}_i$ as x_{ij}: it is the number of word w_j in all the context windows for center word w_i in the entire data set. As a result, the loss function of the skip-gram model can be expressed in a different way:

$$-\sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{V}} x_{ij} \log q_{ij}.$$

We add up the number of all the context words for the central target word w_i to get x_i, and record the conditional probability x_{ij}/x_i for generating context word w_j based on central target word w_i as p_{ij}. We can rewrite the loss function of the skip-gram model as

$$-\sum_{i \in \mathcal{V}} x_i \sum_{j \in \mathcal{V}} p_{ij} \log q_{ij}.$$

In the formula above, $\sum_{j \in \mathcal{V}} p_{ij} \log q_{ij}$ computes the conditional probability distribution p_{ij} for context word generation based on the central target word w_i and the cross-entropy of conditional probability distribution q_{ij} predicted by the model. The loss function is weighted using the sum of the number of context words with the central target word w_i. If we minimize the loss function from the formula above, we will be able to allow the predicted conditional probability distribution to approach as close as possible to the true conditional probability distribution.

However, although the most common type of loss function, the cross-entropy loss function is sometimes not a good choice. On the one hand, as we mentioned in the *Approximate Training* section, the cost of letting the model prediction q_{ij} become the legal probability distribution has the sum of all items in the entire dictionary in its denominator. This can easily lead to excessive computational overhead. On the other hand, there are often a lot of uncommon words in the dictionary, and they appear rarely in the data set. In the cross-entropy loss function, the final prediction of the conditional probability distribution on a large number of uncommon words is likely to be inaccurate.

11.5.1 The GloVe Model

To address this, GloVe, a word embedding model that came after word2vec, adopts square loss and makes three changes to the skip-gram model based on this loss[1].

1. Here, we use the non-probability distribution variables $p'_{ij} = x_{ij}$ and $q'_{ij} = \exp(\mathbf{u}_j^\top \mathbf{v}_i)$ and take their logs. Therefore, we get the square loss $\left(\log p'_{ij} - \log q'_{ij} \right)^2 = \left(\mathbf{u}_j^\top \mathbf{v}_i - \log x_{ij} \right)^2$.

2. We add two scalar model parameters for each word w_i: the bias terms b_i (for central target words) and c_i(for context words).

3. Replace the weight of each loss with the function $h(x_{ij})$. The weight function $h(x)$ is a monotone increasing function with the range $[0,1]$.

Therefore, the goal of GloVe is to minimize the loss function.

$$\sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{V}} h(x_{ij}) \left(\mathbf{u}_j^\top \mathbf{v}_i + b_i + c_j - \log x_{ij} \right)^2 .$$

Here, we have a suggestion for the choice of weight function $h(x)$: when $x < c$ (e.g $c = 100$), make $h(x) = (x/c)^\alpha$ (e.g $\alpha = 0.75$), otherwise make $h(x) = 1$. Because $h(0) = 0$, the squared loss term for $x_{ij} = 0$ can be simply ignored. When we use mini-batch SGD for training, we conduct random sampling to get a non-zero mini-batch x_{ij} from each time step and compute the gradient to update the model parameters. These non-zero x_{ij} are computed in advance based on the entire data set and they contain global statistics for the data set. Therefore, the name GloVe is taken from Global Vectors.

Notice that if word w_i appears in the context window of word w_j, then word w_j will also appear in the context window of word w_i. Therefore, $x_{ij} = x_{ji}$. Unlike word2vec, GloVe fits the symmetric $\log x_{ij}$ in lieu of the asymmetric conditional probability p_{ij}. Therefore, the central target word vector and context word vector of any word are equivalent in GloVe. However, the two sets of word vectors that are learned by the same word may be different in the end due to different initialization values. After learning all the word vectors, GloVe will use the sum of the central target word vector and the context word vector as the final word vector for the word.

11.5.2 Understanding GloVe from Conditional Probability Ratios

We can also try to understand GloVe word embedding from another perspective. We will continue the use of symbols from earlier in this section, $\mathbb{P}(w_j \mid w_i)$ represents the conditional probability of generating context word w_j with central target word w_i in the data set, and it will be recorded as p_{ij}. From a real example from a large corpus, here we have the following two sets of conditional probabilities with ice and steam as the central target words and the ratio between them[1]:

$w_k =$	solid	gas	water	fashion
$p_1 = \mathbb{P}(w_k \mid$ ice $)$	0.00019	0.000066	0.003	0.000017
$p_2 = \mathbb{P}(w_k \mid$ steam $)$	0.000022	0.00078	0.0022	0.000018
p_1/p_2	8.9	0.085	1.36	0.96

We will be able to observe phenomena such as:

- For a word w_k that is related to ice but not to steam, such as w_k =solid, we would expect a larger conditional probability ratio, like the value 8.9 in the last row of the table above.

- For a word w_k that is related to steam but not to ice, such as w_k =gas, we would expect a smaller conditional probability ratio, like the value 0.085 in the last row of the table above.

- For a word w_k that is related to both ice and steam, such as w_k =water, we would expect a conditional probability ratio close to 1, like the value 1.36 in the last row of the table above.

- For a word w_k that is related to neither ice or steam, such as w_k =fashion, we would expect a conditional probability ratio close to 1, like the value 0.96 in the last row of the table above.

We can see that the conditional probability ratio can represent the relationship between different words more intuitively. We can construct a word vector function to fit the conditional probability ratio more effectively. As we know, to obtain any ratio of this type requires three words w_i, w_j, and w_k. The conditional probability ratio with w_i as the central target word is p_{ij}/p_{ik}. We can find a function that uses word vectors to fit this conditional probability ratio.

$$f(\boldsymbol{u}_j, \boldsymbol{u}_k, \boldsymbol{v}_i) \approx \frac{p_{ij}}{p_{ik}}.$$

The possible design of function f here will not be unique. We only need to consider a more reasonable possibility. Notice that the conditional probability ratio is a scalar, we can limit f to be a scalar function: $f(\boldsymbol{u}_j, \boldsymbol{u}_k, \boldsymbol{v}_i) = f\left((\boldsymbol{u}_j - \boldsymbol{u}_k)^\top \boldsymbol{v}_i\right)$. After exchanging index j with k, we will be able to see that function f satisfies the condition $f(x)f(-x) = 1$, so one possibility could be $f(x) = \exp(x)$. Thus:

$$f(\boldsymbol{u}_j, \boldsymbol{u}_k, \boldsymbol{v}_i) = \frac{\exp\left(\boldsymbol{u}_j^\top \boldsymbol{v}_i\right)}{\exp\left(\boldsymbol{u}_k^\top \boldsymbol{v}_i\right)} \approx \frac{p_{ij}}{p_{ik}}.$$

One possibility that satisfies the right side of the approximation sign is $\exp\left(\boldsymbol{u}_j^\top \boldsymbol{v}_i\right) \approx \alpha p_{ij}$, where α is a constant. Considering that $p_{ij} = x_{ij}/x_i$, after taking the logarithm we get $\boldsymbol{u}_j^\top \boldsymbol{v}_i \approx \log \alpha + \log x_{ij} - \log x_i$. We use additional bias terms to fit $-\log \alpha + \log x_i$, such as the central target word bias term b_i and context word bias term c_j:

$$\boldsymbol{u}_j^\top \boldsymbol{v}_i + b_i + c_j \approx \log(x_{ij}).$$

By taking the square error and weighting the left and right sides of the formula above, we can get the loss function of GloVe.

Summary

- In some cases, the cross-entropy loss function may have a disadvantage. GloVe uses squared loss and the word vector to fit global statistics computed in advance based on the entire data set.

- The central target word vector and context word vector of any word are equivalent in GloVe.

Exercises

- If a word appears in the context window of another word, how can we use the distance between them in the text sequence to redesign the method for computing the conditional probability p_{ij}? Hint: See section 4.2 from the paper GloVe[1].

- For any word, will its central target word bias term and context word bias term be equivalent to each other in GloVe? Why?

Reference

[1] Pennington, J., Socher, R., & Manning, C. (2014). Glove: Global vectors for word representation. In Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP) (pp. 1532-1543).

Scan the QR Code to Discuss

11.6 Finding Synonyms and Analogies

In the *Implementation of Word2vec* section, we trained a word2vec word embedding model on a small-scale data set and searched for synonyms using the cosine similarity of word vectors. In practice, word vectors pre-trained on a large-scale corpus can often be applied to downstream natural language processing tasks. This section will demonstrate how to use these pre-trained word vectors to find synonyms and analogies. We will continue to apply pre-trained word vectors in subsequent sections.

11.6.1 Using Pre-trained Word Vectors

MXNet's `contrib.text` package provides functions and classes related to natural language processing (see the GluonNLP tool package[1] for more details). Next, let us check out names of the provided pre-trained word embeddings.

```
In [1]: from mxnet import nd
        from mxnet.contrib import text

        text.embedding.get_pretrained_file_names().keys()
```

```
Out[1]: dict_keys(['glove', 'fasttext'])
```

Given the name of the word embedding, we can see which pre-trained models are provided by the word embedding. The word vector dimensions of each model may be different or obtained by pre-training on different data sets.

```
In [2]: print(text.embedding.get_pretrained_file_names('glove'))

['glove.42B.300d.txt', 'glove.6B.50d.txt', 'glove.6B.100d.txt', 'glove.6B.200d.txt',
↪  'glove.6B.300d.txt', 'glove.840B.300d.txt', 'glove.twitter.27B.25d.txt',
↪  'glove.twitter.27B.50d.txt', 'glove.twitter.27B.100d.txt',
↪  'glove.twitter.27B.200d.txt']
```

The general naming conventions for pre-trained GloVe models are model.(data set.)number of words in data set.word vector dimension.txt. For more information, please refer to the GloVe and fastText project sites [2,3]. Below, we use a 50-dimensional GloVe word vector based on Wikipedia subset pre-training. The corresponding word vector is automatically downloaded the first time we create a pre-trained word vector instance.

```
In [3]: glove_6b50d = text.embedding.create(
            'glove', pretrained_file_name='glove.6B.50d.txt')
```

Print the dictionary size. The dictionary contains 400,000 words and a special unknown token.

```
In [4]: len(glove_6b50d)

Out[4]: 400001
```

We can use a word to get its index in the dictionary, or we can get the word from its index.

```
In [5]: glove_6b50d.token_to_idx['beautiful'], glove_6b50d.idx_to_token[3367]

Out[5]: (3367, 'beautiful')
```

11.6.2 Applying Pre-trained Word Vectors

Below, we demonstrate the application of pre-trained word vectors, using GloVe as an example.

Finding Synonyms

Here, we re-implement the algorithm used to search for synonyms by cosine similarity introduced in the *Implementation of Word2vec* section. In order to reuse the logic for seeking the k nearest neighbors when seeking analogies, we encapsulate this part of the logic separately in the knn (k-nearest neighbors) function.

```
In [6]: def knn(W, x, k):
            # The added 1e-9 is for numerical stability
            cos = nd.dot(W, x.reshape((-1,))) / (
                (nd.sum(W * W, axis=1) + 1e-9).sqrt() * nd.sum(x * x).sqrt())
            topk = nd.topk(cos, k=k, ret_typ='indices').asnumpy().astype('int32')
            return topk, [cos[i].asscalar() for i in topk]
```

Then, we search for synonyms by pre-training the word vector instance embed.

```
In [7]: def get_similar_tokens(query_token, k, embed):
            topk, cos = knn(embed.idx_to_vec,
                            embed.get_vecs_by_tokens([query_token]), k+1)
            for i, c in zip(topk[1:], cos[1:]):  # Remove input words
                print('cosine sim=%.3f: %s' % (c, (embed.idx_to_token[i])))
```

The dictionary of pre-trained word vector instance `glove_6b50d` already created contains 400,000 words and a special unknown token. Excluding input words and unknown words, we search for the three words that are the most similar in meaning to chip.

```
In [8]: get_similar_tokens('chip', 3, glove_6b50d)

cosine sim=0.856: chips
cosine sim=0.749: intel
cosine sim=0.749: electronics
```

Next, we search for the synonyms of baby and beautiful.

```
In [9]: get_similar_tokens('baby', 3, glove_6b50d)

cosine sim=0.839: babies
cosine sim=0.800: boy
cosine sim=0.792: girl

In [10]: get_similar_tokens('beautiful', 3, glove_6b50d)

cosine sim=0.921: lovely
cosine sim=0.893: gorgeous
cosine sim=0.830: wonderful
```

Finding Analogies

In addition to seeking synonyms, we can also use the pre-trained word vector to seek the analogies between words. For example, man:woman::son:daughter is an example of analogy, man is to woman as son is to daughter. The problem of seeking analogies can be defined as follows: for four words in the analogical relationship $a : b :: c : d$, given the first three words, a, b and c, we want to find d. Assume the word vector for the word w is $\text{vec}(w)$. To solve the analogy problem, we need to find the word vector that is most similar to the result vector of $\text{vec}(c) + \text{vec}(b) - \text{vec}(a)$.

```
In [11]: def get_analogy(token_a, token_b, token_c, embed):
             vecs = embed.get_vecs_by_tokens([token_a, token_b, token_c])
             x = vecs[1] - vecs[0] + vecs[2]
             topk, cos = knn(embed.idx_to_vec, x, 1)
             return embed.idx_to_token[topk[0]]  # Remove unknown words
```

Verify the male-female analogy.

```
In [12]: get_analogy('man', 'woman', 'son', glove_6b50d)

Out[12]: 'daughter'
```

Capital-country analogy: beijing is to china as tokyo is to what? The answer should be japan.

```
In [13]: get_analogy('beijing', 'china', 'tokyo', glove_6b50d)

Out[13]: 'japan'
```

Adjective-superlative adjective analogy: bad is to worst as big is to what? The answer should be biggest.

```
In [14]: get_analogy('bad', 'worst', 'big', glove_6b50d)
Out[14]: 'biggest'
```

Present tense verb-past tense verb analogy: do is to did as go is to what? The answer should be went.

```
In [15]: get_analogy('do', 'did', 'go', glove_6b50d)
Out[15]: 'went'
```

Summary

- Word vectors pre-trained on a large-scale corpus can often be applied to downstream natural language processing tasks.
- We can use pre-trained word vectors to seek synonyms and analogies.

Exercises

- Test the fastText results.
- If the dictionary is extremely large, how can we accelerate finding synonyms and analogies?

Reference

[1] GluonNLP tool package. https://gluon-nlp.mxnet.io/

[2] GloVe project website. https://nlp.stanford.edu/projects/glove/

[3] fastText project website. https://fasttext.cc/

Scan the QR Code to Discuss

11.7 Text Sentiment Classification: Using Recurrent Neural Networks

Text classification is a common task in natural language processing, which transforms a sequence of text of indefinite length into a category of text. This section will focus on one of the sub-questions in this field: using text sentiment classification to analyze the emotions of the text's author. This problem is also called sentiment analysis and has a wide range of applications. For example, we can analyze user reviews of products to obtain user satisfaction statistics, or analyze user sentiments about market conditions and use it to predict future trends.

Similar to search synonyms and analogies, text classification is also a downstream application of word embedding. In this section, we will apply pre-trained word vectors and bidirectional recurrent neural networks with multiple hidden layers. We will use them to determine whether a text sequence of indefinite length contains positive or negative emotion. Import the required package or module before starting the experiment.

```
In [1]: import sys
        sys.path.insert(0, '..')

        import collections
        import d2l
        from mxnet import gluon, init, nd
        from mxnet.contrib import text
        from mxnet.gluon import data as gdata, loss as gloss, nn, rnn, utils as gutils
        import os
        import random
        import tarfile
```

11.7.1 Text Sentiment Classification Data

We use Stanford's Large Movie Review Dataset as the data set for text sentiment classification[1]. This data set is divided into two data sets for training and testing purposes, each containing 25,000 movie reviews downloaded from IMDb. In each data set, the number of comments labeled as positive and negative is equal.

Reading Data

We first download this data set to the ../data path and extract it to ../data/aclImdb.

```
In [2]: # This function has been saved in the d2l package for future use
        def download_imdb(data_dir='../data'):
            url = ('http://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz')
            sha1 = '01ada507287d82875905620988597833ad4e0903'
            fname = gutils.download(url, data_dir, sha1_hash=sha1)
            with tarfile.open(fname, 'r') as f:
                f.extractall(data_dir)

        download_imdb()
```

Next, read the training and test data sets. Each example is a review and its corresponding label: 1 indicates positive and 0 indicates negative.

```
In [3]: # This function has been saved in the d2l package for future use
        def read_imdb(folder='train'):
            data = []
            for label in ['pos', 'neg']:
                folder_name = os.path.join('../data/aclImdb/', folder, label)
                for file in os.listdir(folder_name):
                    with open(os.path.join(folder_name, file), 'rb') as f:
                        review = f.read().decode('utf-8').replace('\n', '').lower()
                        data.append([review, 1 if label == 'pos' else 0])
            random.shuffle(data)
            return data

        train_data, test_data = read_imdb('train'), read_imdb('test')
```

Data Preprocessing

We need to segment each review to get a review with segmented words. The `get_tokenized_imdb` function defined here uses the easiest method: word tokenization based on spaces.

```
In [4]: # This function has been saved in the d2l package for future use
        def get_tokenized_imdb(data):
            def tokenizer(text):
                return [tok.lower() for tok in text.split(' ')]
            return [tokenizer(review) for review, _ in data]
```

Now, we can create a dictionary based on the training data set with the words segmented. Here, we have filtered out words that appear less than 5 times.

```
In [5]: # This function has been saved in the d2l package for future use
        def get_vocab_imdb(data):
            tokenized_data = get_tokenized_imdb(data)
            counter = collections.Counter([tk for st in tokenized_data for tk in st])
            return text.vocab.Vocabulary(counter, min_freq=5)

        vocab = get_vocab_imdb(train_data)
        '# Words in vocab:', len(vocab)
Out[5]: ('# Words in vocab:', 46151)
```

Because the reviews have different lengths, so they cannot be directly combined into mini-batches, we define the `preprocess_imdb` function to segment each comment, convert it into a word index through a dictionary, and then fix the length of each comment to 500 by truncating or adding 0s.

```
In [6]: # This function has been saved in the d2l package for future use
        def preprocess_imdb(data, vocab):
            # Make the length of each comment 500 by truncating or adding 0s
            max_l = 500

            def pad(x):
                return x[:max_l] if len(x) > max_l else x + [0] * (max_l - len(x))

            tokenized_data = get_tokenized_imdb(data)
```

```
        features = nd.array([pad(vocab.to_indices(x)) for x in tokenized_data])
        labels = nd.array([score for _, score in data])
        return features, labels
```

Create Data Iterator

Now, we will create a data iterator. Each iteration will return a mini-batch of data.

```
In [7]: batch_size = 64
        train_set = gdata.ArrayDataset(*preprocess_imdb(train_data, vocab))
        test_set = gdata.ArrayDataset(*preprocess_imdb(test_data, vocab))
        train_iter = gdata.DataLoader(train_set, batch_size, shuffle=True)
        test_iter = gdata.DataLoader(test_set, batch_size)
```

Print the shape of the first mini-batch of data and the number of mini-batches in the training set.

```
In [8]: for X, y in train_iter:
            print('X', X.shape, 'y', y.shape)
            break
        '#batches:', len(train_iter)

X (64, 500) y (64,)

Out[8]: ('#batches:', 391)
```

11.7.2 Use a Recurrent Neural Network Model

In this model, each word first obtains a feature vector from the embedding layer. Then, we further encode the feature sequence using a bidirectional recurrent neural network to obtain sequence information. Finally, we transform the encoded sequence information to output through the fully connected layer. Specifically, we can concatenate hidden states of bidirectional long-short term memory in the initial time step and final time step and pass it to the output layer classification as encoded feature sequence information. In the `BiRNN` class implemented below, the `Embedding` instance is the embedding layer, the `LSTM` instance is the hidden layer for sequence encoding, and the `Dense` instance is the output layer for generated classification results.

```
In [9]: class BiRNN(nn.Block):
            def __init__(self, vocab, embed_size, num_hiddens, num_layers, **kwargs):
                super(BiRNN, self).__init__(**kwargs)
                self.embedding = nn.Embedding(len(vocab), embed_size)
                # Set Bidirectional to True to get a bidirectional recurrent neural
                # network
                self.encoder = rnn.LSTM(num_hiddens, num_layers=num_layers,
                                        bidirectional=True, input_size=embed_size)
                self.decoder = nn.Dense(2)

            def forward(self, inputs):
                # The shape of inputs is (batch size, number of words). Because LSTM
                # needs to use sequence as the first dimension, the input is
                # transformed and the word feature is then extracted. The output shape
                # is (number of words, batch size, word vector dimension).
                embeddings = self.embedding(inputs.T)
```

```
        # The shape of states is (number of words, batch size, 2 * number of
        # hidden units).
        states = self.encoder(embeddings)
        # Concatenate the hidden states of the initial time step and final
        # time step to use as the input of the fully connected layer. Its
        # shape is (batch size, 4 * number of hidden units)
        encoding = nd.concat(states[0], states[-1])
        outputs = self.decoder(encoding)
        return outputs
```

Create a bidirectional recurrent neural network with two hidden layers.

```
In [10]: embed_size, num_hiddens, num_layers, ctx = 100, 100, 2, d2l.try_all_gpus()
         net = BiRNN(vocab, embed_size, num_hiddens, num_layers)
         net.initialize(init.Xavier(), ctx=ctx)
```

Load Pre-trained Word Vectors

Because the training data set for sentiment classification is not very large, in order to deal with overfitting, we will directly use word vectors pre-trained on a larger corpus as the feature vectors of all words. Here, we load a 100-dimensional GloVe word vector for each word in the dictionary `vocab`.

```
In [11]: glove_embedding = text.embedding.create(
             'glove', pretrained_file_name='glove.6B.100d.txt', vocabulary=vocab)
```

Then, we will use these word vectors as feature vectors for each word in the reviews. Note that the dimensions of the pre-trained word vectors need to be consistent with the embedding layer output size `embed_size` in the created model. In addition, we no longer update these word vectors during training.

```
In [12]: net.embedding.weight.set_data(glove_embedding.idx_to_vec)
         net.embedding.collect_params().setattr('grad_req', 'null')
```

Train and Evaluate the Model

Now, we can start training.

```
In [13]: lr, num_epochs = 0.01, 5
         trainer = gluon.Trainer(net.collect_params(), 'adam', {'learning_rate': lr})
         loss = gloss.SoftmaxCrossEntropyLoss()
         d2l.train(train_iter, test_iter, net, loss, trainer, ctx, num_epochs)

training on [gpu(0), gpu(1)]
epoch 1, loss 0.6221, train acc 0.625, test acc 0.811, time 54.1 sec
epoch 2, loss 0.4204, train acc 0.812, test acc 0.826, time 53.8 sec
epoch 3, loss 0.3818, train acc 0.834, test acc 0.832, time 53.7 sec
epoch 4, loss 0.3494, train acc 0.851, test acc 0.838, time 53.6 sec
epoch 5, loss 0.3236, train acc 0.861, test acc 0.839, time 53.6 sec
```

Finally, define the prediction function.

```
In [14]: # This function has been saved in the d2l package for future use
         def predict_sentiment(net, vocab, sentence):
             sentence = nd.array(vocab.to_indices(sentence), ctx=d2l.try_gpu())
```

```
                    label = nd.argmax(net(sentence.reshape((1, -1))), axis=1)
                    return 'positive' if label.asscalar() == 1 else 'negative'
```

Then, use the trained model to classify the sentiments of two simple sentences.

```
In [15]: predict_sentiment(net, vocab, ['this', 'movie', 'is', 'so', 'great'])

Out[15]: 'positive'

In [16]: predict_sentiment(net, vocab, ['this', 'movie', 'is', 'so', 'bad'])

Out[16]: 'negative'
```

Summary

- Text classification transforms a sequence of text of indefinite length into a category of text. This is a downstream application of word embedding.

- We can apply pre-trained word vectors and recurrent neural networks to classify the emotions in a text.

Exercises

- Increase the number of epochs. What accuracy rate can you achieve on the training and testing data sets? What about trying to re-tune other hyper-parameters?

- Will using larger pre-trained word vectors, such as 300-dimensional GloVe word vectors, improve classification accuracy?

- Can we improve the classification accuracy by using the spaCy word tokenization tool? You need to install spaCy: `pip install spacy` and install the English package: `python -m spacy download en`. In the code, first import spacy: `import spacy`. Then, load the spacy English package: `spacy_en = spacy.load('en')`. Finally, define the function `def tokenizer(text): return [tok.text for tok in spacy_en.tokenizer(text)]` and replace the original `tokenizer` function. It should be noted that GloVe's word vector uses - to connect each word when storing noun phrases. For example, the phrase new york is represented as new-york in GloVe. After using spaCy tokenization, new york may be stored as new york.

Reference

[1] Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., & Potts, C. (2011, June). Learning word vectors for sentiment analysis. In Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies-volume 1 (pp. 142-150). Association for Computational Linguistics.

11.8 Text Sentiment Classification: Using Convolutional Neural Networks (textCNN)

In the Convolutional Neural Networks chapter, we explored how to process two-dimensional image data with two-dimensional convolutional neural networks. In the previous language models and text classification tasks, we treated text data as a time series with only one dimension, and naturally, we used recurrent neural networks to process such data. In fact, we can also treat text as a one-dimensional image, so that we can use one-dimensional convolutional neural networks to capture associations between adjacent words. This section describes a groundbreaking approach to applying convolutional neural networks to text analysis: textCNN[1]. First, import the packages and modules required for the experiment.

```
In [1]: import sys
        sys.path.insert(0, '..')

        import d2l
        from mxnet import gluon, init, nd
        from mxnet.contrib import text
        from mxnet.gluon import data as gdata, loss as gloss, nn
```

11.8.1 One-dimensional Convolutional Layer

Before introducing the model, let us explain how a one-dimensional convolutional layer works. Like a two-dimensional convolutional layer, a one-dimensional convolutional layer uses a one-dimensional cross-correlation operation. In the one-dimensional cross-correlation operation, the convolution window starts from the leftmost side of the input array and slides on the input array from left to right successively. When the convolution window slides to a certain position, the input subarray in the window and kernel array are multiplied and summed by element to get the element at the corresponding location in the output array. As shown in Figure 10.4, the input is a one-dimensional array with a width of 7 and the width of the kernel array is 2. As we can see, the output width is $7 - 2 + 1 = 6$ and the first element is obtained by performing multiplication by element on the leftmost input subarray with a width of 2 and kernel array and then summing the results.
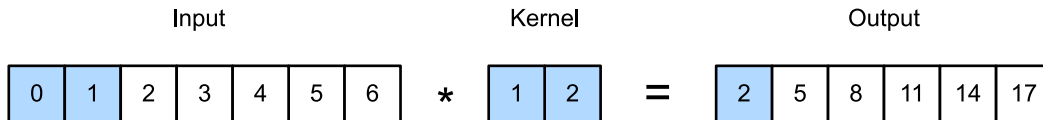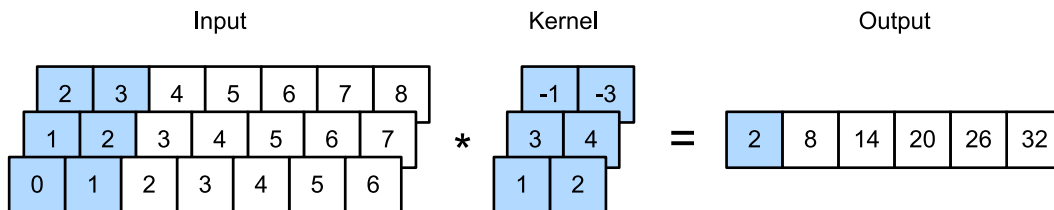
Fig. 11.4: One-dimensional cross-correlation operation. The shaded parts are the first output element as well as the input and kernel array elements used in its calculation: $0 \times 1 + 1 \times 2 = 2$.

Next, we implement one-dimensional cross-correlation in the `corr1d` function. It accepts the input array `X` and kernel array `K` and outputs the array `Y`.

```
In [2]: def corr1d(X, K):
            w = K.shape[0]
            Y = nd.zeros((X.shape[0] - w + 1))
            for i in range(Y.shape[0]):
                Y[i] = (X[i: i + w] * K).sum()
            return Y
```

Now, we will reproduce the results of the one-dimensional cross-correlation operation in Figure 10.4.

```
In [3]: X, K = nd.array([0, 1, 2, 3, 4, 5, 6]), nd.array([1, 2])
        corr1d(X, K)

Out[3]:
        [ 2.  5.  8. 11. 14. 17.]
        <NDArray 6 @cpu(0)>
```

The one-dimensional cross-correlation operation for multiple input channels is also similar to the two-dimensional cross-correlation operation for multiple input channels. On each channel, it performs the one-dimensional cross-correlation operation on the kernel and its corresponding input and adds the results of the channels to get the output. Figure 10.5 shows a one-dimensional cross-correlation operation with three input channels.
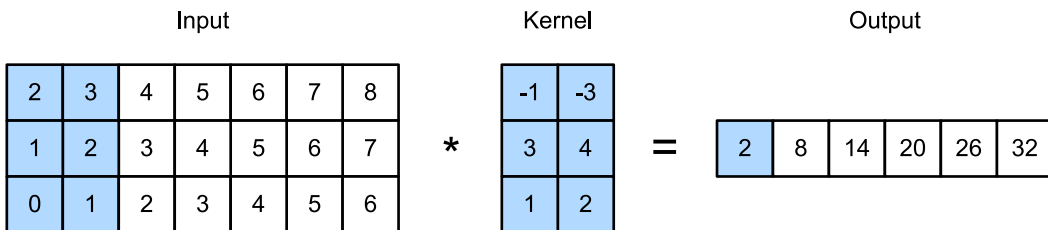

Fig. 11.5: One-dimensional cross-correlation operation with three input channels. The shaded parts are the first output element as well as the input and kernel array elements used in its calculation: $0 \times 1 + 1 \times 2 + 1 \times 3 + 2 \times 4 + 2 \times (-1) + 3 \times (-3) = 2$.

Now, we reproduce the results of the one-dimensional cross-correlation operation with multi-input channel in Figure 10.5.

11.8. Text Sentiment Classification: Using Convolutional Neural Networks (textCNN) 563

```
In [4]: def corr1d_multi_in(X, K):
            # First, we traverse along the 0th dimension (channel dimension) of X and
            # K. Then, we add them together by using * to turn the result list into a
            # positional argument of the add_n function
            return nd.add_n(*[corr1d(x, k) for x, k in zip(X, K)])

        X = nd.array([[0, 1, 2, 3, 4, 5, 6],
                      [1, 2, 3, 4, 5, 6, 7],
                      [2, 3, 4, 5, 6, 7, 8]])
        K = nd.array([[1, 2], [3, 4], [-1, -3]])
        corr1d_multi_in(X, K)

Out[4]:
        [ 2.  8. 14. 20. 26. 32.]
        <NDArray 6 @cpu(0)>
```

The definition of a two-dimensional cross-correlation operation tells us that a one-dimensional cross-correlation operation with multiple input channels can be regarded as a two-dimensional cross-correlation operation with a single input channel. As shown in Figure 10.6, we can also present the one-dimensional cross-correlation operation with multiple input channels in Figure 10.5 as the equivalent two-dimensional cross-correlation operation with a single input channel. Here, the height of the kernel is equal to the height of the input.

Fig. 11.6: Two-dimensional cross-correlation operation with a single input channel. The highlighted parts are the first output element and the input and kernel array elements used in its calculation: $2 \times (-1) + 3 \times (-3) + 1 \times 3 + 2 \times 4 + 0 \times 1 + 1 \times 2 = 2$.

Both the outputs in Figure 10.4 and Figure 10.5 have only one channel. We discussed how to specify multiple output channels in a two-dimensional convolutional layer in the *Multiple Input and Output Channels* section. Similarly, we can also specify multiple output channels in the one-dimensional convolutional layer to extend the model parameters in the convolutional layer.

11.8.2 Max-Over-Time Pooling Layer

Similarly, we have a one-dimensional pooling layer. The max-over-time pooling layer used in TextCNN actually corresponds to a one-dimensional global maximum pooling layer. Assuming that the input contains multiple channels, and each channel consists of values on different time steps, the output of each

channel will be the largest value of all time steps in the channel. Therefore, the input of the max-over-time pooling layer can have different time steps on each channel.

To improve computing performance, we often combine timing examples of different lengths into a mini-batch and make the lengths of each timing example in the batch consistent by appending special characters (such as 0) to the end of shorter examples. Naturally, the added special characters have no intrinsic meaning. Because the main purpose of the max-over-time pooling layer is to capture the most important features of timing, it usually allows the model to be unaffected by the manually added characters.

11.8.3 Read and Preprocess IMDb Data Sets

We still use the same IMDb data set as n the previous section for sentiment analysis. The following steps for reading and preprocessing the data set are the same as in the previous section.

```
In [5]: batch_size = 64
        d2l.download_imdb()
        train_data, test_data = d2l.read_imdb('train'), d2l.read_imdb('test')
        vocab = d2l.get_vocab_imdb(train_data)
        train_iter = gdata.DataLoader(gdata.ArrayDataset(
            *d2l.preprocess_imdb(train_data, vocab)), batch_size, shuffle=True)
        test_iter = gdata.DataLoader(gdata.ArrayDataset(
            *d2l.preprocess_imdb(test_data, vocab)), batch_size)
```

11.8.4 The TextCNN Model

TextCNN mainly uses a one-dimensional convolutional layer and max-over-time pooling layer. Suppose the input text sequence consists of n words, and each word is represented by a d-dimension word vector. Then the input example has a width of n, a height of 1, and d input channels. The calculation of textCNN can be mainly divided into the following steps:

1. Define multiple one-dimensional convolution kernels and use them to perform convolution calculations on the inputs. Convolution kernels with different widths may capture the correlation of different numbers of adjacent words.

2. Perform max-over-time pooling on all output channels, and then concatenate the pooling output values of these channels in a vector.

3. The concatenated vector is transformed into the output for each category through the fully connected layer. A dropout layer can be used in this step to deal with overfitting.
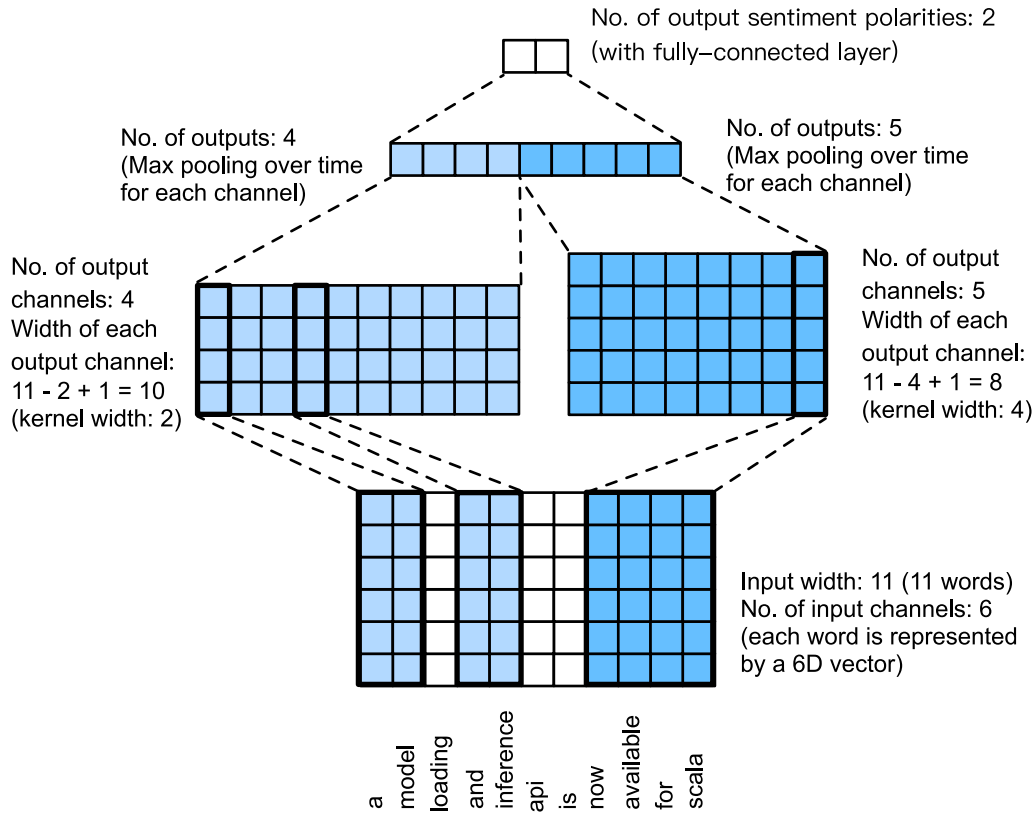
Fig. 11.7: TextCNN design.

Figure 10.7 gives an example to illustrate the textCNN. The input here is a sentence with 11 words, with each word represented by a 6-dimensional word vector. Therefore, the input sequence has a width of 11 and 6 input channels. We assume there are two one-dimensional convolution kernels with widths of 2 and 4, and 4 and 5 output channels, respectively. Therefore, after one-dimensional convolution calculation, the width of the four output channels is $11 - 2 + 1 = 10$, while the width of the other five channels is $11 - 4 + 1 = 8$. Even though the width of each channel is different, we can still perform max-over-time pooling for each channel and concatenate the pooling outputs of the 9 channels into a 9-dimensional vector. Finally, we use a fully connected layer to transform the 9-dimensional vector into a 2-dimensional output: positive sentiment and negative sentiment predictions.

Next, we will implement a textCNN model. Compared with the previous section, in addition to replacing the recurrent neural network with a one-dimensional convolutional layer, here we use two embedding layers, one with a fixed weight and another that participates in training.

```
In [6]: class TextCNN(nn.Block):
            def __init__(self, vocab, embed_size, kernel_sizes, num_channels,
```

```
                **kwargs):
        super(TextCNN, self).__init__(**kwargs)
        self.embedding = nn.Embedding(len(vocab), embed_size)
        # The embedding layer does not participate in training
        self.constant_embedding = nn.Embedding(len(vocab), embed_size)
        self.dropout = nn.Dropout(0.5)
        self.decoder = nn.Dense(2)
        # The max-over-time pooling layer has no weight, so it can share an
        # instance
        self.pool = nn.GlobalMaxPool1D()
        # Create multiple one-dimensional convolutional layers
        self.convs = nn.Sequential()
        for c, k in zip(num_channels, kernel_sizes):
            self.convs.add(nn.Conv1D(c, k, activation='relu'))

    def forward(self, inputs):
        # Concatenate the output of two embedding layers with shape of
        # (batch size, number of words, word vector dimension) by word vector
        embeddings = nd.concat(
            self.embedding(inputs), self.constant_embedding(inputs), dim=2)
        # According to the input format required by Conv1D, the word vector
        # dimension, that is, the channel dimension of the one-dimensional
        # convolutional layer, is transformed into the previous dimension
        embeddings = embeddings.transpose((0, 2, 1))
        # For each one-dimensional convolutional layer, after max-over-time
        # pooling, an NDArray with the shape of (batch size, channel size, 1)
        # can be obtained. Use the flatten function to remove the last
        # dimension and then concatenate on the channel dimension
        encoding = nd.concat(*[nd.flatten(
            self.pool(conv(embeddings))) for conv in self.convs], dim=1)
        # After applying the dropout method, use a fully connected layer to
        # obtain the output
        outputs = self.decoder(self.dropout(encoding))
        return outputs
```

Create a TextCNN instance. It has 3 convolutional layers with kernel widths of 3, 4, and 5, all with 100 output channels.

```
In [7]: embed_size, kernel_sizes, nums_channels = 100, [3, 4, 5], [100, 100, 100]
        ctx = d2l.try_all_gpus()
        net = TextCNN(vocab, embed_size, kernel_sizes, nums_channels)
        net.initialize(init.Xavier(), ctx=ctx)
```

Load Pre-trained Word Vectors

As in the previous section, load pre-trained 100-dimensional GloVe word vectors and initialize the embedding layers `embedding` and `constant_embedding`. Here, the former participates in training while the latter has a fixed weight.

```
In [8]: glove_embedding = text.embedding.create(
            'glove', pretrained_file_name='glove.6B.100d.txt', vocabulary=vocab)
        net.embedding.weight.set_data(glove_embedding.idx_to_vec)
        net.constant_embedding.weight.set_data(glove_embedding.idx_to_vec)
        net.constant_embedding.collect_params().setattr('grad_req', 'null')
```

Train and Evaluate the Model

Now we can train the model.

```
In [9]: lr, num_epochs = 0.001, 5
        trainer = gluon.Trainer(net.collect_params(), 'adam', {'learning_rate': lr})
        loss = gloss.SoftmaxCrossEntropyLoss()
        d2l.train(train_iter, test_iter, net, loss, trainer, ctx, num_epochs)

training on [gpu(0), gpu(1)]
epoch 1, loss 0.6047, train acc 0.716, test acc 0.830, time 19.5 sec
epoch 2, loss 0.3563, train acc 0.845, test acc 0.842, time 18.9 sec
epoch 3, loss 0.2673, train acc 0.891, test acc 0.859, time 18.9 sec
epoch 4, loss 0.1801, train acc 0.932, test acc 0.863, time 18.9 sec
epoch 5, loss 0.1092, train acc 0.960, test acc 0.867, time 18.8 sec
```

Below, we use the trained model to the classify sentiments of two simple sentences.

```
In [10]: d2l.predict_sentiment(net, vocab, ['this', 'movie', 'is', 'so', 'great'])

Out[10]: 'positive'

In [11]: d2l.predict_sentiment(net, vocab, ['this', 'movie', 'is', 'so', 'bad'])

Out[11]: 'negative'
```

Summary

- We can use one-dimensional convolution to process and analyze timing data.

- A one-dimensional cross-correlation operation with multiple input channels can be regarded as a two-dimensional cross-correlation operation with a single input channel.

- The input of the max-over-time pooling layer can have different numbers of time steps on each channel.

- TextCNN mainly uses a one-dimensional convolutional layer and max-over-time pooling layer.

Exercises

- Tune the hyper-parameters and compare the two sentiment analysis methods, using recurrent neural networks and using convolutional neural networks, as regards accuracy and operational efficiency.

- Can you further improve the accuracy of the model on the test set by using the three methods introduced in the previous section: tuning hyper-parameters, using larger pre-trained word vectors, and using the spaCy word tokenization tool?

- What other natural language processing tasks can you use textCNN for?

Reference

[1] Kim, Y. (2014). Convolutional neural networks for sentence classification. arXiv preprint arXiv:1408.5882.

Scan the QR Code to Discuss

11.9 Encoder-Decoder (seq2seq)

We have processed and analyzed variable-length input sequences in the previous chapters. However, in many applications, both the input and output can be variable-length sequences. For instance, in the case of machine translation, the input can be a variable-length sequence of English text, and the output can be a variable-length sequence of French text.

English input: They, are, watching, .

French output: Ils, regardent, .

When the input and output are both variable-length sequences, we can use the encoder-decoder[1] or the seq2seq model[2]. Both models use two recurrent neural networks (RNNs) named encoders and decoders. The encoder is used to analyze the input sequence and the decoder is used to generate the output sequence.

Figure 10.8 depicts a method for translating the English sentences above into French sentences using an encoder-decoder. In the training data set, we can attach a special symbol <eos> (end of sequence) after each sentence to indicate the termination of the sequence. For each time step, the encoder generates inputs following the order of words, punctuation, and special symbols <eos> in the English sentence. Figure 10.8 uses the hidden state of the encoder at the final time step as the encoding information for the input sentence. The decoder uses the encoding information of the input sentence, the output of the last time step, and the hidden state as inputs for each time step. We hope that the decoder can correctly output the translated French words, punctuation, and special symbols <eos> at each time step. It should be noted that the input of the decoder at the initial time step uses a special symbol <bos> to indicate the beginning of the sequence.
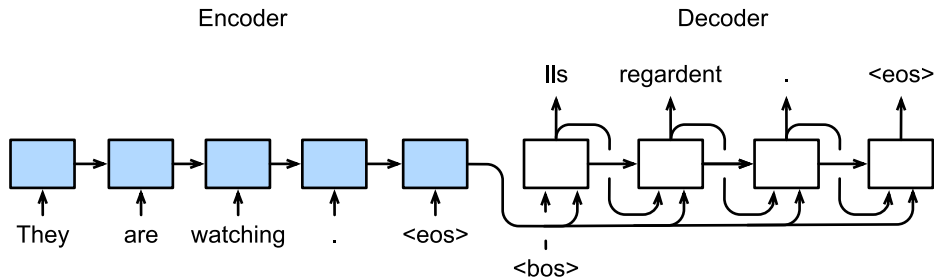
Fig. 11.8: Use an encoder-decoder to translate this sentence from English to French. The encoder and decoder are each recurrent neural networks.

Next, we are going to introduce the definitions of the encoder and decoder individually.

11.9.1 Encoder

The role of the encoder is to transform an input sequence of variable length into a fixed-length context variable c, and encode the input sequence information in that context variable. The most commonly used encoder is an RNN.

We will consider a time-series data instance with a batch size of 1. We assume that the input sequence is $x_1, \ldots, x_T$, such that x_i is the ith word in the input sentence. At time step t, the RNN will enter feature vector $\boldsymbol{x}_t$ for x_t and hidden state $\boldsymbol{h}_{t-1}$ from the previous time step will be transformed into the current hidden state $\boldsymbol{h}_t$. We can use function f to express the transformation of the RNN's hidden layer:

$$\boldsymbol{h}_t = f(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}).$$

Next, the encoder transforms the hidden state of each time step into context variables through custom function q.

$$\boldsymbol{c} = q(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_T).$$

For example, when we select $q(\boldsymbol{h}_1, \ldots, \boldsymbol{h}_T) = \boldsymbol{h}_T$, the context variable is the hidden state of input sequence $\boldsymbol{h}_T$ for the final time step.

The encoder discussed above is a unidirectional RNN, and the hidden state of each time step depends only on itself and the input subsequences from previous time steps. We can also construct encoders using bidirectional RNNs. In this case, the hidden state from each time step of the encoder depends on the subsequence before and after the time step (including the input of the current time step), which encodes the information of the entire sequence.

11.9.2 Decoder

As we just mentioned, the context variable c of the encoder's output encodes the entire input sequence $x_1, \ldots, x_T$. Given the output sequence $y_1, y_2, \ldots, y_{T'}$ in the training example, for each time step t' (the symbol differs from the input sequence and the encoder's time step t), the conditional probability of decoder output $y_{t'}$ will be based on the previous output sequence $y_1, \ldots, y_{t'-1}$ and context variable c, i.e. $\mathbb{P}(y_{t'} \mid y_1, \ldots, y_{t'-1}, c)$.

Therefore, we can use another RNN as a decoder. At time step t' of the output sequence, the decoder uses the output $y_{t'-1}$ from the previous time step and context variable c as its input and transforms their hidden state $s_{t'-1}$ from the previous time step into hidden state $s_{t'}$ of the current time step. Therefore, we can use function f to express the transformation of the decoder's hidden layer:

$$s_{t'} = g(y_{t'-1}, c, s_{t'-1}).$$

After obtaining the hidden state of the decoder, we can use a custom output layer and the softmax operation to compute $\mathbb{P}(y_{t'} \mid y_1, \ldots, y_{t'-1}, c)$. For example, using hidden state $s_{t'}$ based on the current time step of the decoder, the output $y_{t'-1}$ from the previous time step, and the context variable c to compute the probability distribution of output $y_{t'}$ from the current time step.

11.9.3 Model Training

According to the maximum likelihood estimation, we can maximize the conditional probability of the output sequence based on the input sequence

$$\mathbb{P}(y_1, \ldots, y_{T'} \mid x_1, \ldots, x_T) = \prod_{t'=1}^{T'} \mathbb{P}(y_{t'} \mid y_1, \ldots, y_{t'-1}, x_1, \ldots, x_T)$$
$$= \prod_{t'=1}^{T'} \mathbb{P}(y_{t'} \mid y_1, \ldots, y_{t'-1}, c),$$

to get the loss of the output sequence

$$-\log \mathbb{P}(y_1, \ldots, y_{T'} \mid x_1, \ldots, x_T) = -\sum_{t'=1}^{T'} \log \mathbb{P}(y_{t'} \mid y_1, \ldots, y_{t'-1}, c),$$

In model training, the mean of losses for all the output sequences is usually used as a loss function that needs to be minimized. In the model prediction discussed in Figure 10.8, we need to use the output of the decoder from the previous time step as the input to the current time step. In contrast, in training, we can also use the label of the label sequence from the previous time step as the input of the decoder for the current time step. This is called teacher forcing.

Summary

- The encoder-decoder (seq2seq) model can input and output a sequence of variable length.

- The encoder-decoder uses two RNNs.

- In encoder-decoder training, we can use teacher forcing.

Exercises

- In addition to machine translation, what other applications can you think of for encoder-decoder?

- What methods can be used to design the output layer of the decoder?

Reference

[1] Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078.

[2] Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. In Advances in neural information processing systems (pp. 3104-3112).

Scan the QR Code to Discuss

11.10 Beam Search

In the previous section, we discussed how to train an encoder-decoder with input and output sequences that are both of variable length. In this section, we are going to introduce how to use the encoder-decoder to predict sequences of variable length.

As in the previous section, when preparing to train the data set, we normally attach a special symbol <eos> after each sentence to indicate the termination of the sequence. We will continue to use this mathematical symbol in the discussion below. For ease of discussion, we assume that the output of the decoder is a sequence of text. Let the size of output text dictionary $\mathcal{Y}$ (contains special symbol <eos>) be $|\mathcal{Y}|$, and the maximum length of the output sequence be T'. There are a total $\mathcal{O}(|\mathcal{Y}|^{T'})$ types of possible

output sequences. All the subsequences after the special symbol <eos> in these output sequences will be discarded.

11.10.1 Greedy Search

First, we will take a look at a simple solution: greedy search. For any time step t' of the output sequence, we are going to search for the word with the highest conditional probability from $|\mathcal{Y}|$ numbers of words, with

$$y_{t'} = \underset{y \in \mathcal{Y}}{\operatorname{argmax}} \, \mathbb{P}(y \mid y_1, \ldots, y_{t'-1}, \boldsymbol{c})$$

as the output. Once the <eos> symbol is detected, or the output sequence has reached its maximum length T', the output is completed.

As we mentioned in out discussion of the decoder, the conditional probability of generating an output sequence based on the input sequence is $\prod_{t'=1}^{T'} \mathbb{P}(y_{t'} \mid y_1, \ldots, y_{t'-1}, \boldsymbol{c})$. We will take the output sequence with the highest conditional probability as the optimal sequence. The main problem with greedy search is that there is no guarantee that the optimal sequence will be obtained.

Take a look at the example below. We assume that there are four words A, B, C, and <eos> in the output dictionary. The four numbers under each time step in Figure 10.9 represent the conditional probabilities of generating A, B, C, and <eos> at that time step. At each time step, greedy search selects the word with the highest conditional probability. Therefore, the output sequence A, B, C, <eos> will be generated in Figure 10.9. The conditional probability of this output sequence is $0.5 \times 0.4 \times 0.4 \times 0.6 = 0.048$.

Time step	1	2	3	4
A	0.5	0.1	0.2	0.0
B	0.2	0.4	0.2	0.2
C	0.2	0.3	0.4	0.2
<eos>	0.1	0.2	0.2	0.6

Fig. 11.9: The four numbers under each time step represent the conditional probabilities of generating A, B, C, and <eos> at that time step. At each time step, greedy search selects the word with the highest conditional probability.

Now, we will look at another example shown in Figure 10.10. Unlike in Figure 10.9, Figure 10.10 selects the word C for it has the second highest conditional probability at time step 2. Since the output subsequences of time steps 1 and 2, on which time step 3 is based, are changed from A and B in Fig. 10.9 to A and C in Fig. 10.10, the conditional probability of each word generated at time step 3 has also changed in Fig. 10.10. We choose the word B, which has the highest conditional probability. Now, the output subsequences of time step 4 based on the first three time steps are A, C, and B, which are different from A, B, and C in Fig. 10.9. Therefore, the conditional probability of generating each word in time

step 4 in Figure 10.10 is also different from that in Figure 10.9. We find that the conditional probability of the output sequence A, C, B, <eos> at the current time step is $0.5 \times 0.3 \times 0.6 \times 0.6 = 0.054$, which is higher than the conditional probability of the output sequence obtained by greedy search. Therefore, the output sequence A, B, C, <eos> obtained by the greedy search is not an optimal sequence.

Time step	1	2	3	4
A	0.5	0.1	0.1	0.1
B	0.2	0.4	0.6	0.2
C	0.2	0.3	0.2	0.1
<eos>	0.1	0.2	0.1	0.6

Fig. 11.10: The four numbers under each time step represent the conditional probabilities of generating A, B, C, and <eos> at that time step. At time step 2, the word C, which has the second highest conditional probability, is selected.

11.10.2 Exhaustive Search

If the goal is to obtain the optimal sequence, we may consider using exhaustive search: an exhaustive examination of all possible output sequences, which outputs the sequence with the highest conditional probability.

Although we can use an exhaustive search to obtain the optimal sequence, its computational overhead $\mathcal{O}(|\mathcal{Y}|^{T'})$ is likely to be excessively high. For example, when $|\mathcal{Y}| = 10000$ and $T' = 10$, we will need to evaluate $10000^{10} = 10^{40}$ sequences. This is next to impossible to complete. The computational overhead of greedy search is $\mathcal{O}(|\mathcal{Y}| T')$, which is usually significantly less than the computational overhead of an exhaustive search. For example, when $|\mathcal{Y}| = 10000$ and $T' = 10$, we only need to evaluate $10000 \times 10 = 1 \times 10^5$ sequences.

11.10.3 Beam Search

Beam search is an improved algorithm based on greedy search. It has a hyper-parameter named beam size. We set it to k. At time step 1, we select k words with the highest conditional probability to be the first words of the k candidate output sequences. For each subsequent time step, we are going to select the k output sequences with the highest conditional probability from the total of $k |\mathcal{Y}|$ possible output sequences based on the k candidate output sequences from the previous time step. These will be the candidate output sequence for that time step. Finally, we will filter out the sequences containing the special symbol <eos> from the candidate output sequences of each time step and discard all the subsequences after it to obtain a set of final candidate output sequences.
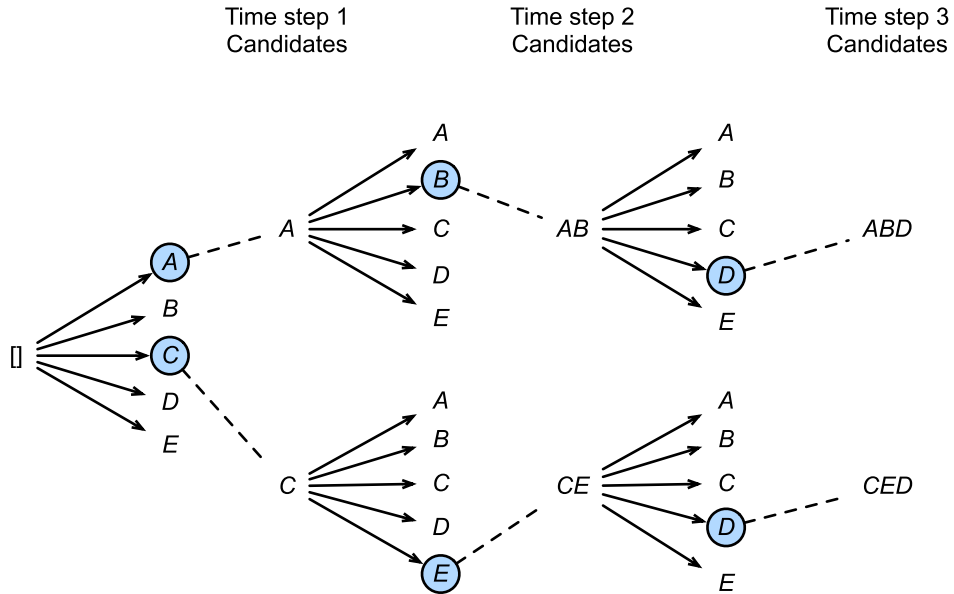
Fig. 11.11: The beam search process. The beam size is 2 and the maximum length of the output sequence is 3. The candidate output sequences are A, C, AB, CE, ABD, and CED.

Figure 10.11 demonstrates the process of beam search with an example. Suppose that the vocabulary of the output sequence only contains five elements: $\mathcal{Y} = \{A, B, C, D, E\}$ where one of them is a special symbol <eos>. Set beam size to 2, the maximum length of the output sequence to 3. At time step 1 of the output sequence, suppose the words with the highest conditional probability $\mathbb{P}(y_1 \mid \boldsymbol{c})$ are A and C. At time step 2, we compute $\mathbb{P}(y_2 \mid A, \boldsymbol{c})$ and $\mathbb{P}(y_2 \mid C, \boldsymbol{c})$ for all $y_2 \in \mathcal{Y}$, and pick the largest two among these 10 values: say $\mathbb{P}(B \mid A, \boldsymbol{c})$ and $\mathbb{P}(E \mid C, \boldsymbol{c})$. Then at time step 3, we compute $\mathbb{P}(y_3 \mid A, B, \boldsymbol{c})$ and $\mathbb{P}(y_3 \mid C, E, \boldsymbol{c})$ for all $y_3 \in \mathcal{Y}$, and pick the largest two among these 10 values: say $\mathbb{P}(D \mid A, B, \boldsymbol{c})$ and $\mathbb{P}(D \mid C, E, \boldsymbol{c})$. As a result, we obtain 6 candidates output sequences: (1) A; (2):math:C; (3) A, B; (4):math:C, E; (5):math:A, B, D; and (6):math:C, E, D. In the end, we will get the set of final candidate output sequences based on these 6 sequences.

In the set of final candidate output sequences, we will take the sequence with the highest score as the output sequence from those below:

$$\frac{1}{L^\alpha} \log \mathbb{P}(y_1, \ldots, y_L) = \frac{1}{L^\alpha} \sum_{t'=1}^{L} \log \mathbb{P}(y_{t'} \mid y_1, \ldots, y_{t'-1}, \boldsymbol{c}),$$

Here, L is the length of the final candidate sequence and the selection for α is generally 0.75. The L^α on the denominator is a penalty on the logarithmic addition scores for the longer sequences above. The computational overhead $\mathcal{O}(k \, |\mathcal{Y}| \, T')$ of the beam search can be obtained through analysis. The result is a computational overhead between those of greedy search and exhaustive search. In addition, greedy

search can be treated as a beam search with a beam size of 1. Beam search strikes a balance between computational overhead and search quality using a flexible beam size of k.

Summary

- Methods for predicting variable-length sequences include greedy search, exhaustive search, and beam search.

- Beam search strikes a balance between computational overhead and search quality using a flexible beam size.

Exercises

- Can we treat an exhaustive search as a beam search with a special beam size? Why?

- We used language models to create lyrics in the *Implementation of the Recurrent Neural Network from Scratch* section. Which kind of search does this output use? Can you improve it?

Scan the QR Code to Discuss

11.11 Attention Mechanism

As we learned in the *Encoder-Decoder(seq2seq)* section, the decoder relies on the same context variable at each time step to obtain input sequence information. When the encoder is an RNN, the context variable will be from the hidden state of its final time step.

Now, let us take another look at the translation example mentioned in that section: the input is an English sequence They, are, watching, ., and the output is a French sequence Ils, regardent, .. It is not hard to see that the decoder only needs to use partial information from the input sequence to generate each word in the output sequence. For example, at time step 1 of the output sequence, the decoder can mainly rely on the information of They and are to generate Ils. At time step 2, mainly the encoded information from watching is used to generate regardent. Finally, at time step 3, the period . is mapped directly. At each time step, it looks like the decoder is assigning different attentions to the encoded information of different time steps in the input sequence. This is the source of the attention mechanism[1].

Here, we will continue to use RNN as an example. The attention mechanism obtains the context variable by weighting the hidden state of all time steps of the encoder. The decoder adjusts these weights, i.e., attention weights, at each time step so that different portions of the input sequence can be focused on at different time steps and encoded into context variables of the corresponding time step. In this section, we will discuss how the attention mechanism works.

In the *Encoder-Decoder(seq2seq)* section, we were able to distinguish between the input sequence/encoder index t and the output sequence/decoder index t'. In that section, $\boldsymbol{s}_{t'} = g(\boldsymbol{y}_{t'-1}, \boldsymbol{c}, \boldsymbol{s}_{t'-1})$ is the hidden state of the decoder at time step t'. Here, $\boldsymbol{y}_{t'-1}$ is the feature representation of output $y_{t'-1}$ from the previous time step $t'-1$, and any time step t' uses the same context variable $\boldsymbol{c}$. However, in the attention mechanism, each time step of the decoder will use variable context variables. If $\boldsymbol{c}_{t'}$ is the context variable of the decoder at time step t', then the hidden state of the decoder at that time step can be rewritten as

$$\boldsymbol{s}_{t'} = g(\boldsymbol{y}_{t'-1}, \boldsymbol{c}_{t'}, \boldsymbol{s}_{t'-1}).$$

The key here is to figure out how to compute the context variable $\boldsymbol{c}_{t'}$ and use it to update the hidden state $\boldsymbol{s}_{t'}$. Below, we will introduce these two key points separately.

11.11.1 Compute the Context Variable

Figure 10.12 depicts how the attention mechanism computes the context variable for the decoder at time step 2. First, function a will compute the input of the softmax operation based on the hidden state of the decoder at time step 1 and the hidden states of the encoder at each time step. The Softmax operation outputs a probability distribution and weights the hidden state of each time step of the encoder to obtain a context variable.
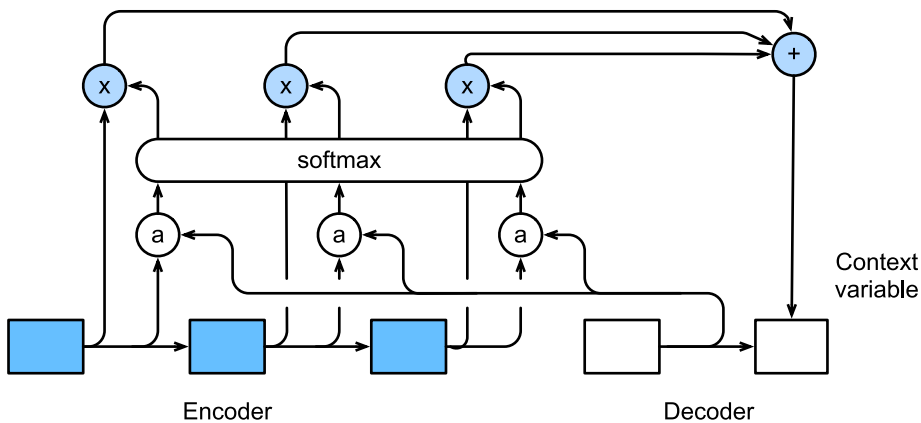
Fig. 11.12: Attention mechanism based on the encoder-decoder.

Specifically, if we know that the hidden state of the encoder at time step t is $\boldsymbol{h}_t$ and the total number of

time steps is T, then the context variable of the decoder at time step t' is the weighted average on all the hidden states of the encoder:

$$c_{t'} = \sum_{t=1}^{T} \alpha_{t't} h_t,$$

When t' is given, the value of weight $\alpha_{t't}$ at $t = 1, \ldots, T$ is a probability distribution. In order to obtain the probability distribution, we are going to use the softmax operation:

$$\alpha_{t't} = \frac{\exp(e_{t't})}{\sum_{k=1}^{T} \exp(e_{t'k})}, \quad t = 1, \ldots, T.$$

Now we need to define how to compute input $e_{t't}$ of the softmax operation in the formula above. Since $e_{t't}$ depends on both the decoder's time step t' and the encoder's time step t, we might as well use the decoder's hidden state $s_{t'-1}$ at that time step $t'-1$ and the encoder's hidden state h_t at that time step as the input and compute $e_{t't}$ with function a.

$$e_{t't} = a(s_{t'-1}, h_t).$$

Here, we have several options for function a. If the two input vectors are of the same length, a simple choice is to compute their inner product $a(s, h) = s^\top h$. In the paper that first introduced the attention mechanism, the authors transformed the concatenated input through a multilayer perceptron with a single hidden layer[1].

$$a(s, h) = v^\top \tanh(W_s s + W_h h),$$

Here, v, W_s, and W_h are all model parameters that can be learned.

Vectorization

We can also use vectorization to compute more efficiently within the attention mechanism. Generally speaking, the input of the attention model consists of query entries, key entries, and value entries. There is also a one-to-one correspondence between the key entries and value entries. Here, the value entry is a set of entries that requires a weighted average. In the weighted average, the weight of the value entry is obtained by computing the query entry and the key entry corresponding to the value entry.

In the example above, the query entry is the hidden state of the decoder, and the key entry and value entry are hidden states of the encoder. Now, we will look at a common simple case where the encoder and decoder have h hidden units and we have the function $a(s, h) = s^\top h$. Assume that we want to compute the context vector $c_{t'} \in \mathbb{R}^h$ based on the single hidden state of the decoder $s_{t'-1} \in \mathbb{R}^h$ and all the hidden states of the encoder $h_t \in \mathbb{R}^h, t = 1, \ldots, T$. We can let the query entry matrix $Q \in \mathbb{R}^{1 \times h}$ be $s_{t'-1}^\top$ and the key entry matrix $K \in \mathbb{R}^{T \times h}$ have the same value as the entry matrix $V \in \mathbb{R}^{T \times h}$, with all the values in row t set to $h_t^\top$. Now, we only need to use vectorization

$$\text{softmax}(QK^\top)V$$

to compute the transposed context vector $c_{t'}^\top$. When the query entry matrix Q has n rows, the formula above will be able to obtain the output matrix of row n. The output matrix and the query entry matrix correspond one-to-one on the same row.

11.11.2 Update the Hidden State

Using the gated recurrent unit (GRU) as an example, we can modify the design of the GRU slightly in the decoder[1]. The decoder's hidden state at time step t' will be

$$s_{t'} = z_{t'} \odot s_{t'-1} + (1 - z_{t'}) \odot \tilde{s}_{t'},$$

Here, the candidate implied states of the reset gate and update gate are

$$
\begin{aligned}
r_{t'} &= \sigma(W_{yr} y_{t'-1} + W_{sr} s_{t'-1} + W_{cr} c_{t'} + b_r), \\
z_{t'} &= \sigma(W_{yz} y_{t'-1} + W_{sz} s_{t'-1} + W_{cz} c_{t'} + b_z), \\
\tilde{s}_{t'} &= \tanh(W_{ys} y_{t'-1} + W_{ss}(s_{t'-1} \odot r_{t'}) + W_{cs} c_{t'} + b_s),
\end{aligned}
$$

Here, W and b with subscripts are the weight parameters and bias parameters of the GRU.

Summary

- We can use different context variables at each time step of the decoder and assign different attentions to the information encoded in different time steps of the input sequence.

- Generally speaking, the input of the attention model consists of query entries, key entries, and value entries. There is also a one-to-one correspondence between the key entries and value entries.

- With the attention mechanism, we can adopt vectorization for higher efficiency.

Exercises

- Based on the model design in this section, why can't we concatenate hidden state $s_{t'-1}^\top \in \mathbb{R}^{1 \times h}, t' \in 1, \ldots, T'$ from different time steps of the decoder to create the query entry matrix $Q \in \mathbb{R}^{T' \times h}$ to compute context variable $c_{t'}^\top, t' \in 1, \ldots, T'$ of the attention mechanism at different time steps simultaneously?

- Without modifying the function `gru` from the *Gated Recurrent Unit (GRU)* section, how can we use it to implement the decoder introduced in this section?

- In addition to natural language processing, where else can the attention mechanism be applied?

Reference

[1] Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473.

Scan the QR Code to Discuss

11.12 Machine Translation

Machine translation refers to the automatic translation of a segment of text from one language to another. Because a sequence of texts does not necessarily retain the same length in different languages, we use machine translation as an example to introduce the applications of the encoder-decoder and attention mechanism.

11.12.1 Read and Pre-process Data

We will define some special symbols first. The <pad> (padding) symbol is added after a shorter sequence until each sequence is equal in length and the <bos> and <eos> symbols indicate the beginning and end of the sequence.

```
In [1]: import collections
        import io
        import math
        from mxnet import autograd, gluon, init, nd
        from mxnet.contrib import text
        from mxnet.gluon import data as gdata, loss as gloss, nn, rnn

        PAD, BOS, EOS = '<pad>', '<bos>', '<eos>'
```

Then, we define two auxiliary functions to preprocess the data to be read later.

```
In [2]: # For a sequence, we record all the words in all_tokens in order to
        # subsequently construct the dictionary, then we add PAD after the sequence,
        # until the length becomes max_seq_len. Then, we record the sequence in
        # all_seqs
        def process_one_seq(seq_tokens, all_tokens, all_seqs, max_seq_len):
            all_tokens.extend(seq_tokens)
            seq_tokens += [EOS] + [PAD] * (max_seq_len - len(seq_tokens) - 1)
            all_seqs.append(seq_tokens)
```

```
        # Use all the words to construct a dictionary. Construct an NDArray instance
        # after transforming the words in all sequences into a word index
        def build_data(all_tokens, all_seqs):
            vocab = text.vocab.Vocabulary(collections.Counter(all_tokens),
                                          reserved_tokens=[PAD, BOS, EOS])
            indices = [vocab.to_indices(seq) for seq in all_seqs]
            return vocab, nd.array(indices)
```

For simplicity, we use a very small French-English data set here. In this data set, each line is a French sentence and its corresponding English sentence, separated by '\t'. When reading data, we attach the <eos> symbol at the end of the sentence, and if necessary, make the length of each sequence max_seq_len by adding the <pad> symbol. We create separate dictionaries for French and English words. The index of French words and the index of the English words are independent of each other.

```
In [3]: def read_data(max_seq_len):
            # In and out are the abbreviations of input and output, respectively
            in_tokens, out_tokens, in_seqs, out_seqs = [], [], [], []
            with io.open('../data/fr-en-small.txt') as f:
                lines = f.readlines()
            for line in lines:
                in_seq, out_seq = line.rstrip().split('\t')
                in_seq_tokens, out_seq_tokens = in_seq.split(' '), out_seq.split(' ')
                # If a sequence is longer than the max_seq_len after adding EOS, this
                # example will be ignored
                if max(len(in_seq_tokens), len(out_seq_tokens)) > max_seq_len - 1:
                    continue
                process_one_seq(in_seq_tokens, in_tokens, in_seqs, max_seq_len)
                process_one_seq(out_seq_tokens, out_tokens, out_seqs, max_seq_len)
            in_vocab, in_data = build_data(in_tokens, in_seqs)
            out_vocab, out_data = build_data(out_tokens, out_seqs)
            return in_vocab, out_vocab, gdata.ArrayDataset(in_data, out_data)
```

Set the maximum length of the sequence to 7, then review the first example read. The example contains a French word index sequence and an English word index sequence.

```
In [4]: max_seq_len = 7
        in_vocab, out_vocab, dataset = read_data(max_seq_len)
        dataset[0]
Out[4]: (
        [ 6.  5. 46.  4.  3.  1.  1.]
        <NDArray 7 @cpu(0)>,
        [ 9.  5. 28.  4.  3.  1.  1.]
        <NDArray 7 @cpu(0)>)
```

11.12.2 Encoder-Decoder with Attention Mechanism

We will use an encoder-decoder with an attention mechanism to translate a short French paragraph into English. Next, we will show how to implement the model.

Encoder

In the encoder, we use the word embedding layer to obtain a feature index from the word index of the input language and then input it into a multi-level gated recurrent unit. As we mentioned in the *Concise Implementation of Recurrent Neural Networks* section, Gluon's `rnn.GRU` instance also returns the multi-layer hidden states of the output and final time steps after forward calculation. Here, the output refers to the hidden state of the hidden layer of the last layer at each time step, and it does not involve output layer calculation. The attention mechanism uses these output as key items and value items.

```
In [5]: class Encoder(nn.Block):
            def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                         drop_prob=0, **kwargs):
                super(Encoder, self).__init__(**kwargs)
                self.embedding = nn.Embedding(vocab_size, embed_size)
                self.rnn = rnn.GRU(num_hiddens, num_layers, dropout=drop_prob)

            def forward(self, inputs, state):
                # The input shape is (batch size, number of time steps). Change the
                # example dimension and time step dimension of the output
                embedding = self.embedding(inputs).swapaxes(0, 1)
                return self.rnn(embedding, state)

            def begin_state(self, *args, **kwargs):
                return self.rnn.begin_state(*args, **kwargs)
```

Next, we will create a mini-batch sequence input with a batch size of 4 and 7 time steps. We assume the number of hidden layers of the gated recurrent unit is 2 and the number of hidden units is 16. The output shape returned by the encoder after performing forward calculation on the input is (number of time steps, batch size, number of hidden units). The shape of the multi-layer hidden state of the gated recurrent unit in the final time step is (number of hidden layers, batch size, number of hidden units). For the gated recurrent unit, the `state` list contains only one element, which is the hidden state. If long short-term memory is used, the `state` list will also contain another element, which is the memory cell.

```
In [6]: encoder = Encoder(vocab_size=10, embed_size=8, num_hiddens=16, num_layers=2)
        encoder.initialize()
        output, state = encoder(nd.zeros((4, 7)), encoder.begin_state(batch_size=4))
        output.shape, state[0].shape
Out[6]: ((7, 4, 16), (2, 4, 16))
```

Attention Mechanism

Before we introduce how to implement vectorization calculation for the attention mechanism, we will take a look at the `flatten` option for a `Dense` instance. When the input dimension is greater than 2, by default, the `Dense` instance will treat all dimensions other than the first dimension (example dimension) as feature dimensions that require affine transformation, and will automatically convert the input into a two-dimensional matrix with rows of behavioral examples and columns of features. After calculation, the shape of the output matrix is (number of examples, number of outputs). If we want the fully connected layer to only perform affine transformation on the last dimension of the input while keeping the shapes of the other dimensions unchanged, we need to set the `flatten` option of the `Dense` instance to

`False`. In the following example, the fully connected layer only performs affine transformation on the last dimension of the input, therefore, only the last dimension of the output shape becomes the number of outputs of the fully connected layer, i.e. 2.

```
In [7]: dense = nn.Dense(2, flatten=False)
        dense.initialize()
        dense(nd.zeros((3, 5, 7))).shape

Out[7]: (3, 5, 2)
```

We will implement the function a defined in the *Attention Mechanism* section to transform the concatenated input through a multilayer perceptron with a single hidden layer. The input of the hidden layer is a one-to-one concatenation between the hidden state of the decoder and the hidden state of the encoder on all time steps, which uses tanh as the activation function. The number of outputs of the output layer is 1. Neither of the 2 `Dense` instances use a bias or flatten. Here, the length of the vector v in the a function definition is a hyper-parameter, i.e. `attention_size`.

```
In [8]: def attention_model(attention_size):
            model = nn.Sequential()
            model.add(nn.Dense(attention_size, activation='tanh', use_bias=False,
                               flatten=False),
                      nn.Dense(1, use_bias=False, flatten=False))
            return model
```

The inputs of the attention model include query items, key items, and value items. Assume the encoder and decoder have the same number of hidden units. The query item here is the hidden state of the decoder in the previous time step, with a shape of (batch size, number of hidden units); the key and the value items are the hidden states of the encoder at all time steps, with a shape of (number of time steps, batch size, number of hidden units). The attention model returns the context variable of the current time step, and the shape is (batch size, number of hidden units).

```
In [9]: def attention_forward(model, enc_states, dec_state):
            # Broadcast the decoder hidden state to the same shape as the encoder
            # hidden state and then perform concatenation
            dec_states = nd.broadcast_axis(
                dec_state.expand_dims(0), axis=0, size=enc_states.shape[0])
            enc_and_dec_states = nd.concat(enc_states, dec_states, dim=2)
            # The shape is (number of time steps, batch size, 1)
            e = model(enc_and_dec_states)
            # Perform the softmax operation on the time step dimension
            alpha = nd.softmax(e, axis=0)
            # This returns the context variable
            return (alpha * enc_states).sum(axis=0)
```

In the example below, the encoder has 10 time steps and a batch size of 4. Both the encoder and the decoder have 8 hidden units. The attention model returns a mini-batch of context vectors, and the length of each context vector is equal to the number of hidden units of the encoder. Therefore, the output shape is (4, 8).

```
In [10]: seq_len, batch_size, num_hiddens = 10, 4, 8
         model = attention_model(10)
         model.initialize()
         enc_states = nd.zeros((seq_len, batch_size, num_hiddens))
```

```
            dec_state = nd.zeros((batch_size, num_hiddens))
            attention_forward(model, enc_states, dec_state).shape
Out[10]: (4, 8)
```

Decoder with Attention Mechanism

We directly use the hidden state of the encoder in the final time step as the initial hidden state of the decoder. This requires that the encoder and decoder RNNs have the same numbers of layers and hidden units.

In forward calculation of the decoder, we first calculate and obtain the context vector of the current time step by using the attention model introduced above. Since the input of the decoder comes from the word index of the output language, we obtain the feature expression of the input through the word embedding layer, and then concatenate the context vector in the feature dimension. We calculate the output and hidden state of the current time step through the gated recurrent unit, using the concatenated results and the hidden state of the previous time step. Finally, we use the fully connected layer to transform the output into predictions for each output word, with the shape of (batch size, output dictionary size).

```
In [11]: class Decoder(nn.Block):
             def __init__(self, vocab_size, embed_size, num_hiddens, num_layers,
                          attention_size, drop_prob=0, **kwargs):
                 super(Decoder, self).__init__(**kwargs)
                 self.embedding = nn.Embedding(vocab_size, embed_size)
                 self.attention = attention_model(attention_size)
                 self.rnn = rnn.GRU(num_hiddens, num_layers, dropout=drop_prob)
                 self.out = nn.Dense(vocab_size, flatten=False)

             def forward(self, cur_input, state, enc_states):
                 # Use the attention mechanism to calculate the context vector
                 c = attention_forward(self.attention, enc_states, state[0][-1])
                 # The embedded input and the context vector are concatenated in the
                 # feature dimension
                 input_and_c = nd.concat(self.embedding(cur_input), c, dim=1)
                 # Add a time step dimension, with 1 time step, for the concatenation
                 # of the input and the context vector
                 output, state = self.rnn(input_and_c.expand_dims(0), state)
                 # Remove the time step dimension, so the output shape is (batch size,
                 # output dictionary size)
                 output = self.out(output).squeeze(axis=0)
                 return output, state

             def begin_state(self, enc_state):
                 # Directly use the hidden state of the final time step of the encoder
                 # as the initial hidden state of the decoder.
                 return enc_state
```

11.12.3 Training

We first implement the `batch_loss` function to calculate the loss of a mini-batch. The input of the decoder in the initial time step is the special character `BOS`. After that, the input of the decoder in a

given time step is the word from the example output sequence in the previous time step, that is, teacher forcing. Also, just as in the implementation in the *Implementation of Word2vec* section, we also use mask variables here to avoid the impact of padding on loss function calculations.

```
In [12]: def batch_loss(encoder, decoder, X, Y, loss):
             batch_size = X.shape[0]
             enc_state = encoder.begin_state(batch_size=batch_size)
             enc_outputs, enc_state = encoder(X, enc_state)
             # Initialize the hidden state of the decoder
             dec_state = decoder.begin_state(enc_state)
             # The input of decoder at the initial time step is BOS
             dec_input = nd.array([out_vocab.token_to_idx[BOS]] * batch_size)
             # We will use the mask variable to ignore the loss when the label is PAD
             mask, num_not_pad_tokens = nd.ones(shape=(batch_size,)), 0
             l = nd.array([0])
             for y in Y.T:
                 dec_output, dec_state = decoder(dec_input, dec_state, enc_outputs)
                 l = l + (mask * loss(dec_output, y)).sum()
                 dec_input = y  # Use teacher forcing
                 num_not_pad_tokens += mask.sum().asscalar()
                 # When we encounter EOS, words after the sequence will all be PAD and
                 # the mask for the corresponding position is set to 0
                 mask = mask * (y != out_vocab.token_to_idx[EOS])
             return l / num_not_pad_tokens
```

In the training function, we need to update the model parameters of the encoder and the decoder at the same time.

```
In [13]: def train(encoder, decoder, dataset, lr, batch_size, num_epochs):
             encoder.initialize(init.Xavier(), force_reinit=True)
             decoder.initialize(init.Xavier(), force_reinit=True)
             enc_trainer = gluon.Trainer(encoder.collect_params(), 'adam',
                                         {'learning_rate': lr})
             dec_trainer = gluon.Trainer(decoder.collect_params(), 'adam',
                                         {'learning_rate': lr})
             loss = gloss.SoftmaxCrossEntropyLoss()
             data_iter = gdata.DataLoader(dataset, batch_size, shuffle=True)
             for epoch in range(num_epochs):
                 l_sum = 0.0
                 for X, Y in data_iter:
                     with autograd.record():
                         l = batch_loss(encoder, decoder, X, Y, loss)
                     l.backward()
                     enc_trainer.step(1)
                     dec_trainer.step(1)
                     l_sum += l.asscalar()
                 if (epoch + 1) % 10 == 0:
                     print("epoch %d, loss %.3f" % (epoch + 1, l_sum /
     ↪   len(data_iter)))
```

Next, we create a model instance and set hyper-parameters. Then, we can train the model.

```
In [14]: embed_size, num_hiddens, num_layers = 64, 64, 2
         attention_size, drop_prob, lr, batch_size, num_epochs = 10, 0.5, 0.01, 2, 50
         encoder = Encoder(len(in_vocab), embed_size, num_hiddens, num_layers,
                           drop_prob)
         decoder = Decoder(len(out_vocab), embed_size, num_hiddens, num_layers,
```

```
                          attention_size, drop_prob)
        train(encoder, decoder, dataset, lr, batch_size, num_epochs)

epoch 10, loss 0.540
epoch 20, loss 0.319
epoch 30, loss 0.222
epoch 40, loss 0.088
epoch 50, loss 0.028
```

11.12.4 Prediction

We introduced three methods to generate the output of the decoder at each time step in the *Beam Search* section. Here we implement the simplest method, greedy search.

```
In [15]: def translate(encoder, decoder, input_seq, max_seq_len):
             in_tokens = input_seq.split(' ')
             in_tokens += [EOS] + [PAD] * (max_seq_len - len(in_tokens) - 1)
             enc_input = nd.array([in_vocab.to_indices(in_tokens)])
             enc_state = encoder.begin_state(batch_size=1)
             enc_output, enc_state = encoder(enc_input, enc_state)
             dec_input = nd.array([out_vocab.token_to_idx[BOS]])
             dec_state = decoder.begin_state(enc_state)
             output_tokens = []
             for _ in range(max_seq_len):
                 dec_output, dec_state = decoder(dec_input, dec_state, enc_output)
                 pred = dec_output.argmax(axis=1)
                 pred_token = out_vocab.idx_to_token[int(pred.asscalar())]
                 # When an EOS symbol is found at any time step, the output sequence is
                 # complete
                 if pred_token == EOS:
                     break
                 else:
                     output_tokens.append(pred_token)
                     dec_input = pred
             return output_tokens
```

Simply test the model. Enter the French sentence ils regardent.. The translated English sentence should be they are watching.

```
In [16]: input_seq = 'ils regardent .'
         translate(encoder, decoder, input_seq, max_seq_len)

Out[16]: ['they', 'are', 'watching', '.']
```

11.12.5 Evaluation of Translation Results

BLEU (Bilingual Evaluation Understudy) is often used to evaluate machine translation results[1]. For any subsequence in the model prediction sequence, BLEU evaluates whether this subsequence appears in the label sequence.

Specifically, the precision of the subsequence with n words is p_n. It is the ratio of the number of subsequences with n matched words for the prediction sequence and label sequence to the number of subse-

quences with n words in the prediction sequence. For example, assume the label sequence is A, B, C, D, E, F, and the prediction sequence is A, B, B, C, D. Then $p_1 = 4/5$, $p_2 = 3/4$, $p_3 = 1/3$, $and\, p_4 = 0$. Assume len_{label} and len_{pred} are the numbers of words in the label sequence and the prediction sequence. Then, BLEU is defined as

$$\exp\left(\min\left(0, 1 - \frac{len_{\text{label}}}{len_{\text{pred}}}\right)\right) \prod_{n=1}^{k} p_n^{1/2^n},$$

Here, k is the maximum number of words in the subsequence we wish to match. It can be seen that the BLEU is 1 when the prediction sequence and the label sequence are identical.

Because matching longer subsequences is more difficult than matching shorter subsequences, BLEU gives greater weight to the precision of longer subsequence matches. For example, when p_n is fixed at 0.5, as n increases, $0.5^{1/2} \approx 0.7, 0.5^{1/4} \approx 0.84, 0.5^{1/8} \approx 0.92$, $and\, 0.5^{1/16} \approx 0.96$. In addition, the prediction of shorter sequences by the model tends to obtain higher p_n values. Therefore, the coefficient before the multiplication term in the above equation is a penalty to the shorter output. For example, when $k = 2$, we assume the label sequence is A, B, C, D, E, F and the prediction sequence is A, B. Although $p_1 = p_2 = 1$, the penalty factor is $\exp(1 - 6/2) \approx 0.14$, so the BLEU is also close to 0.14.

Next, we calculate the BLEU

```
In [17]: def bleu(pred_tokens, label_tokens, k):
             len_pred, len_label = len(pred_tokens), len(label_tokens)
             score = math.exp(min(0, 1 - len_label / len_pred))
             for n in range(1, k + 1):
                 num_matches = 0
                 for i in range(len_pred - n + 1):
                     if ' '.join(pred_tokens[i: i + n]) in ' '.join(label_tokens):
                         num_matches += 1
                 score *= math.pow(num_matches / (len_pred - n + 1), math.pow(0.5, n))
             return score
```

and define an auxiliary printing function.

```
In [18]: def score(input_seq, label_seq, k):
             pred_tokens = translate(encoder, decoder, input_seq, max_seq_len)
             label_tokens = label_seq.split(' ')
             print('bleu %.3f, predict: %s' % (bleu(pred_tokens, label_tokens, k),
                                               ' '.join(pred_tokens)))
```

A correct prediction receives a score of 1.

```
In [19]: score('ils regardent .', 'they are watching .', k=2)
```

```
bleu 1.000, predict: they are watching .
```

Test an example that is not in the training set.

```
In [20]: score('ils sont canadiens .', 'they are canadian .', k=2)
```

```
bleu 0.658, predict: they are actors .
```

Summary

- We can apply encoder-decoder and attention mechanisms to machine translation.

- BLEU can be used to evaluate translation results.

Exercises

- If the encoder and decoder have different number of hidden units or layers, how can we improve the decoder's hidden state initialization method?

- During training, we experiment by replacing teacher forcing with the output of the decoder at the previous time step as the input of the decoder at the current time step. Has the result changed?

- Try to train the model with larger translation data sets, such as WMT[2] and Tatoeba Project[3].

Reference

[1] Papineni, K., Roukos, S., Ward, T., & Zhu, W. J. (2002, July). BLEU: a method for automatic evaluation of machine translation. In Proceedings of the 40th annual meeting on association for computational linguistics (pp. 311-318). Association for Computational Linguistics.

[2] WMT. http://www.statmt.org/wmt14/translation-task.html

[3] Tatoeba Project. http://www.manythings.org/anki/

Scan the QR Code to Discuss

12

Appendix

12.1 List of Main Symbols

The main symbols used in this book are listed below.

12.1.1 Numbers

x	Scalar
$\boldsymbol{x}$	Vector
$\boldsymbol{X}$	Matrix
X	Tensor

12.1.2 Sets

$\mathcal{X}$	Set
$\mathbb{R}$	Real number set
$\mathbb{R}^n$	Real number vector set of n dimension
$\mathbb{R}^{x \times y}$	Real number matrix set with x rows and y columns

12.1.3 Operators

$(\cdot)^{\top}$	Vector or matrix transposition
$\odot$	Multiply by element
$\|\mathcal{X}\|$	Number of elements in the set $\mathcal{X}$
$\|\cdot\|_p$	L_p norm
$\|\cdot\|$	L_2 norm
$\sum$	Continuous addition
$\prod$	Continuous multiplication

12.1.4 Functions

$f(\cdot)$	Function
$\log(\cdot)$	Natural logarithmic function
$\exp(\cdot)$	Exponential function

12.1.5 Derivatives and Gradients

$\frac{dy}{dx}$	Derivative of y with respect to x
:math:'frac{partial y}{partial x}'	Partial derivative of y with respect to x
$\nabla_{\cdot} y$	Gradient of y with respect to $\cdot$

12.1.6 Probability and Statistics

$\mathbb{P}(\cdot)$	Probability distribution
$\cdot \sim \mathbb{P}$	Random variable $\cdot$ obeys the probability distribution $\mathbb{P}$
$\mathbb{P}(\cdot \mid \cdot)$	Conditional probability
$\mathbb{E}_{\cdot}(f(\cdot))$	Expectation of $f(\cdot)$ with respect to $\cdot$

12.1.7 Complexity

$\mathcal{O}$	Big O notation

12.2 Mathematical Basis

This section summarizes the basic knowledge of linear algebra, differentiation, and probability required to understand the contents in this book. To avoid long discussions of mathematical knowledge not required to understand this book, a few definitions in this section are slightly simplified.

12.2.1 Linear Algebra

Below we summarize the concepts of vectors, matrices, operations, norms, eigenvectors, and eigenvalues.

Vectors

Vectors in this book refer to column vectors. An n-dimensional vector $\boldsymbol{x}$ can be written as

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix},$$

where $x_1, \ldots, x_n$ are elements of the vector. To express that $\boldsymbol{x}$ is an n-dimensional vector with elements from the set of real numbers, we write $\boldsymbol{x} \in \mathbb{R}^n$ or $\boldsymbol{x} \in \mathbb{R}^{n \times 1}$.

Matrices

An expression for a matrix with m rows and n columns can be written as

$$\boldsymbol{X} = \begin{bmatrix} x_{11} & x_{12} & \ldots & x_{1n} \\ x_{21} & x_{22} & \ldots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \ldots & x_{mn} \end{bmatrix},$$

Here, x_{ij} is the element in row i and column j in the matrix $\boldsymbol{X}$ ($1 \leq i \leq m, 1 \leq j \leq n$). To express that $\boldsymbol{X}$ is a matrix with m rows and n columns consisting of elements from the set of real numbers, we write $\boldsymbol{X} \in \mathbb{R}^{m \times n}$. It is not difficult to see that vectors are a special class of matrices.

Operations

Assume the elements in the n-dimensional vector $\boldsymbol{a}$ are $a_1, \ldots, a_n$, and the elements in the n-dimensional vector $\boldsymbol{b}$ are $b_1, \ldots, b_n$. The dot product (internal product) of vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ is a scalar:

$$\boldsymbol{a} \cdot \boldsymbol{b} = a_1 b_1 + \ldots + a_n b_n.$$

Assume two matrices with m rows and n columns:

$$
A = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} & \ldots & b_{1n} \\ b_{21} & b_{22} & \ldots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \ldots & b_{mn} \end{bmatrix}.
$$

The transpose of a matrix A with m rows and n columns is a matrix with n rows and m columns whose rows are formed from the columns of the original matrix:

$$
A^\top = \begin{bmatrix} a_{11} & a_{21} & \ldots & a_{m1} \\ a_{12} & a_{22} & \ldots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \ldots & a_{mn} \end{bmatrix}.
$$

To add two matrices of the same shape, we add them element-wise:

$$
A + B = \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} & \ldots & a_{1n} + b_{1n} \\ a_{21} + b_{21} & a_{22} + b_{22} & \ldots & a_{2n} + b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} + b_{m1} & a_{m2} + b_{m2} & \ldots & a_{mn} + b_{mn} \end{bmatrix}.
$$

We use the symbol $\odot$ to indicate the element-wise multiplication of two matrices:

$$
A \odot B = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \ldots & a_{1n}b_{1n} \\ a_{21}b_{21} & a_{22}b_{22} & \ldots & a_{2n}b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{m1} & a_{m2}b_{m2} & \ldots & a_{mn}b_{mn} \end{bmatrix}.
$$

Define a scalar k. Multiplication of scalars and matrices is also an element-wise multiplication:

$$
kA = \begin{bmatrix} ka_{11} & ka_{12} & \ldots & ka_{1n} \\ ka_{21} & ka_{22} & \ldots & ka_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ ka_{m1} & ka_{m2} & \ldots & ka_{mn} \end{bmatrix}.
$$

Other operations such as scalar and matrix addition, and division by an element are similar to the multiplication operation in the above equation. Calculating the square root or taking logarithms of a matrix are performed by calculating the square root or logarithm, respectively, of each element of the matrix to obtain a matrix with the same shape as the original matrix.

Matrix multiplication is different from element-wise matrix multiplication. Assume A is a matrix with m rows and p columns and B is a matrix with p rows and n columns. The product (matrix multiplication)

of these two matrices is denoted

$$
\boldsymbol{AB} =
\begin{bmatrix}
a_{11} & a_{12} & \dots & a_{1p} \\
a_{21} & a_{22} & \dots & a_{2p} \\
\vdots & \vdots & \ddots & \vdots \\
a_{i1} & a_{i2} & \dots & a_{ip} \\
\vdots & \vdots & \ddots & \vdots \\
a_{m1} & a_{m2} & \dots & a_{mp}
\end{bmatrix}
\begin{bmatrix}
b_{11} & b_{12} & \dots & b_{1j} & \dots & b_{1n} \\
b_{21} & b_{22} & \dots & b_{2j} & \dots & b_{2n} \\
\vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
b_{p1} & b_{p2} & \dots & b_{pj} & \dots & b_{pn}
\end{bmatrix},
$$

is a matrix with m rows and n columns, with the element in row i and column j ($1 \le i \le m, 1 \le j \le n$) equal to

$$
a_{i1}b_{1j} + a_{i2}b_{2j} + \dots + a_{ip}b_{pj} = \sum_{k=1}^{p} a_{ik}b_{kj}.
$$

Norms

Assume the elements in the n-dimensional vector $\boldsymbol{x}$ are $x_1, \dots, x_n$. The L_p norm of the vector $\boldsymbol{x}$ is

$$
\|\boldsymbol{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}.
$$

For example, the L_1 norm of $\boldsymbol{x}$ is the sum of the absolute values of the vector elements:

$$
\|\boldsymbol{x}\|_1 = \sum_{i=1}^{n} |x_i|.
$$

While the L_2 norm of $\boldsymbol{x}$ is the square root of the sum of the squares of the vector elements:

$$
\|\boldsymbol{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}.
$$

We usually use $\|\boldsymbol{x}\|$ to refer to the L_2 norm of $\boldsymbol{x}$.

Assume $\boldsymbol{X}$ is a matrix with m rows and n columns. The Frobenius norm of matrix $\boldsymbol{X}$ is the square root of the sum of the squares of the matrix elements:

$$
\|\boldsymbol{X}\|_F = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij}^2},
$$

Here, x_{ij} is the element of matrix $\boldsymbol{X}$ in row i and column j.

Eigenvectors and Eigenvalues

Let A be a matrix with n rows and n columns. If λ is a scalar and v is a non-zero n-dimensional vector with

$$Av = \lambda v,$$

then v is an called eigenvector vector of matrix A, and λ is called an eigenvalue of A corresponding to v.

12.2.2 Differentials

Here we briefly introduce some basic concepts and calculations for differentials.

Derivatives and Differentials

Assume the input and output of function $f : \mathbb{R} \to \mathbb{R}$ are both scalars. The derivative f is defined as

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h},$$

when the limit exists (and f is said to be differentiable). Given $y = f(x)$, where x and y are the arguments and dependent variables of function f, respectively, the following derivative and differential expressions are equivalent:

$$f'(x) = y' = \frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}f}{\mathrm{d}x} = \frac{\mathrm{d}}{\mathrm{d}x} f(x) = \mathrm{D}f(x) = \mathrm{D}_x f(x),$$

Here, the symbols D and $\mathrm{d}/\mathrm{d}x$ are also called differential operators. Common differential calculations are $\mathrm{D}C = 0$ (C is a constant), $\mathrm{D}x^n = nx^{n-1}$ (n is a constant), and $\mathrm{D}e^x = e^x$, $\mathrm{D}\ln(x) = 1/x$.

If functions f and g are both differentiable and C is a constant, then

$$\frac{\mathrm{d}}{\mathrm{d}x}[Cf(x)] = C\frac{\mathrm{d}}{\mathrm{d}x}f(x),$$

$$\frac{\mathrm{d}}{\mathrm{d}x}[f(x) + g(x)] = \frac{\mathrm{d}}{\mathrm{d}x}f(x) + \frac{\mathrm{d}}{\mathrm{d}x}g(x),$$

$$\frac{\mathrm{d}}{\mathrm{d}x}[f(x)g(x)] = f(x)\frac{\mathrm{d}}{\mathrm{d}x}[g(x)] + g(x)\frac{\mathrm{d}}{\mathrm{d}x}[f(x)],$$

$$\frac{\mathrm{d}}{\mathrm{d}x}\left[\frac{f(x)}{g(x)}\right] = \frac{g(x)\frac{\mathrm{d}}{\mathrm{d}x}[f(x)] - f(x)\frac{\mathrm{d}}{\mathrm{d}x}[g(x)]}{[g(x)]^2}.$$

If functions $y = f(u)$ and $u = g(x)$ are both differentiable, then the Chain Rule states that

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}y}{\mathrm{d}u}\frac{\mathrm{d}u}{\mathrm{d}x}.$$

Taylor Expansion

The Taylor expansion of function f is given by the infinite sum

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n,$$

when it exists. Here, $f^{(n)}$ is the nth derivative of f, and $n!$ is the factorial of n. For a sufficiently small number ϵ, we can replace x and a with $x + \epsilon$ and x respectively to obtain

$$f(x + \epsilon) \approx f(x) + f'(x)\epsilon + \mathcal{O}(\epsilon^2).$$

Because ϵ is sufficiently small, the above formula can be simplified to

$$f(x + \epsilon) \approx f(x) + f'(x)\epsilon.$$

Partial Derivatives

Let $u = f(x_1, x_2, \ldots, x_n)$ be a function with n arguments. The partial derivative of u with respect to its ith parameter x_i is

$$\frac{\partial u}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, \ldots, x_{i-1}, x_i + h, x_{i+1}, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{h}.$$

The following partial derivative expressions are equivalent:

$$\frac{\partial u}{\partial x_i} = \frac{\partial f}{\partial x_i} = f_{x_i} = f_i = D_i f = D_{x_i} f.$$

To calculate $\partial u / \partial x_i$, we simply treat $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$ as constants and calculate the derivative of u with respect to x_i.

Gradients

Assume the input of function $f : \mathbb{R}^n \to \mathbb{R}$ is an n-dimensional vector $\boldsymbol{x} = [x_1, x_2, \ldots, x_n]^\top$ and the output is a scalar. The gradient of function $f(\boldsymbol{x})$ with respect to $\boldsymbol{x}$ is a vector of n partial derivatives:

$$\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = \left[\frac{\partial f(\boldsymbol{x})}{\partial x_1}, \frac{\partial f(\boldsymbol{x})}{\partial x_2}, \ldots, \frac{\partial f(\boldsymbol{x})}{\partial x_n} \right]^\top.$$

To be concise, we sometimes use $\nabla f(\boldsymbol{x})$ to replace $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$.

If $\boldsymbol{A}$ is a matrix with m rows and n columns, and $\boldsymbol{x}$ is an n-dimensional vector, the following identities

hold:

$$\nabla_{\boldsymbol{x}} \boldsymbol{A}\boldsymbol{x} = \boldsymbol{A}^\top,$$
$$\nabla_{\boldsymbol{x}} \boldsymbol{x}^\top \boldsymbol{A} = \boldsymbol{A},$$
$$\nabla_{\boldsymbol{x}} \boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{x} = (\boldsymbol{A} + \boldsymbol{A}^\top)\boldsymbol{x},$$
$$\nabla_{\boldsymbol{x}} \|\boldsymbol{x}\|^2 = \nabla_{\boldsymbol{x}} \boldsymbol{x}^\top \boldsymbol{x} = 2\boldsymbol{x}.$$

Similarly if $\boldsymbol{X}$ is a matrix, the

$$\nabla_{\boldsymbol{X}} \|\boldsymbol{X}\|_F^2 = 2\boldsymbol{X}.$$

Hessian Matrices

Assume the input of function $f : \mathbb{R}^n \to \mathbb{R}$ is an n-dimensional vector $\boldsymbol{x} = [x_1, x_2, \ldots, x_n]^\top$ and the output is a scalar. If all second-order partial derivatives of function f exist and are continuous, then the Hessian matrix $\boldsymbol{H}$ of f is a matrix with m rows and n columns given by

$$\boldsymbol{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix},$$

Here, the second-order partial derivative is evaluated

$$\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{\partial}{\partial x_i}\left(\frac{\partial f}{\partial x_j}\right).$$

12.2.3 Probability

Finally, we will briefly introduce conditional probability, expectation, and uniform distribution.

Conditional Probability

Denote the probability of event A and event B as $\mathbb{P}(A)$ and $\mathbb{P}(B)$, respectively. The probability of the simultaneous occurrence of the two events is denoted as $\mathbb{P}(A \cap B)$ or $\mathbb{P}(A, B)$. If B has non-zero probability, the conditional probability of event A given that B has occurred is

$$\mathbb{P}(A \mid B) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(B)},$$

That is,

$$\mathbb{P}(A \cap B) = \mathbb{P}(B)\mathbb{P}(A \mid B) = \mathbb{P}(A)\mathbb{P}(B \mid A).$$

If

$$\mathbb{P}(A \cap B) = \mathbb{P}(A)\mathbb{P}(B),$$

then A and B are said to be independent of each other.

Expectation

A random variable takes values that represent possible outcomes of an experiment. The expectation (or average) of the random variable X is denoted as

$$\mathbb{E}(X) = \sum_x x\mathbb{P}(X = x).$$

Uniform Distribution

Assume random variable X obeys a uniform distribution over $[a, b]$, i.e. $X \sim U(a, b)$. In this case, random variable X has the same probability of being any number between a and b.

Summary

- This section summarizes the basic knowledge of linear algebra, differentiation, and probability required to understand the contents in this book.

Exercise

- Find the gradient of function $f(\boldsymbol{x}) = 3x_1^2 + 5e^{x_2}$.

Scan the QR Code to Discuss

12.3 Using Jupyter Notebook

This section describes how to edit and run the code in this book using Jupyter Notebook. Make sure you have installed Jupyter Notebook and obtained the code for this book according to the steps in the Acquiring and Running Codes in This Book section.

12.3.1 Edit and Run the Code in This Book Locally

Now we describe how to use Jupyter Notebook to edit and run code of the book locally. Suppose that the local path of code of the book is xx/yy/d2l-en-1.0/. Change directory to this path in command mode (`cd xx/yy/d2l-en-1.0`), then run command `jupyter notebook`. Now open http://localhost:8888 (usually automatically opened) in the browser, and you will see the interface of Jupyter Notebook and all the folders containing code of the book, as shown in Figure 11.1.
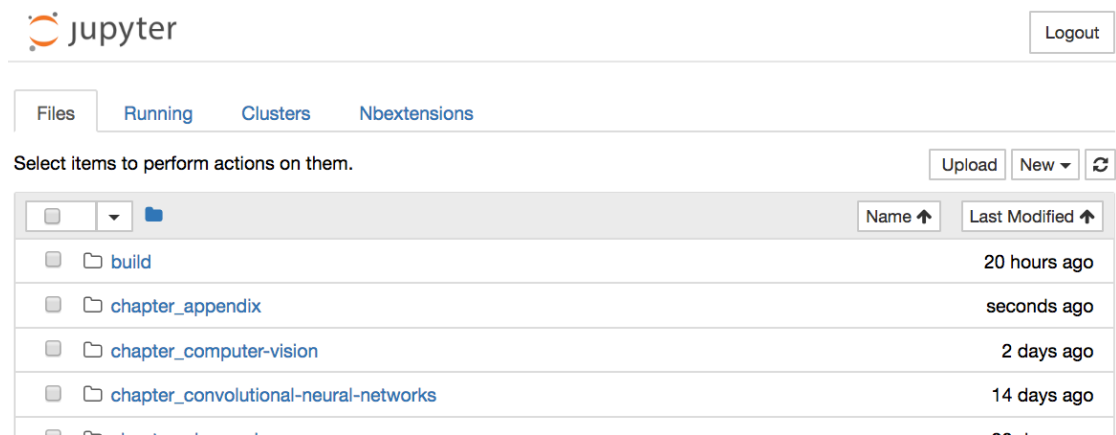
Fig. 12.1: The folders containing the code in this book.

You can access the notebook files by clicking on the folder displayed on the webpage. They usually have the suffix ipynb. For the sake of brevity, we create a temporary test.ipynb file, and the content displayed after you click it is as shown in Figure 11.2. This notebook includes a markdown cell and code cell. The content in the markdown cell includes This is A Title and This is text. The code cell contains two lines of Python code.
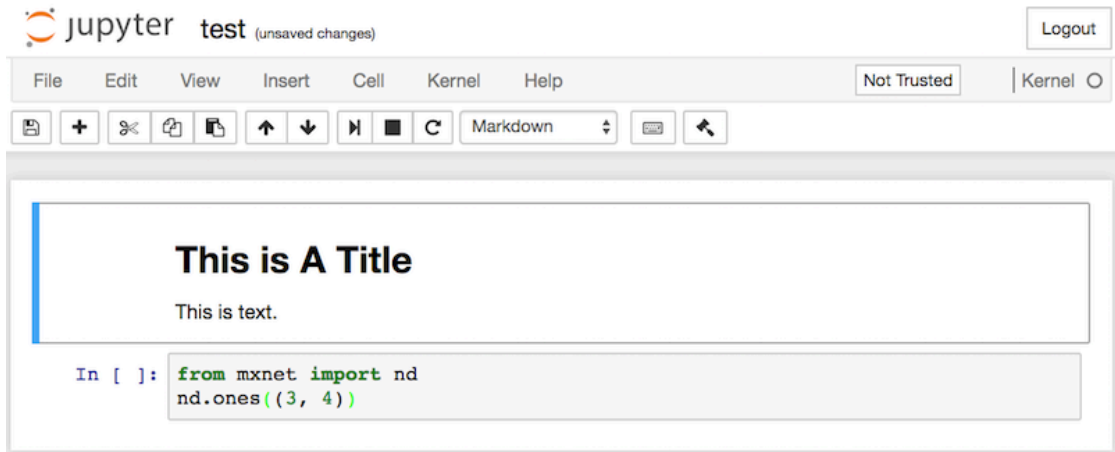
Fig. 12.2: Markdown and code cells in the text.ipynb file.

Double click on the markdown cell, to enter edit mode. Add a new text string Hello world. at the end of the cell, as shown in Figure 11.3.

Fig. 12.3: Edit the markdown cell.

As shown in Figure 11.4, click Cell → Run Cells in the menu bar to run the edited cell.
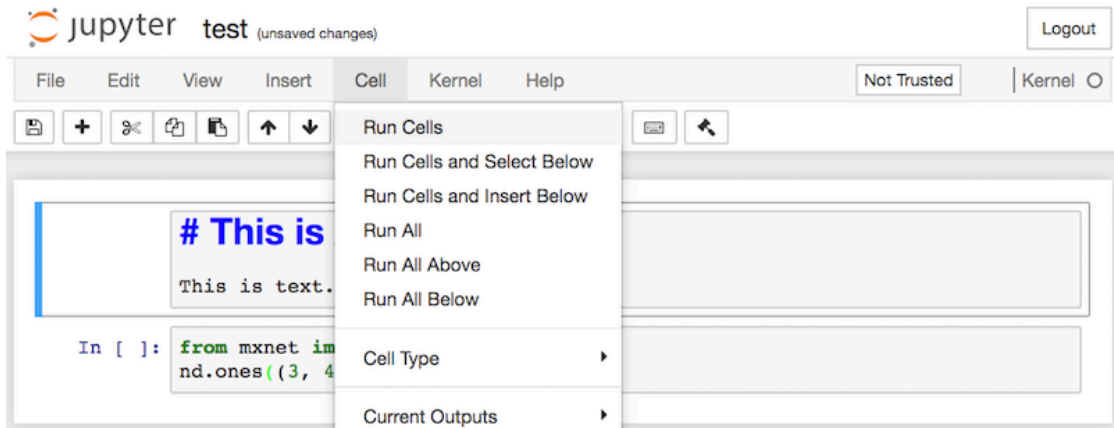
Fig. 12.4: Run the cell.

After running, the markdown cell is as shown in Figure 11.5.
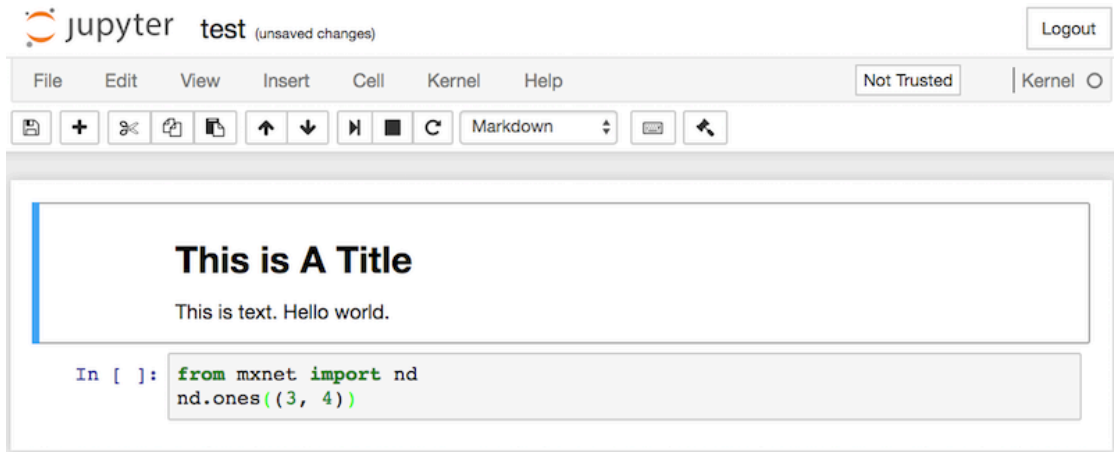
Fig. 12.5: The markdown cell after editing.

Next, click on the code cell. Add the multiply by 2 operation `*  2` after the last line of code, as shown in Figure 11.6.
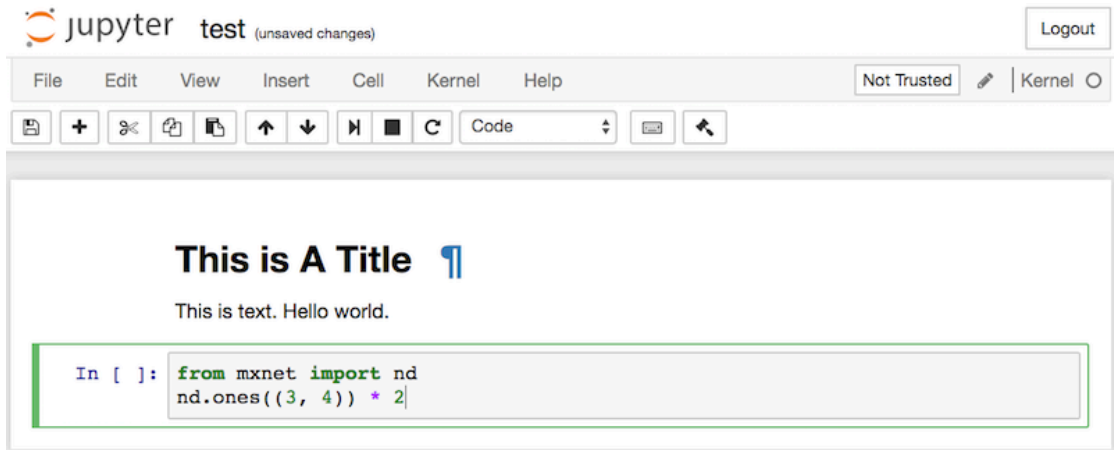
Fig. 12.6: Edit the code cell.

You can also run the cell with a shortcut (Ctrl + Enter by default) and obtain the output result from Figure 11.7.
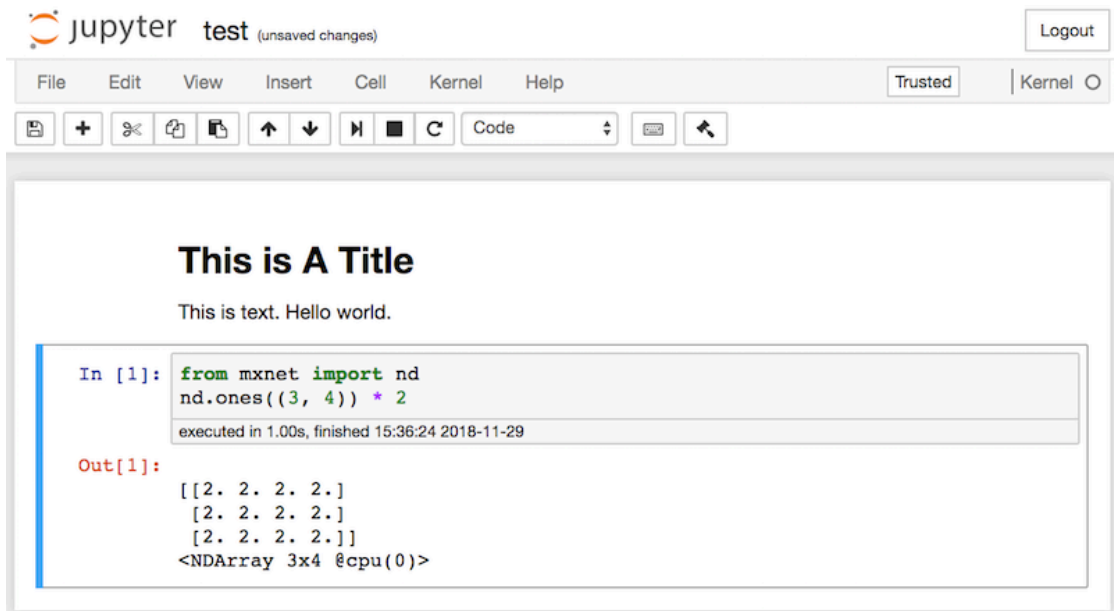
Fig. 12.7: Run the code cell to obtain the output.

When a notebook contains more cells, we can click Kernel → Restart & Run All in the menu bar to run

all the cells in the entire notebook. By clicking Help $\rightarrow$ Edit Keyboard Shortcuts in the menu bar, you can edit the shortcuts according to your preferences.

12.3.2 Advanced Options

Below are some advanced options for using Jupyter Notebook. You can use this section as a reference based on your interests.

Read and Write GitHub Source Files with Jupyter Notebook

If you wish to contribute to the content of this book, you need to modify the source file (.md file, not .ipynb file) in the markdown format on GitHub. With the notedown plugin, we can use Jupyter Notebook to modify and run the source code in markdown format. Linux/MacOS users can execute the following commands to obtain the GitHub source files and activate the runtime environment.

```
git clone https://github.com/d2l-ai/d2l-en.git
cd d2l-en
conda env create -f environment.yml
source activate gluon # Windows users run "activate gluon"
```

Next, install the notedown plugin, run Jupyter Notebook, and load the plugin:

```
pip install https://github.com/mli/notedown/tarball/master
jupyter notebook --NotebookApp.contents_manager_class='notedown.NotedownContentsManager
 ↪'
```

If you want to turn on the notedown plugin by default each time you run Jupyter Notebook, you follow the procedure below.

First, execute the following command to generate a Jupyter Notebook configuration file (if it has already been generated, you can skip this step).

```
jupyter notebook --generate-config
```

Then, add the following line to the end of the Jupyter Notebook configuration file (for Linux/macOS, usually in the path ~/.jupyter/jupyter_notebook_config.py):

```
c.NotebookApp.contents_manager_class = 'notedown.NotedownContentsManager'
```

After that, you only need to run the `jupyter notebook` command to turn on the notedown plugin by default.

Run Jupyter Notebook on a Remote Server

Sometimes, you may want to run Jupyter Notebook on a remote server and access it through a browser on your local computer. If Linux or MacOS is installed on you local machine (Windows can also support

this function through third-party software such as PuTTY), you can use port mapping:

```
ssh myserver -L 8888:localhost:8888
```

The above is the address of the remote server `myserver`. Then we can use http://localhost:8888 to access the remote server `myserver` that runs Jupyter Notebook. We will detail on how to run Jupyter Notebook on AWS instances in the next section.

Operation Timing

We can use the ExecutionTime plugin to time the execution of each code cell in a Jupyter notebook. Use the following commands to install the plugin:

```
pip install jupyter_contrib_nbextensions
jupyter contrib nbextension install --user
jupyter nbextension enable execute_time/ExecuteTime
```

Summary

- You can edit and run the code in this book using Jupyter Notebook.

Exercise

- Try to edit and run the code in this book locally.

Scan the QR Code to Discuss

12.4 Using AWS to Run Code

If your local machine has limited computing resources, you can use cloud computing services to obtain more powerful computing resources and use them to run the deep learning code in this document. In this section, we will show you how to apply for instances and use Jupyter Notebook to run code on AWS (Amazon's cloud computing service). The example here includes two steps:

1. Apply for a K80 GPU p2.xlarge instance.

---

2. Install CUDA and the corresponding MXNet GPU version.

The process to apply for other instance types and install other MXNet versions is basically the same as that described here.

12.4.1 Apply for an Account and Log In

First, we need to register an account at https://aws.amazon.com/. It usually requires a credit card.

After logging into your AWS account, click EC2 (marked by the red box in Figure 11.8) to go to the EC2 panel.
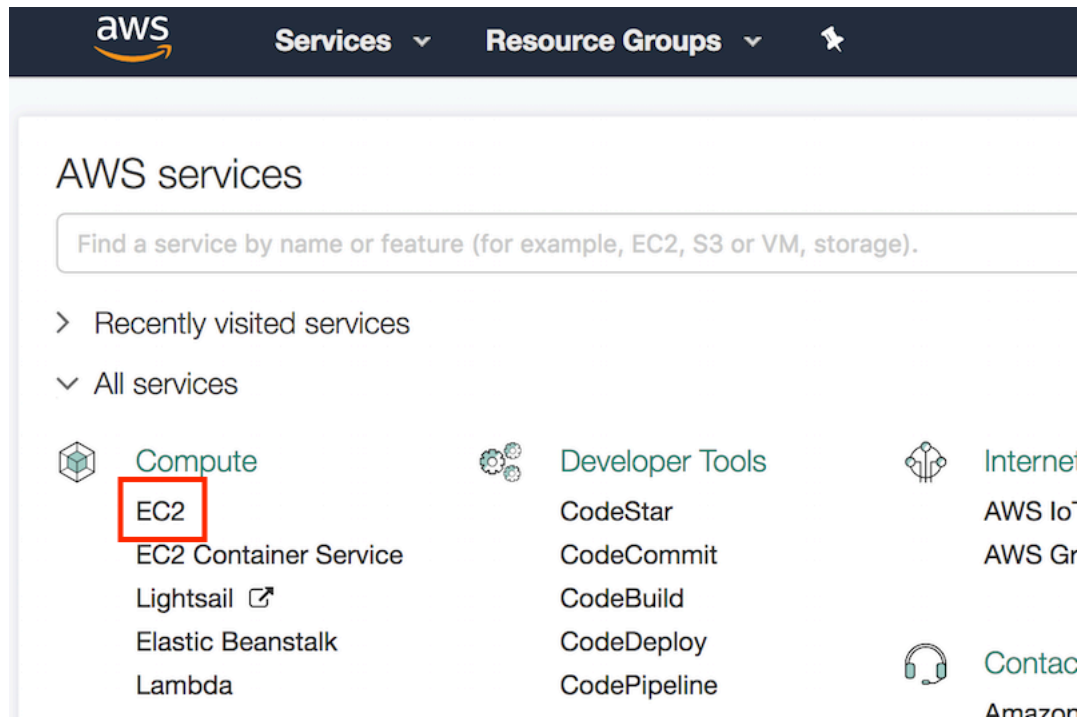
Fig. 12.8: Log into your AWS account.

12.4.2 Create and Run an EC2 Instance

Figure 11.9 shows the EC2 panel. In the area marked by the red box in Figure 11.9, select a nearby data center to reduce latency. If you are located in China you can select a nearby Asia Pacific region, such as Asia Pacific (Seoul). Please note that some data centers may not have GPU instances. Click the Launch Instance button marked by the red box in Figure 11.8 to launch your instance.
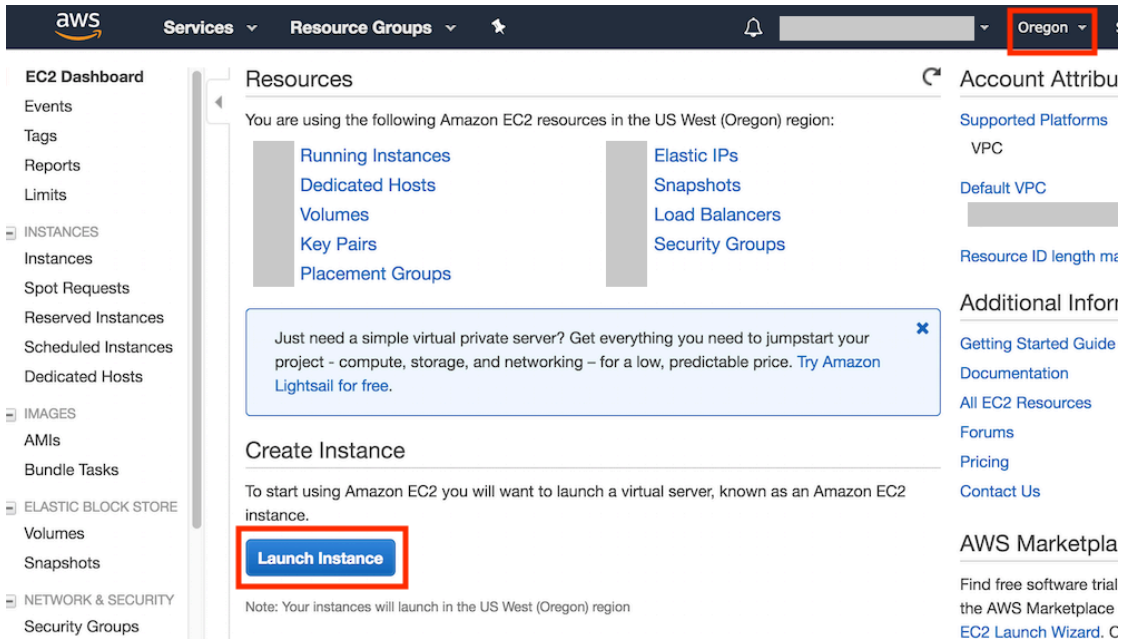
Fig. 12.9: EC2 panel.

The row at the top of Figure 11.10 shows the seven steps in the instance configuration process. In the first step 1. Choose AMI, choose Ubuntu 16.04 for the operating system.
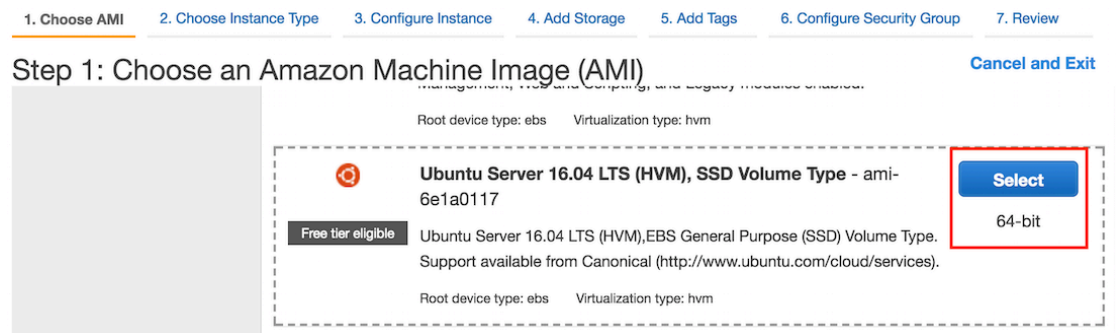
Fig. 12.10: Choose an operating system.

EC2 provides many different instance configurations to choose from. As shown in Figure 11.11, In Step 2: Choose an Instance Typechoose a p2.xlarge instance with K80 GPU. We can also choose instances with multiple GPUs such as p2.16xlarge. If you want to compare machine configurations and fees of different instances, you may refer to https://www.ec2instances.info/.

Fig. 12.11: Choose an instance.

Before choosing an instance, we suggest you check if there are quantity restrictions by clicking the Limits label in the bar on the, as left shown in Figure 11.9. As shown in Figure 11.12, this account can only open one p2.xlarge instance per region. If you need to open more instances, click on the Request limit increase link to apply for a higher instance quota. Generally, it takes one business day to process an application.

Fig. 12.12: Instance quantity restrictions.

In this example, we keep the default configurations for the steps 3. Configure Instance, 5. Add Tags, and 6. Configure Security Group. Tap on 4. Add Storage and increase the default hard disk size to 40 GB. Note that you will need about 4 GB to install CUDA.

Step 4: Add Storage

Volume Type ⓘ	Device ⓘ	Snapshot ⓘ	Size (GiB) ⓘ	Volume Type ⓘ	IOPS ⓘ	Th (M
Root	/dev/sda1	snap-	40	General Purpose S ⬍	100 / 3000	N/

Add New Volume

Fig. 12.13: Modify instance hard disk size.

Finally, go to 7. Review and click Launch to launch the configured instance. The system will now prompt you to select the key pair used to access the instance. If you do not have a key pair, select Create a new key pair in the first drop-down menu in Figure 11.14 to generate a key pair. Subsequently, you can select Choose an existing key pair for this menu and then select the previously generated key pair. Click Launch Instances to launch the created instance.
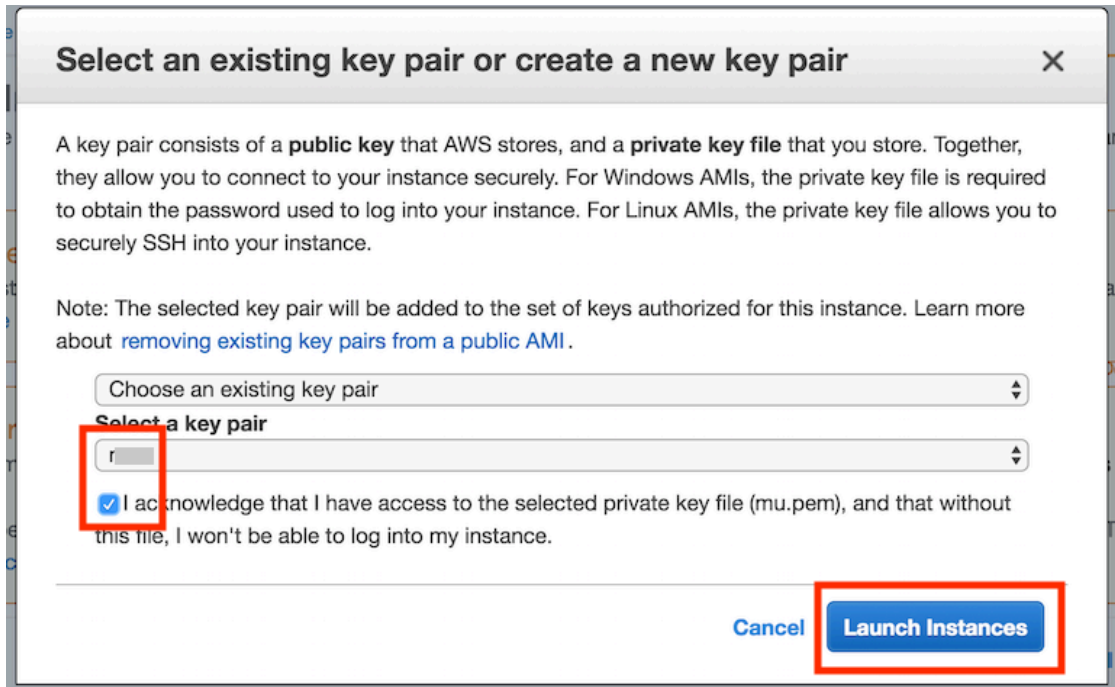
Fig. 12.14: Select a key pair.

Click the instance ID shown in Figure 11.15 to view the status of this instance.
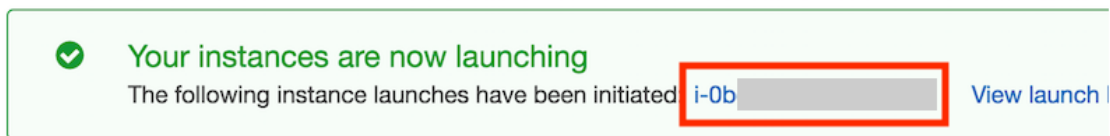
Fig. 12.15: C lick the instance ID.

As shown in Figure 11.16, after the instance state turns green, right-click the instance and select Connect to view the instance access method. For example, enter the following in the command line:

```
ssh -i "/path/to/key.pem" ubuntu@ec2-xx-xxx-xxx-xxx.y.compute.amazonaws.com
```

Here, /path/to/key.pem is the path of the locally-stored key used to access the instance. When the com-

mand line prompts Are you sure you want to continue connecting (yes/no), enter yes and press Enter to log into the instance.
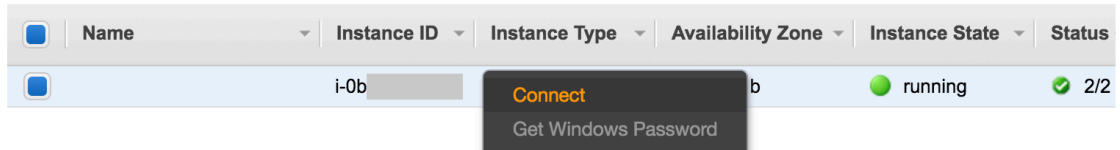
Fig. 12.16: View instance access and startup method.

12.4.3 Install CUDA

If you log into a GPU instance, you need to download and install CUDA. First, update and install the package needed for compilation.

```
sudo apt-get update && sudo apt-get install -y build-essential git libgfortran3
```

NVIDIA releases a major version of CUDA every year. Here we download the latest CUDA 9.0 when the book is written. Visit the official website of NVIDIA (https://developer.nvidia.com/cuda-90-download-archive) to obtain the download link of CUDA 9.0, as shown in Figure 11.17.

Fig. 12.17: Find the CUDA 9.0 download address.

After finding the download address, download and install CUDA 9.0. For example:

```
# The download link and file name are subject to change, so always use those
# from the NVIDIA website
wget https://developer.nvidia.com/compute/cuda/9.0/Prod/local_installers/cuda_9.0.176_
↪384.81_linux-run
sudo sh cuda_9.0.176_384.81_linux-run
```

Press Ctrl+C to jump out of the document and answer the following questions.

```
Do you accept the previously read EULA?
accept/decline/quit: accept
Install NVIDIA Accelerated Graphics Driver for Linux-x86_64 384.81?
(y)es/(n)o/(q)uit: y
Do you want to install the OpenGL libraries?
(y)es/(n)o/(q)uit [ default is yes ]: y
Do you want to run nvidia-xconfig?
This will ... vendors.
(y)es/(n)o/(q)uit [ default is no ]: n
Install the CUDA 9.0 Toolkit?
(y)es/(n)o/(q)uit: y
Enter Toolkit Location
 [ default is /usr/local/cuda-9.0 ]:
Do you want to install a symbolic link at /usr/local/cuda?
(y)es/(n)o/(q)uit: y
Install the CUDA 9.0 Samples?
(y)es/(n)o/(q)uit: n
```

After installing the program, run the following command to view the instance GPU.

```
nvidia-smi
```

Finally, add CUDA to the library path to help other libraries find it.

```
echo "export LD_LIBRARY_PATH=\${LD_LIBRARY_PATH}:/usr/local/cuda-9.0/lib64" >> ~/.
↪bashrc
```

12.4.4 Acquire the Code for this Book and Install MXNet GPU Version

We have introduced the way to obtaining code of the book and setting up the running environment in Section Getting started with Gluon. First, install Miniconda of the Linux version (website: https://conda.io/miniconda.html), such as

```
# The download link and file name are subject to change, so always use those
# from the Miniconda website
wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
sudo sh Miniconda3-latest-Linux-x86_64.sh
```

Now, you need to answer the following questions:

```
Do you accept the license terms? [yes|no]
[no] >>> yes
Do you wish the installer to prepend the Miniconda3 install location
to PATH in your /home/ubuntu/.bashrc ? [yes|no]
[no] >>> yes
```

After installation, run `source ~/.bashrc` once to activate CUDA and Conda. Next, download the code for this book and install and activate the Conda environment.

```
mkdir d2l-en && cd d2l-en
curl https://www.d2l.ai/d2l-en-1.0.zip -o d2l-en.zip
unzip d2l-en.zip && rm d2l-en.zip
conda env create -f environment.yml
source activate gluon
```

The MXNet CPU version is installed in the environment by default. Now, you must replace it with the MXNet GPU version. As the CUDA version is 9.0, install `mxnet-cu90`. Generally speaking, if your CUDA version is x.y, the corresponding MXNET version is `mxnet-cuxy`.

```
pip uninstall mxnet
# X.Y.Z should be replaced with the version number depended on by the book
pip install mxnet-cu90==X.Y.Z
```

12.4.5 Run Jupyter Notebook

Now, you can run Jupyter Notebook:

```
jupyter notebook
```

Figure 11.18 shows the possible output after you run Jupyter Notebook. The last row is the URL for port 8888.

Fig. 12.18: Output after running Jupyter Notebook. The last row is the URL for port 8888.

Because the instance you created does not expose port 8888, you can launch SSH in the local command line and map the instance to the local port 8889.

```
# This command must be run in the local command line
ssh -i "/path/to/key.pem" ubuntu@ec2-xx-xxx-xxx-xxx.y.compute.amazonaws.com -L
↪8889:localhost:8888
```

Finally, copy the URL shown in the last line of the Jupyter Notebook output in Figure 11.18 to your local browser and change 8888 to 8889. Press Enter to use Jupyter Notebook to run the instance code from your local browser.

12.4.6 Close Unused Instances

As cloud services are billed by use duration, you will generally want to close instances you no longer use.

If you plan on restarting the instance after a short time, right-click on the example shown in Figure 11.16 and select Instance State → Stop to stop the instance. When you want to use it again, select Instance State → Start to restart the instance. In this situation, the restarted instance will retain the information stored on its hard disk before it was stopped (for example, you do not have to reinstall CUDA and other runtime environments). However, stopped instances will still be billed a small amount for the hard disk space retained.

If you do not plan to use the instance again for a long time, right-click on the example in Figure 11.16 and select Image → Create to create an image of the instance. Then, select Instance State → Terminate to terminate the instance (it will no longer be billed for hard disk space). The next time you want to use this instance, you can follow the steps for creating and running an EC2 instance described in this section to create an instance based on the saved image. The only difference is that, in 1. Choose AMI shown in Figure 11.10, you must use the My AMIs option on the left to select your saved image. The created instance will retain the information stored on the image hard disk. For example, you will not have to reinstall CUDA and other runtime environments.

Summary

- You can use cloud computing services to obtain more powerful computing resources and use them to run the deep learning code in this document.

Exercise

- The cloud offers convenience, but it does not come cheap. Research the prices of cloud services and find ways to reduce overhead.

12.5 GPU Purchase Guide

Deep learning training generally requires large volumes of computing resources. Currently, GPUs are the most common computation acceleration hardware used for deep learning. Compared with CPUs, GPUs are cheaper and provide more intensive computing. On the one hand, GPUs can deliver the same compute power at a tenth of the price of CPUs. On the other hand, a single sever can generally support 8 or 16 GPUs. Therefore, the GPU quantity can be viewed as a standard to measure the deep learning compute power of a server.

12.5.1 Selecting a GPU

At present, AMD and NVIDIA are the two main manufacturers of dedicated GPUs. NVIDIA was the first to enter the deep learning field and provides better support for deep learning frameworks. Therefore, most buyers choose NVIDIA GPUs.

NVIDIA provides two types of GPUs, targeting individual uses (such as the GTX series) and enterprise users (such as the Tesla series). The two types of GPUs provide comparable compute power. However, the enterprise user GPUs generally use passive heat dissipation and add a memory check function. Therefore, these GPUs are more suitable for data centers and usually cost ten times more than individual user GPUs.

If you are a large company with 100 or more servers, you should consider the NVIDIA Tesla series for enterprise users. If you are a lab or small to mid-size company with 10 to 100 servers, you should consider the NVIDIA DGX series if your budget is sufficient. Otherwise, you can consider more cost-effective servers, such as Supermicro, and then purchase and install GTX series GPUs.

NVIDIA generally releases a new GPU version every one or two years, such as the GTX 1000 series released in 2017. Each series offers several different models that provide different performance levels.

GPU performance is primarily a combination of the following three parameters:

1. Compute power: Generally we look for 32-bit floating-point compute power. 16-bit floating point training is also entering the mainstream. If you are only interested in prediction, you can also use 8-bit integer.

2. Memory size: As your models become larger or the batches used during training grow bigger, you will need more GPU memory.

3. Memory bandwidth: You can only get the most out of your compute power when you have sufficient memory bandwidth.

For most users, it is enough to look at compute power. The GPU memory should be no less than 4 GB. However, if the GPU must simultaneously display graphical interfaces, we recommend a memory size of at least 6 GB. There is generally not much variation in memory bandwidth, with few options to choose from.

Figure 11.19 compares the 32-bit floating-point compute power and price of the various GTX 900 and 1000 series models. The prices are the suggested prices found on Wikipedia.
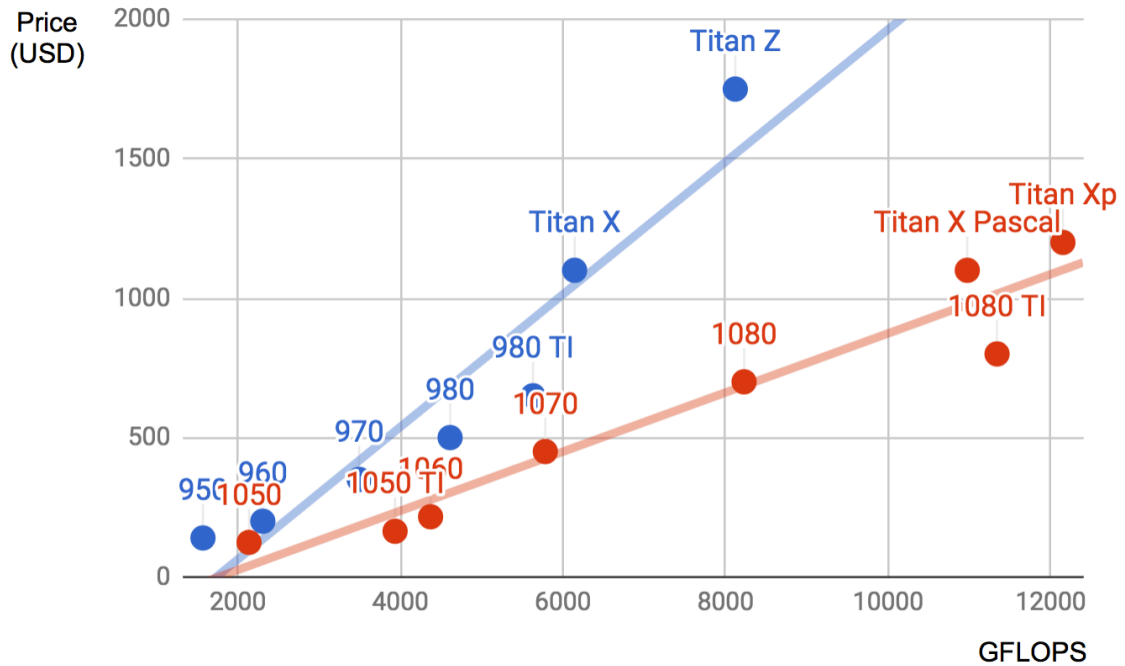
Fig. 12.19: Floating-point compute power and price comparison.

From Figure 11.19, we can see two things:

1. Within each series, price and performance are roughly proportional. However, the newer models offer better cost effectiveness, as can be seen by comparing the 980 Ti and 1080 Ti.

2. The performance to cost ratio of the GTX 1000 series is about two times greater than the 900 series.

If we look at the earlier GTX series, we will observe a similar pattern. Therefore, we recommend you buy the latest GPU model in your budget.

12.5.2 Machine Configuration

Generally, GPUs are primarily used for deep learning training. Therefore, you do not have to purchase high-end CPUs. When deciding on machine configurations, you can find a mid to high-end configuration based on recommendations on the Internet. However, given the power consumption, heat dissipation performance, and size of GPUs, you need to consider three additional factors in machine configurations.

1. Chassis size: GPUs are relatively large, so you should look for a large chassis with a built-in fan.

2. Power source: When purchasing GPUs, you must check the power consumption, as they can range from 50 W to 300 W. When choosing a power source, you must ensure it provides sufficient power and will not overload the data center power supply.

3. Motherboard PCIe card slot: We recommend PCIe 3.0 16x to ensure sufficient bandwidth between the GPU and main memory. If you mount multiple GPUs, be sure to carefully read the motherboard description to ensure that 16x bandwidth is still available when multiple GPUs are used at the same time. Be aware that some motherboards downgrade to 8x or even 4x bandwidth when 4 GPUs are mounted.

Summary

- You should purchase the latest GPU model that you can afford.

- When deciding on machine configurations, you must consider GPU power consumption, heat dissipation, and size.

Exercise

- You can browse the discussions about machine configurations in the forum for this section.

Scan the QR Code to Discuss

12.6 How to Contribute to This Book

You can view the list of contributors [1] for this open-source book. If you want to contribute, you need to install Git and submit a pull request [2] to the book's GitHub code repository. When your pull request is

merged into the code repository by the author, you will become a contributor.

This section describes the basic Git procedure for contributing to this book. If you are familiar with Git operations, you can skip this section.

In the procedure below, we assume that the contributor's GitHub ID is astonzhang.

Step 1: Install Git. The Git open source book details how to install Git [3]. If you do note have a GitHub account, you need to sign up for one[4].

Step 2: Log in to GitHub. Enter the address of the book's code repository in your browser [2]. Click on the Fork button in the red box at the top-right of Figure 11.20 to clone (get a copy of) the code repository of this book.
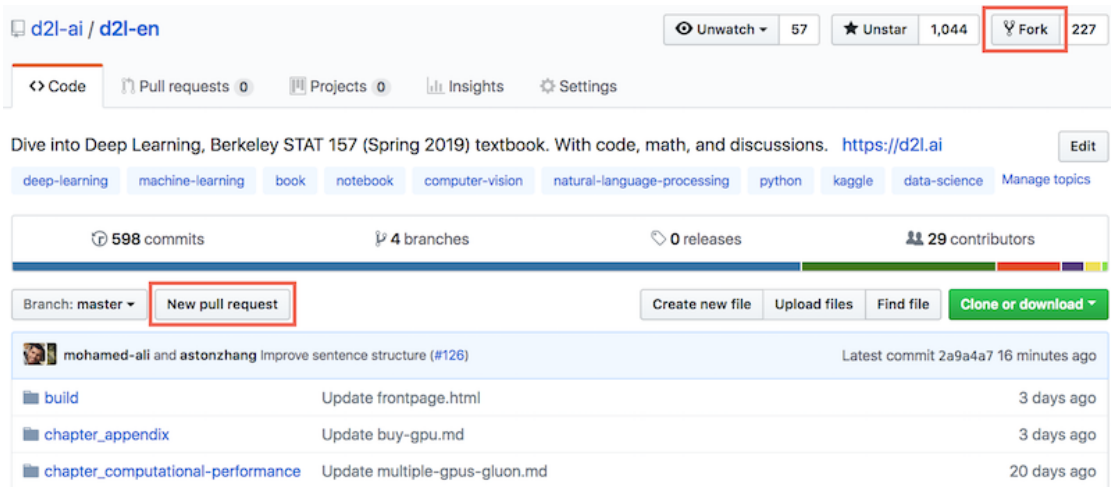
Fig. 12.20: The code repository page.

Now, the code repository of this book will be copied to your username, such as Your GitHub ID/d2l-en shown at the top-left of Figure 11.21.
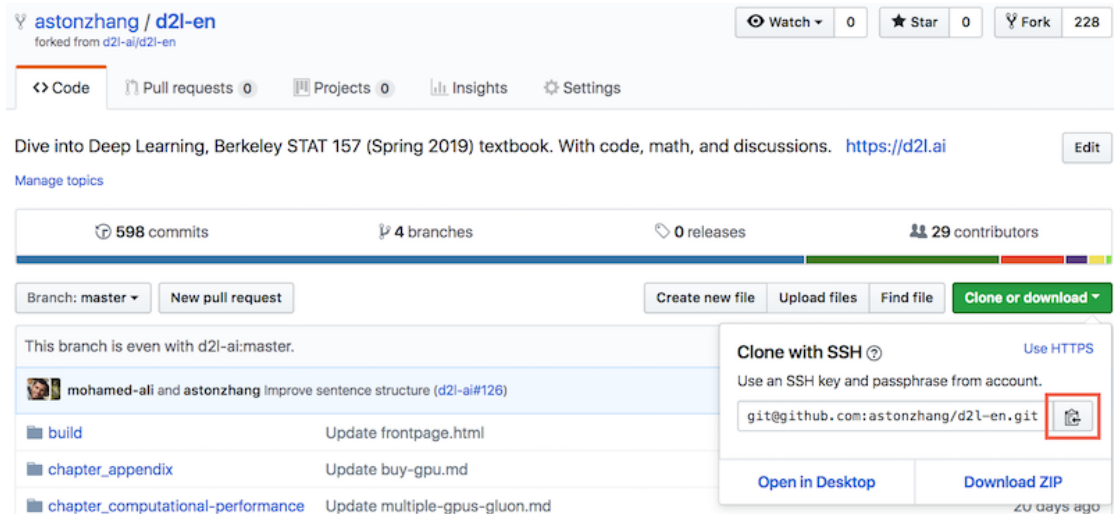
Fig. 12.21: Copy the code repository.

Step 3: Click the green Clone or download button on the right side of Figure 11.21 and click the button in the red box to copy the code repository address under your username. Follow the method described in the Acquiring and Running Codes in This Book section to enter command line mode. Here, we assume you want to save the code repository under the local ~/repo path. Go to this path, type `git clone`, and paste the code repository address under your username. Execute the command:

```
# Replace your_Github_ID with your GitHub username
git clone https://github.com/your_Github_ID/d2l-en.git
```

Now, all the files in the code repository of this book will be saved in the local ~/repo/d2l-en path.

Step 4: Edit the code repository of this book under the local path. Assume we have modified a typo in the file `~/repo/d2l-en/chapter_deep-learning-basics/linear-regression.md`. In command line mode, enter the path `~/repo/d2l-en` and execute the command:

```
git status
```

At this point Git will prompt that the chapter_deep-learning-basics/linear-regression.md file has been modified, as shown in Figure 11.22.

Fig. 12.22: Git prompts that the chapter_deep-learning-basics/linear-regression.md file has been modi-fied.

After confirming the file submitting the change, execute the following command:

```
git add chapter_deep-learning-basics/linear-regression.md
git commit -m 'fix typo in linear-regression.md'
git push
```

Here, `'fix typo in linear-regression.md'` is the description of the submitted change. You can replace this with other meaningful descriptive information.

Step 5: Enter the code repository address of this book [2] in your browser again. Click the New pull request button in the red box on the bottom-left of Figure 11.20. On the page that appears, click the compare across forks link in the red box on the right side of Figure 11.23. Then, click the head fork: d2l-ai/d2l-en button in the red box below. Enter your GitHub ID in the pop-up text box and select Your GitHub-ID/d2l-en from the drop-down menu, as shown in Figure 11.23.
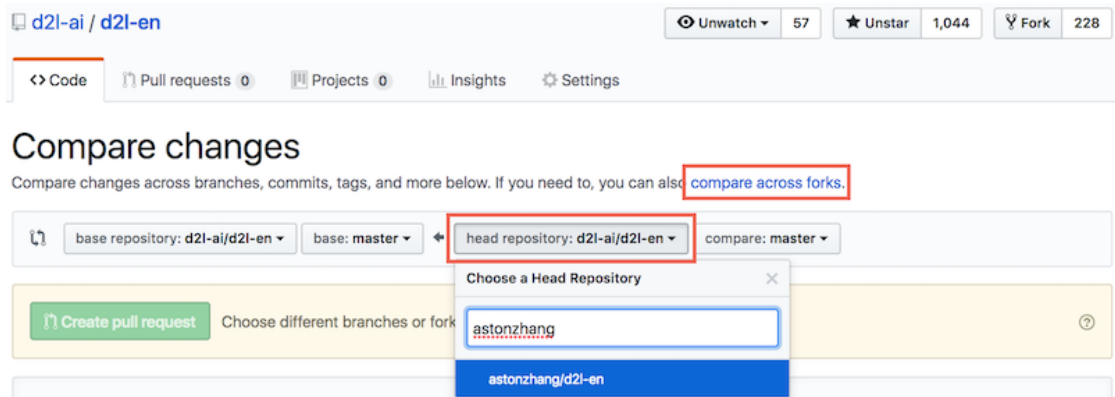
Fig. 12.23: Select the code repository where the source of the change is located.

Step 6: As shown in Figure 11.24, describe the pull request you want to submit in the title and body text boxes. Click the green Create pull request button in the red box to submit the pull request.
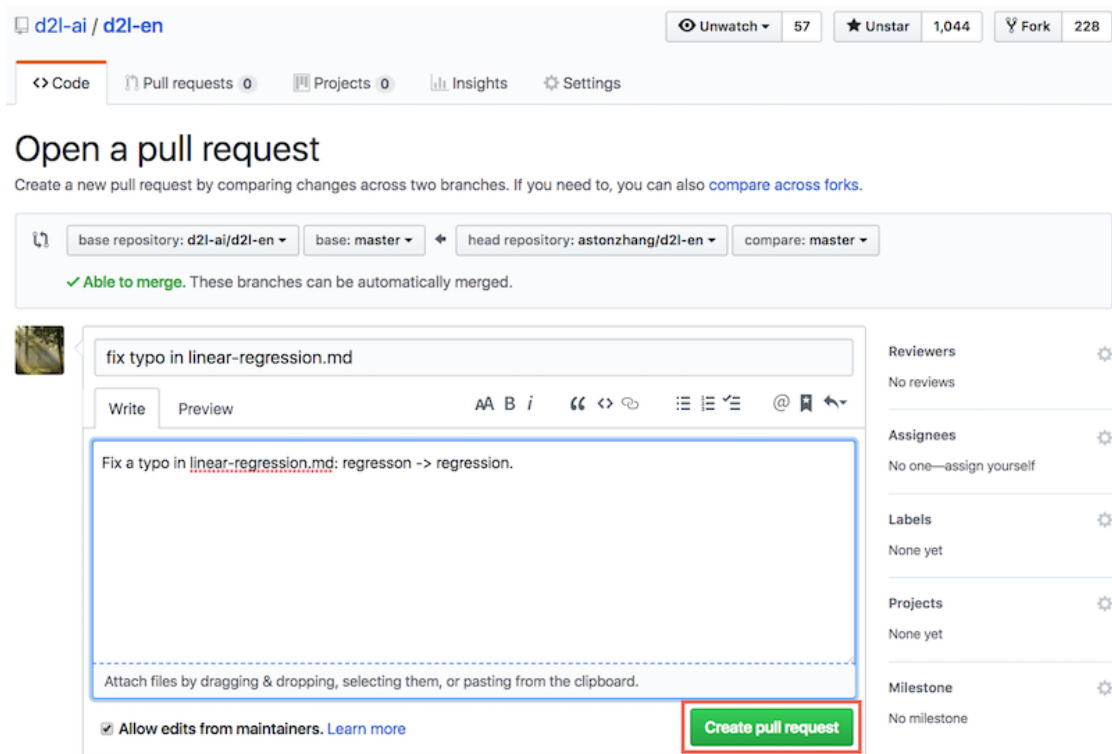
Fig. 12.24: Describe and submit a pull request.

After submitting the request, you will see the page shown in Figure 11.25, which indicates that the pull request has been submitted.
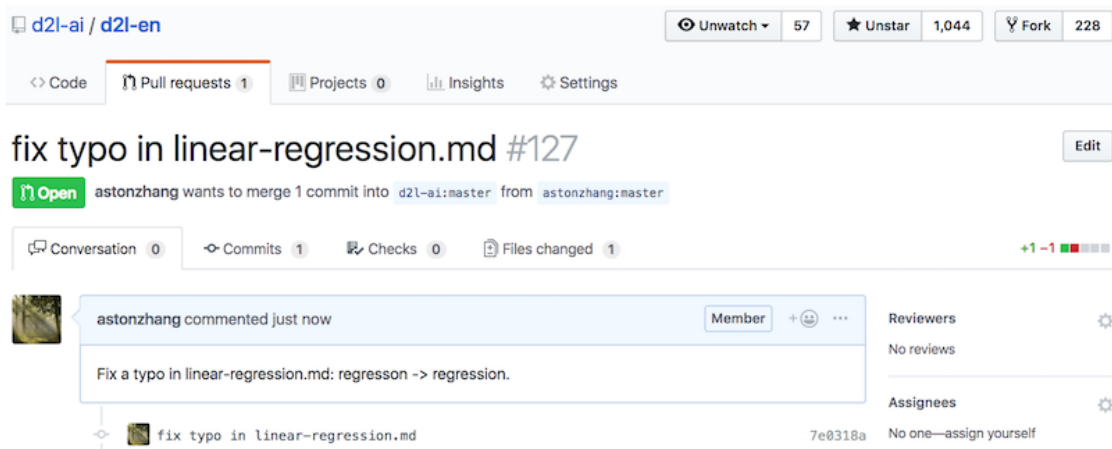
Fig. 12.25: The pull request has been submitted.

Summary

- You can use GitHub to contribute to this book.

Exercise

- If you feel that some parts of the book can be improved, try submitting a pull request.

References

[1] List of contributors to this book. https://github.com/d2l-ai/d2l-en/graphs/contributors

[2] Address of the code repository of this book. https://github.com/d2l-ai/d2l-en

[3] Install Git. https://git-scm.com/book/zh/v2

[4] URL of GitHub. https://github.com/

Scan the QR Code to Discuss

12.7 `d2l` Package Index

Name of Function or Class	Relevant Chapter
`bbox_to_rect`	*Object Detection and Bounding Boxes*
`Benchmark`	*Asynchronous Computation*
`corr2d`	*Two-dimensional Convolutional Layer*
`count_tokens`	*Sentiment Classification: Using Recurrent Neural Networks*
`data_iter`	Linear Regression Implementation from Scratch
`data_iter_consecutive`	*Language Model Data Set (Jay Chou album lyrics)*
`data_iter_random`	*Language Model Data Set (Jay Chou album lyrics)*
`download_imdb`	*Sentiment Classification: Using Recurrent Neural Networks*
`download_voc_pascal`	*Semantic Segmentation and Data Sets*
`evaluate_accuracy`	*Image Augmentation*
`get_data_ch7`	*Mini-batch Stochastic Gradient Descent*
`get_fashion_mnist_labels`	Image Classification Data Set(Fashion-MNIST)
`get_tokenized_imdb`	*Sentiment Classification: Using Recurrent Neural Networks*
`get_vocab_imdb`	*Sentiment Classification: Using Recurrent Neural Networks*
`grad_clipping`	*Recurrent Neural Network Implementation from Scratch*
`linreg`	Linear Regression Implementation from Scratch
`load_data_fashion_mnist`	*Deep Convolutional Neural Networks (AlexNet)*
`load_data_jay_lyrics`	*Language Model Data Set (Jay Chou album lyrics)*
`load_data_pikachu`	*Object Detection Data Set(Pikachu)*
`plt`	Linear Regression Implementation from Scratch
`predict_rnn`	*Recurrent Neural Network Implementation from Scratch*
`predict_rnn_gluon`	*Concise Implementation of Recurrent Neural Networks*
`predict_sentiment`	*Sentiment Classification: Using Recurrent Neural Networks*
`preprocess_imdb`	*Sentiment Classification: Using Recurrent Neural Networks*
`read_imdb`	*Sentiment Classification: Using Recurrent Neural Networks*
`read_voc_images`	*Semantic Segmentation and Data Sets*
`Residual`	*Residual Networks (ResNet)*
`resnet18`	*Concise Implementation of Multi-GPU Computing*
`RNNModel`	*Concise Implementation of Recurrent Neural Networks*
`semilogy`	Model Selection
`set_figsize`	Linear Regression Implementation from Scratch
`sgd`	Linear Regression Implementation from Scratch
`show_bboxes`	*Anchor Boxes*
`show_fashion_mnist`	Image Classification Data Set(Fashion-MNIST)
`show_images`	*Image Augmentation*
`show_trace_2d`	*Gradient Descent and Stochastic Gradient Descent*
`squared_loss`	Linear Regression Implementation from Scratch
`to_onehot`	*Recurrent Neural Network Implementation from Scratch*
`train`	*Image Augmentation*

Name of Function or Class	Relevant Chapter
`train_2d`	*Gradient Descent and Stochastic Gradient Descent*
`train_and_predict_rnn`	*Recurrent Neural Network Implementation from Scratch*
`train_and_predict_rnn_gluon`	*Concise Implementation of Recurrent Neural Networks*
`train_ch3`	Softmax Regression Implementation from Scratch
`train_ch5`	*Convolutional Neural Networks (LeNet)*
`train_ch7`	*Mini-batch Stochastic Gradient Descent*
`train_gluon_ch7`	*Mini-batch Stochastic Gradient Descent*
`try_all_gpus`	*Image Augmentation*
`try_gpu`	*Convolutional Neural Networks (LeNet)*
`use_svg_display`	Linear Regression Implementation from Scratch
`VOC_CLASSES`	*Semantic Segmentation and Data Sets*
`VOC_COLORMAP`	*Semantic Segmentation and Data Sets*
`voc_label_indices`	*Semantic Segmentation and Data Sets*
`voc_rand_crop`	*Semantic Segmentation and Data Sets*
`VOCSegDataset`	*Semantic Segmentation and Data Sets*