Deep Learning with PyTorch

Essential Excerpts

Eli Stevens
Luca Antiga

MANNING

Deep Learning with PyTorch
Essential Excerpts

Eli Stevens and Luca Antiga

Manning Author Picks

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: Erin Twohey, corp-sales@manning.com

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haimes

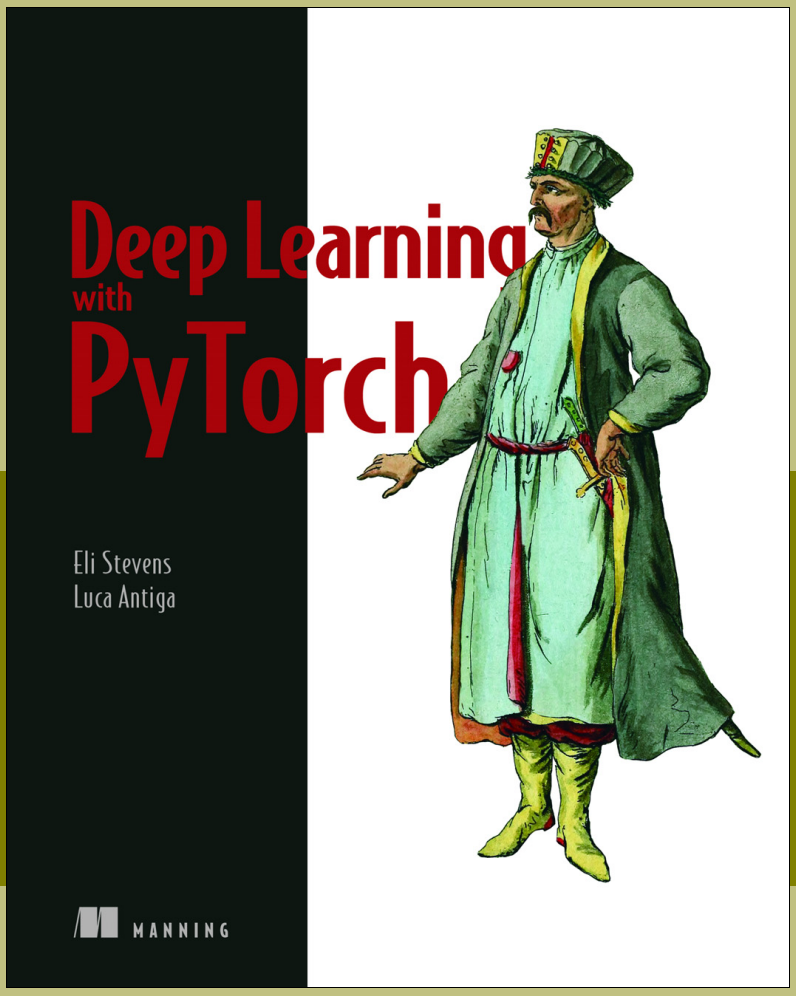
ISBN: 9781617297120
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 - EBM - 24 23 22 21 20 19

contents

about the authors

Eli Stevens has worked in Silicon Valley for the past 15 years as a software engineer, and the past 7 years as Chief Technical Officer of a startup making medical device software.

Luca Antiga is co-founder and CEO of an AI engineering company located in Bergamo, Italy, and a regular contributor to PyTorch.



Deep Learning with PyTorch

by Eli Stevens and Luca Antiga

ISBN 9781617295263
400 pages (estimated)
$49.99
Publication in Winter, 2019 (estimated)

1 Introducing deep learning and the PyTorch library

This chapter covers

- What this book will teach you
- PyTorch's role as a library for building deep learning projects
- The strengths and weaknesses of PyTorch
- The hardware you'll need to follow along with the examples

We're living through exciting times. The landscape of what computers can do is changing by the week. Tasks that only a few years ago were thought to require higher cognition are getting solved by machines at near-superhuman levels of performance. Tasks such as describing a photographic image with a sentence in idiomatic English, playing complex strategy game, and diagnosing a tumor from a radiological scan are all approachable now by a computer. Even more impressively, computers acquire the ability to solve such tasks through examples, rather than human-encoded of handcrafted rules.

It would be disingenuous to assert that machines are learning to "think" in any human sense of the word. Rather, a general class of algorithms is able to approxi-

mate complicated, nonlinear processes extremely effectively. In a way, we're learning that intelligence, as we subjectively perceive it, is a notion that's often conflated with self-awareness, and self-awareness definitely isn't required to solve or carry out these kinds of problems. In the end, the question of computer intelligence may not even be important. As pioneering computer scientist Edsger W. Dijkstra said in "The Threats to Computing Science,"

> *Alan M. Turing thought about . . . the question of whether Machines Can Think, a question . . . about as relevant as the question of whether Submarines Can Swim.*

That general class of algorithms we're talking about falls under the category of *deep learning*, which deals with training mathematical entities named *deep neural networks* on the basis of examples. Deep learning leverages large amounts of data to approximate complex functions whose inputs and outputs are far apart, such as an image (input) and a line of text describing the input (output); a written script (input) and a natural-sounding voice reciting the script (output); or, even more simply, associating an image of a golden retriever with a flag that indicates that a golden retriever is present. This capability allows developers to create programs with functionality that until recently was the exclusive domain of human beings.

1.1 What is PyTorch?

PyTorch is a library for Python programs 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 the library's release, it has grown into one of the most prominent deep learning tools for a broad range of applications.

PyTorch provides a core data structure, the `Tensor`, a multidimensional array that has many similarities with NumPy arrays. From that foundation, a laundry list of features was built to make it easy to get a project up and running, or to design and train investigation into a new neural network architecture. Tensors accelerate mathematical operations (assuming that the appropriate combination of hardware and software is present), and PyTorch has packages for distributed training, worker processes for efficient data loading, and an extensive library of common deep learning functions.

As Python is for programming, PyTorch is both an excellent introduction to deep learning and a tool usable in professional contexts for real-world, high-level work.

We believe that PyTorch should be the first deep learning library you learn. Whether it should be the last is a decision that we'll leave to you.

1.2 What is this book?

This book is intended to be a starting point for software engineers, data scientists, and motivated students who are fluent in Python and want to become comfortable using PyTorch to build deep learning projects. To that end, we take a hands-on approach; we encourage you to keep your computer at the ready so that you can play with the examples and take them a step further.

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

To get the most out of this book, you 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.
- *Willingness to dive in and get your hands dirty*—It'll be much easier for you to learn if you follow along with us.

Deep learning is a huge space. In this book, we'll be covering a tiny part of that space—specifically, using PyTorch for smaller-scope projects. Most of the motivating examples use image processing of 2D and 3D data sets. We focus on practical PyTorch, with the aim of covering enough ground to allow you to solve realistic problems with deep learning or explore new models as they pop up in research literature. A great resource for the latest publications related to deep learning research is the ArXiV public preprint repository, hosted at https://arxiv.org.[1]

1.3 Why PyTorch?

As we've said, deep learning allows you to carry out a wide range of complicated tasks—such as performing machine translation, playing strategy games, and identifying objects in cluttered scenes—by exposing your model to illustrative examples. To do so in practice, you need tools that are flexible so that they can be adapted to your specific problem and efficient, to allow training to occur over large amounts of data in reasonable times. You also need the trained network to perform correctly in the presence of uncertainty in the inputs. In this section, we take a look at some of the reasons why 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 although (like any complicated domain) it has caveats and best practices, using the library generally feels familiar to developers who have used Python previously.

For users who are familiar with NumPy arrays, the PyTorch `Tensor` class will be immediately familiar. PyTorch feels like NumPy, but with GPU acceleration and automatic computation of gradients, which makes it suitable for calculating backward pass data automatically starting from a forward expression.

The `Tensor` API is such that the additional features of the class relevant to deep learning are unobtrusive; the user is mostly free to pretend that those features don't exist until need for them arises.

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

A design driver for PyTorch is expressivity, allowing a developer to implement complicated models without undue complexity being imposed by the library. (The library isn't a 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 in international conferences.[2]

PyTorch also has a compelling story for the transition from research and development to production. Although it initially focused on research workflows, PyTorch has been equipped with a high-performance C++ runtime that users can leverage to deploy models for inference without relying on Python, keeping most of the flexibility of PyTorch without paying the overhead of the Python runtime.

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

1.3.1 The deep learning revolution

In this section, we take a step back and provide some context for where PyTorch fits into the current and historical landscape of deep learning tools.

Until the late 2000s, the broader class of systems that fell into the category "machine learning" relied heavily on *feature engineering*. *Features* are transformations of input data resulting in numerical features that facilitate a downstream algorithm, such as a classifier, to produce correct outcomes on new data. Feature engineering aims to take the original data and come up with *representations* of the same data that can be fed to an algorithm to solve a problem. To tell ones from zeros in images of handwritten digits, for example, you'd 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 in a zero, an eight, or particularly loopy twos.

Deep learning, on the other hand, deals with finding such representations automatically, from raw data, to perform a task successfully. In the ones-versus-zeros example, filters would be refined during training by iteratively looking at pairs of examples and target labels. This isn't to say that feature engineering has no place in deep learning; developers often need to inject some form of knowledge into a learning system. The ability of a neural network to ingest data and extract useful representations on the basis of examples, however, 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 in many disruptive technologies, this fact has led to a change in perspective.

[2] At ICLR 2019, PyTorch appeared as a citation in 252 papers, up from 87 the previous year and at the same level as TensorFlow, which appeared in 266 papers.

On the left side of figure 1.1, a practitioner is busy defining engineering features and feeding them to a learning algorithm. The results of the task will be as good as the features he engineers. On the right side of the figure, with deep learning, the raw data is fed to an algorithm that extracts hierarchical features automatically, based on optimizing the performance of the algorithm on the task. The results will be as good as the practitioner's ability to drive the algorithm toward its goal.

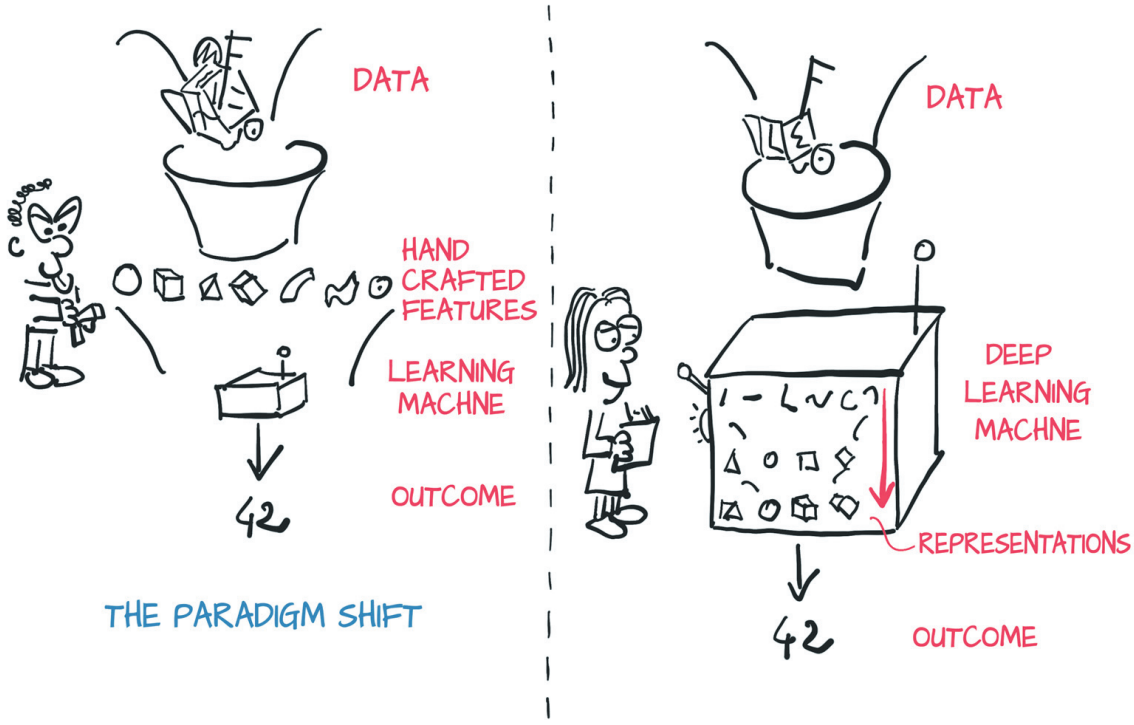


Figure 1.1 **The change in perspective brought by deep learning**

1.3.2 Immediate versus deferred execution

One key differentiator for deep learning libraries is immediate versus deferred execution. Much of PyTorch's ease of use is due to how it implements immediate execution, so we briefly cover that implementation here.

Consider the expression `(a**2 + b**2) ** 0.5` that implements the Pythagorean theorem. If you want to execute this expression, you need to have an `a` and `b` handy, like so:

```
>>> a = 3
>>> b = 4
>>> c = (a**2 + b**2) ** 0.5
>>> c
5.0
```

Immediate execution like this consumes inputs and produces an output *value* (`c` here). PyTorch, like Python in general, defaults to immediate execution (referred to as *eager mode* in the PyTorch documentation). Immediate execution is useful

because if problems arise in executing the expression, the Python interpreter, debugger, and similar tools have direct access to the Python objects involved. Exceptions can be raised directly at the point where the issue occurred.

Alternatively, you could define the Pythagorean expression even before knowing what the inputs are and use that definition to produce the output when the inputs are available. That callable function that you define can be used later, repeatedly, with varied inputs:

```
>>> p = lambda a, b: (a**2 + b**2) ** 0.5
>>> p(1, 2)
2.23606797749979
>>> p(3, 4)
5.0
```

In the second case, you defined a series of operations to perform, resulting in a output function (`p` in this case). You didn't execute anything until later, when you passed in the inputs—an example of deferred execution. Deferred execution means that most exceptions are be raised when the function is called, not when it's defined. For normal Python (as you see here), that's fine, because the interpreter and debuggers have full access to the Python state at the time when the error occurred.

Things get tricky when specialized classes that have heavy operator overloading are used, allowing what looks like immediate execution to be deferred under the hood. These classes can look like the following:

```
>>> a = InputParameterPlaceholder()
>>> b = InputParameterPlaceholder()
>>> c = (a**2 + b**2) ** 0.5
>>> callable(c)
True
>>> c(3, 4)
5.0
```

Often in libraries that use this form of function definition, the operations of squaring `a` and `b`, adding, and taking the square root aren't recorded as high-level Python byte code. Instead, the point usually is to compile the expression into a static computation graph (a graph of basic operations) that has some advantage over pure Python (such as compiling the math directly to machine code for performance reasons).

The fact that the computation graph is built in one place and used in another makes debugging more difficult, because exceptions often lack specificity about what went wrong and Python debugging tools don't have any visibility into the intermediate states of the data. Also, static graphs usually don't mix well with standard Python flow control: they're de-facto domain-specific languages implemented on top of a host language (Python in this case).

Next, we take a more concrete look at the differences between immediate and deferred execution, specifically regarding issues that are relevant to neural networks. We won't be teaching these concepts in any depth here, instead giving you a high-level introduction to the terminology and the relationships among these concepts. Understanding those concepts and relationships lays the groundwork for understand how

libraries like PyTorch that use immediate execution differ from deferred-execution frameworks, even though the underlying math is the same for both types.

The fundamental building block of a neural network is a *neuron*. Neurons are strung together in large numbers to form the network. You see a typical mathematical expression for a single neuron in the first row of figure 1.2: `o = tanh(w * x + b)`. As we explain the execution modes in the following figures, keep these facts in mind:

- `x` is the input to the single-neuron computation.
- `w` and `b` are the parameters or weights of the neuron and can be changed as needed.
- To update the parameters (to produce output that more closely matches what we desire), we assign error to each of the weights via backpropagation and then tweak the weights accordingly.
- Backpropagation requires computing the gradient of the output with respect to the weights (among other things).
- We use automatic differentiation to compute the gradient automatically, saving us the trouble of writing the calculations by hand.

In figure 1.2, the neuron gets compiled into a symbolic graph in which each node represents individual operations (second row), using placeholders for inputs and outputs. Then the graph is evaluated numerically (third row) when concrete numbers are plugged into the placeholders (in this case, the numbers are the values stored in `w`,

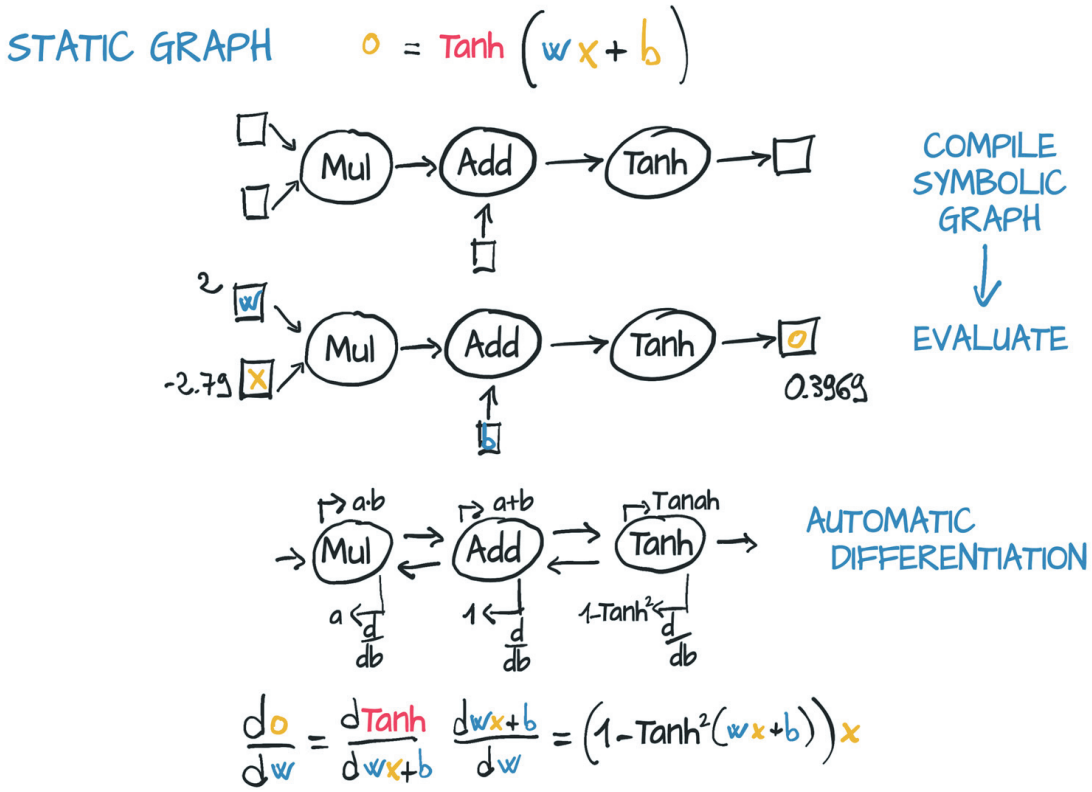


Figure 1.2 **Static graph for a simple computation corresponding to a single neuron**

x, and b). The gradient of the output with respect to the weights is constructed symbolically by automatic differentiation, which traverses the graph backward and multiplies the gradients at individual nodes (fourth row). The corresponding mathematical expression is shown in the fifth row.

One of the major competing deep learning frameworks is TensorFlow, which has a graph mode that uses a similar kind of deferred execution. Graph mode is the default mode of operation in TensorFlow 1.0. By contrast, PyTorch sports a define-by-run dynamic graph engine in which the computation graph is built node by node as the code is eagerly evaluated.

The top half of figure 1.3 shows the same calculation running under a dynamic graph engine. The computation is broken into individual expressions, which are greedily evaluated as they're encountered. The program has no advance notion of the interconnection between computations. The bottom half of the figure shows the behind-the-scenes construction of a dynamic computation graph for the same expression. The expression is still broken into individual operations, but here those operations are eagerly evaluated, and the graph is built incrementally. Automatic differentiation is achieved by traversing the resulting graph backward, similar to static computation graphs. Note that this does not mean dynamic graph libraries are inherently more capa-

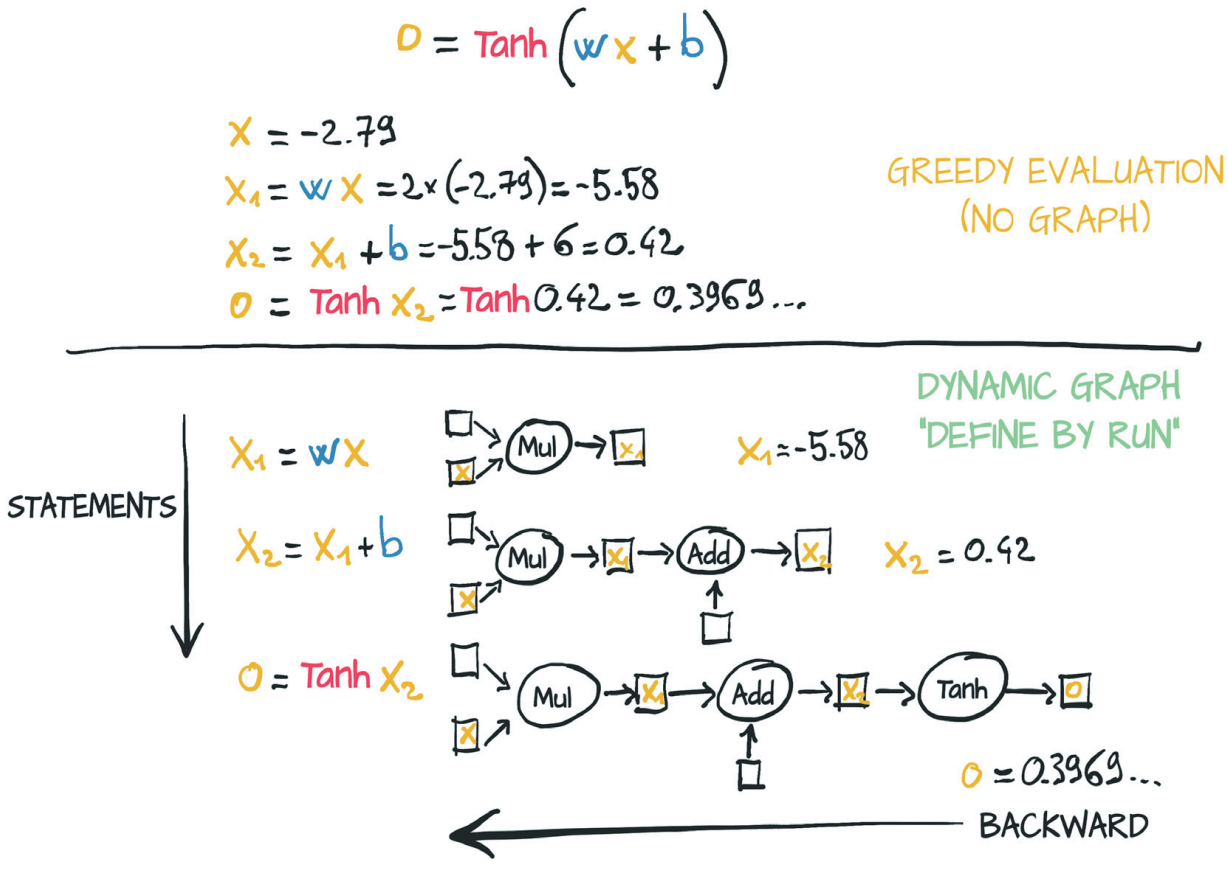


Figure 1.3 Dynamic graph for a simple computation corresponding to a single neuron

ble than static graph libraries, just that it's often easier to accomplish looping or conditional behavior with dynamic graphs.

Dynamic graphs can change during successive forward passes. Different nodes can be invoked according to conditions on the outputs of the preceding nodes, for example, without a need for such conditions to be represented in the graph itself—a distinct advantage over static graph approaches.

The major frameworks are converging toward supporting both modes of operation. PyTorch 1.0 gained the ability to record the execution of a model in a static computation graph or define it through a precompiled scripting language, with the goal of improved performance and ease of putting the model into production. TensorFlow has also gained "eager mode," a new define-by-run API, increasing the library's flexibility as we have discussed.

1.3.3 The deep learning competitive landscape

Although 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 to an era of consolidation and unification.

> **NOTE** The deep learning landscape has been moving so quickly lately that by the time you read this book, some aspects may be out of date. If you're unfamiliar with some of the libraries mentioned here, that's fine.

At the time of PyTorch's first beta release

- Theano and TensorFlow were the premiere low-level deferred-execution libraries.
- 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 roughly two years that followed, the landscape changed dramatically. The community has largely consolidated behind PyTorch or TensorFlow, with the adoption of other libraries dwindling or filling specific niches:

- Theano, one of the first deep learning frameworks, has ceased active development.
- TensorFlow
  - Consumed Keras, promoting it to a first-class API
  - Provided an immediate execution eager mode
  - Announced that TF 2.0 will enable eager mode by default
- PyTorch
  - Consumed Caffe2 for its backend
  - 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
- Added a delayed execution graph mode runtime called TorchScript
- Released version 1.0

TensorFlow has a robust pipeline to production, an extensive industrywide community, and massive mindshare. PyTorch has made huge inroads with the research and teaching community, thanks to its ease of use, and has picked up momentum as researchers and graduates train students and move to industry. Interestingly, with the advent of TorchScript and eager mode, both libraries have seen their feature sets start to converge.

1.4 PyTorch has the batteries included

We've already hinted at a few components of PyTorch. Now we'll take some time to formalize a high-level map of the main components that form PyTorch.

First, PyTorch has the *Py* from *Python*, but there's a lot of non-Python code in it. For performance reasons, most of PyTorch is written in C++ and CUDA[3], a C++-like language from NVIDIA that can be compiled to run with massive parallelism on NVIDIA GPUs. There are ways to run PyTorch directly from C. One of the main motivations for this capability is providing a reliable strategy for deploying models in production. Most of the time, however, you'll interact with PyTorch from Python, building models, training them, and using the trained models to solve problems. Depending on a given use case's requirements for performance and scale, a pure-Python solution can be sufficient to put models into production. It can be perfectly viable to use a Flask web server to wrap a PyTorch model using the Python API, for example.

Indeed, the Python API is where PyTorch shines in term of usability and integration with the wider Python ecosystem. Next, we take a peek at the mental model of PyTorch.

At its core, PyTorch is a library that provides *multidimensional arrays*, called *tensors* in PyTorch parlance, and an extensive library of operations on them is provided by the `torch` module. Both tensors and related operations can run on the CPU or GPU. Running on the GPU results in massive speedups compared with CPU (especially if you're willing to pay for a top-end GPU), and with PyTorch doing so, it doesn't require more than an additional function call or two. The second core thing that PyTorch provides allows tensors to keep track of the operations performed on them and to compute derivatives of an output with respect to any of its inputs analytically via backpropagation. This capability is provided natively by tensors and further refined in `torch.autograd`.

We could argue that by having tensors and the autograd-enabled tensor standard library, PyTorch could be used for more than neural networks, and we'd be correct: PyTorch can be used for physics, rendering, optimization, simulation, modeling, and so on. We're likely to see PyTorch being used in creative ways across the spectrum of scientific applications.

[3] https://www.geforce.com/hardware/technology/cuda

But PyTorch is first and foremost a deep learning library, and as such, it provides all the building blocks needed to build and train neural networks. Figure 1.4 shows a standard setup that loads data, trains a model, and then deploys that model 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. These components can be used to build and initialize the untrained model shown in the center of figure 1.4.

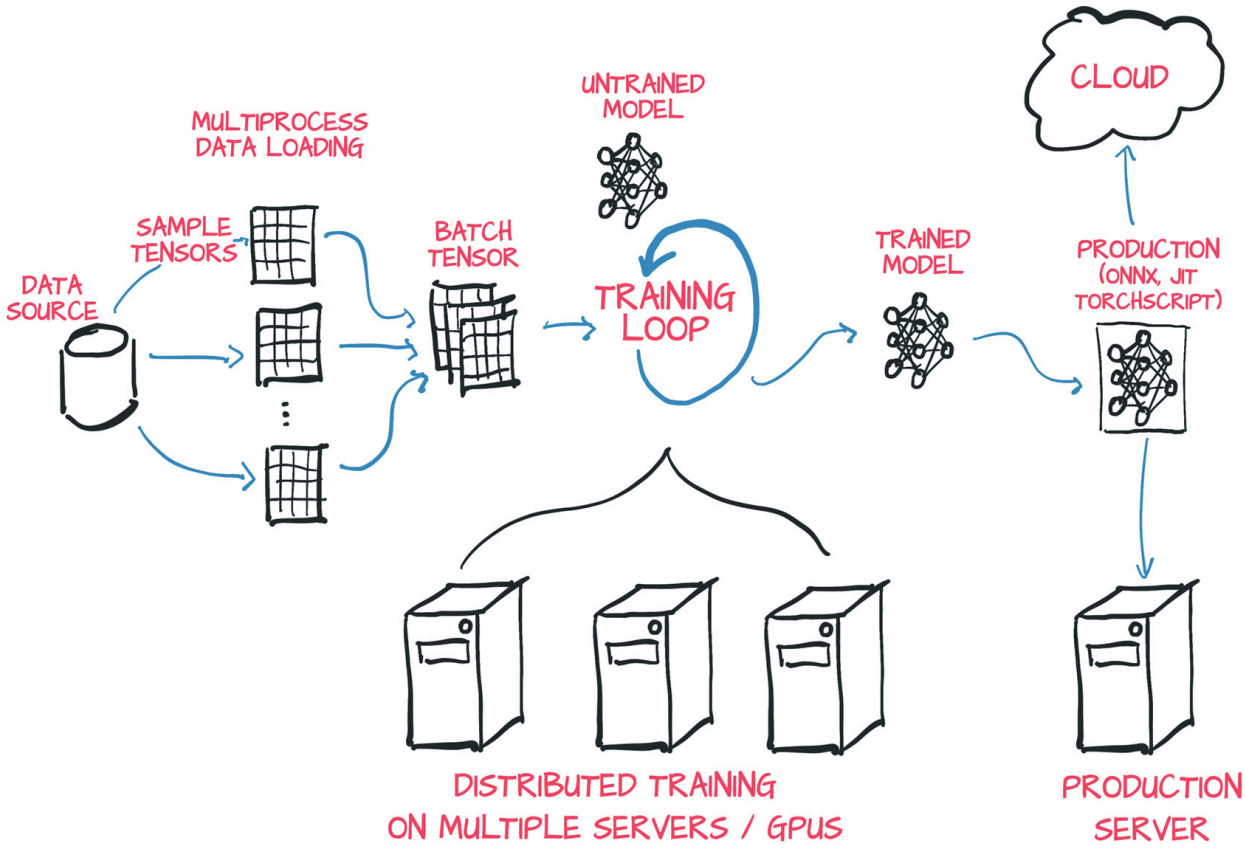


Figure 1.4 **Basic high-level structure of a PyTorch project, with data loading, training, and deployment to production**

To train this model, you need a few things (besides the loop itself, which can be a standard Python `for` loop): 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 be performing the calculations needed for training the model.

Utilities for data loading and handling can be found in `torch.util.data`. The two main classes you'll work with are `Dataset`, which acts as the bridge between your custom data (in whatever format it might be in), and a standardized PyTorch `Tensor`. The other class you'll see a lot of is `DataLoader`, which 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.

In the simplest case, the model will be running the required calculations on the local CPU or on a single GPU, so when the training loop has the data, computation can start immediately. It's more common, however, to want to use specialized hardware such as multiple GPUs or to have multiple machines contribute their resources to training the model. In those cases, `torch.nn.DataParallel` and `torch.distributed` can be employed to leverage the additional hardware available.

When you have results from running your model on the training data, `torch.optim` provides standard ways of updating the model so that the output starts to more closely resemble the answers specified in the training data.

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. This process is as fast as it typically can be on the C++ side, but it incurs the cost of calling that implementation through Python. This cost is minute, but it adds up.

To bypass the cost of the Python interpreter and offer the opportunity to run models independently from a Python runtime, PyTorch also provides a deferred execution model named *TorchScript*. Using TorchScript, PyTorch can serialize a set of instructions that can be invoked independently from Python. You can think of this model as being a virtual machine with a limited instruction set specific to tensor operations. Besides not incurring the costs of calling into Python, this execution mode gives PyTorch the opportunity to Just in Time (JIT) transform sequences of known operations into more efficient fused operations. These features are the basis of the production deployment capabilities of PyTorch.

1.4.1 Hardware for deep learning

Running a pretrained network on new data is within the capabilities of any recent laptop or personal computer. Even retraining a small portion of a pretrained network to specialize it on a new data set doesn't necessarily require specialized hardware. You can follow along with this book on a standard personal computer or laptop. We anticipate, however, that completing a full training run for more-advanced examples will require a CUDA-capable graphical processing unit (GPU), such as a GPU with 8GB of RAM (we suggest an NVIDIA GTX 1070 or better). But those parameters can be adjusted if your hardware has less RAM available.

To be clear: such hardware isn't mandatory if you're willing to wait, but running on a GPU cuts training time by at least an order of magnitude (and usually is 40 to 50 times faster). Taken individually, the operations required to compute parameter updates are fast (from fractions of a second to a few seconds) on modern hardware such as a typical laptop CPU. The issue is that training involves running these operations over and over, many times, incrementally updating the network parameters to minimize training error.

Moderately large networks can take hours to days to train from scratch on large, real-world data sets on workstations equipped with good GPUs. That time can be

reduced by using multiple GPUs on the same machine and even further by using clusters of machines equipped with multiple GPUs. These setups are less prohibitive to access than they sound thanks to the offerings of cloud computing providers. DAWNBench[4] 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 data sets.

If you have a GPU around, great. Otherwise, we suggest checking out the offerings of the various cloud platforms, many of which offer GPU-enabled Jupyter notebooks with PyTorch preinstalled, often with a free quota.

Last consideration: the operating system (OS). PyTorch has supported Linux and macOS from its first release and gained Windows support during 2018. Because current Apple laptops don't include GPUs that support CUDA, the precompiled macOS packages for PyTorch are CPU-only. We try to avoid assuming that you run a particular OS; scripts' command lines should convert to a Windows-compatible form readily. For convenience, whenever possible we list code as though it's running on a Jupyter Notebook.

For installation information, please see the Getting Started guide on the official website.[5] We suggest that Windows users install with Anaconda or Miniconda. Other operating systems, such as Linux, typically have a wider variety of workable options, with Pip being one of the most common installers. Experienced users, of course, are free to install packages in the way that's most compatible with their preferred development environments.

1.4.2 Using Jupyter Notebooks

We're going to assume that you have PyTorch and the other dependencies installed and have verified that things are working. We're going to be making heavy use of Jupyter Notebooks for example code. A Jupyter Notebook shows itself as a page in the browser through which you can run code interactively. The code gets evaluated by a *kernel*, a process running on a server that's ready to receive code to execute and send back the results, which are rendered inline on the page. A notebook maintains the state of the kernel, such as variables defined during the evaluation of code, in memory until it's terminated or restarted. The fundamental unit with which you interact with a notebook is a *cell*, a box on the page where you can type code and have the kernel evaluate it (by choosing the menu item or pressing Shift-Enter). You can add multiple cells to a notebook, and the new cells see the variables you created in the earlier cells. The value returned by the last line of a cell is printed below the cell after execution, and the same goes for plots. By mixing source code, results of evaluations, and Markdown-formatted text cells, you can generate beautiful interactive documents. You can read everything about Jupyter Notebooks on the project website.[6]

---

[4] https://dawn.cs.stanford.edu/benchmark/index.html
[5] https://pytorch.org/get-started/locally
[6] https://jupyter.org

At this point, you'll need to start the notebook server from the root directory of the code checkout from GitHub. How starting the server looks depends on the details of your operating system and on how and where you installed Jupyter. If you have questions, feel free to ask on our forums.[7] When the notebook server starts, your default browser pops up, showing a list of local notebook files.

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

You can find full working code for the listings in this book in our repository on GitHub.[8]

Exercises

- Start Python to get an interactive prompt.
  - What Python version are you using: 2.x or 3.x?
  - Can you `import torch`? What version of PyTorch do you get?
  - What is the result of `torch.cuda.is_available()`? Does it match your expectation based on the hardware you're using?
- Start the Jupyter Notebook server.
  - What version of Python is Jupyter using?
  - Is the location of the `torch` library used by Jupyter the same as the one you imported from the interactive prompt?

Summary

- Deep learning models automatically learn to associate inputs and desired outputs from examples.
- Libraries like PyTorch allow you to build and train neural network models efficiently.
- PyTorch minimizes cognitive overhead while focusing on flexibility and speed. It also defaults to immediate execution for operations.
- TorchScript is a precompiled deferred-execution mode that can be invoked from C++.
- Since the release of PyTorch in early 2017, the deep learning tooling ecosystem has consolidated significantly.
- PyTorch provides several utility libraries to facilitate deep learning projects.

[7] https://forums.manning.com/forums/deep-learning-with-pytorch
[8] https://github.com/deep-learning-with-pytorch/dlwpt-code

2 It starts with a tensor

This chapter covers

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

Deep learning enables many applications, which invariably consist of taking data in some form, such as images or text, and producing data in another form, such as labels, numbers, or more text. Taken from this angle, deep learning 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. The system might note the general shape of a dog and the typical colors of a golden retriever, for example. 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 first step of this process is converting the input into floating-point numbers, as you see in the first step of figure 2.1 (along with many other types of data).

Because a network uses floating-point numbers to deal with information, you need a way to encode real-world data of the kind you want to process into something that's digestible by a network and then decode the output back to something you can understand and use for a purpose.

The transformation from one form of data to another is typically learned by a deep neural network in stages, which means that you can think of the partially transformed data between stages as being a sequence of intermediate representations. For image recognition, early representations can be things (like edge detection) or 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 structure in the data, in a way that's 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. It's important to keep in mind that these intermediate representations (such as the ones shown in the second step of figure 2.1) 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 you can begin the process of converting data to floating-point input, you must have a solid understanding of how PyTorch handles and stores data: as input, as

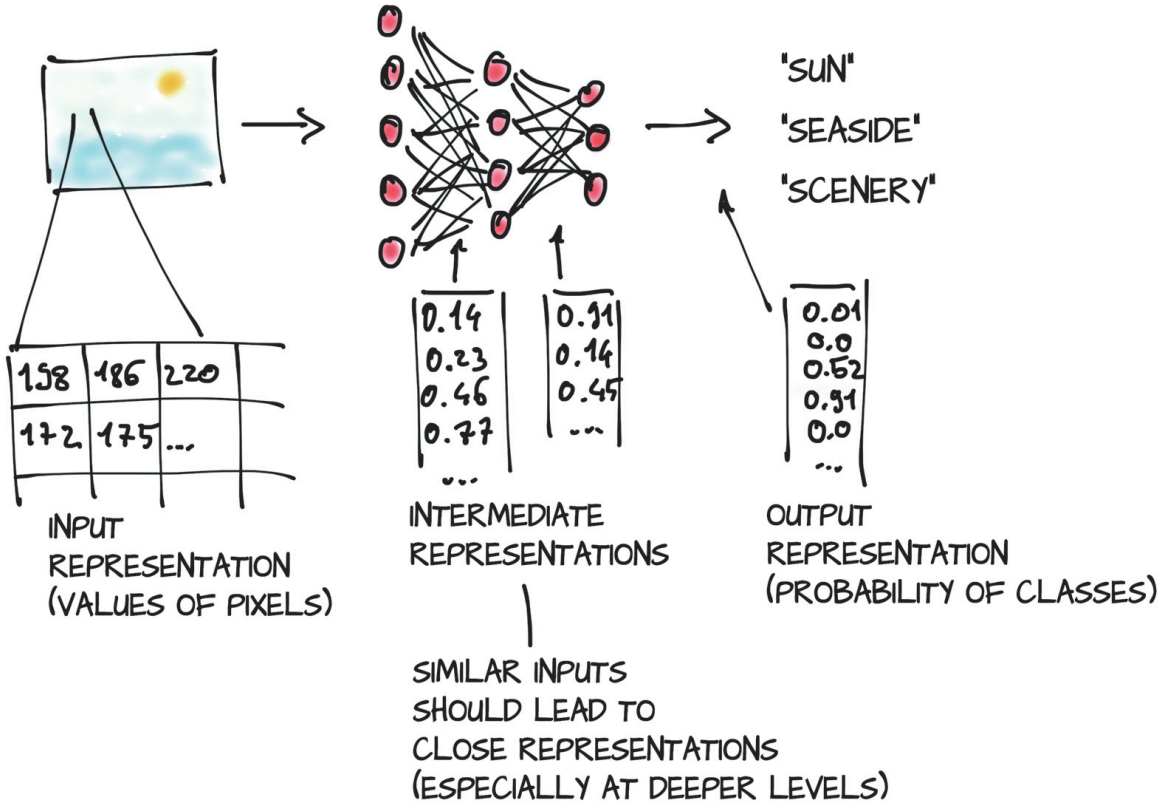


Figure 2.1 **A deep neural network learns how to transform an input representation to an output representation. (Note: The number of neurons and outputs is not to scale.)**

intermediate representations, and as output. This chapter is devoted to providing precisely that understanding.

To this end, PyTorch introduces a fundamental data structure: the tensor. 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 everyone else, *tensor* refers to the generalization of vectors and matrices to an arbitrary number of dimensions, as shown in figure 2.2. Another name for the same concept is *multidimensional arrays*. The dimensionality of a tensor coincides with the number of indexes used to refer to scalar values within the tensor.

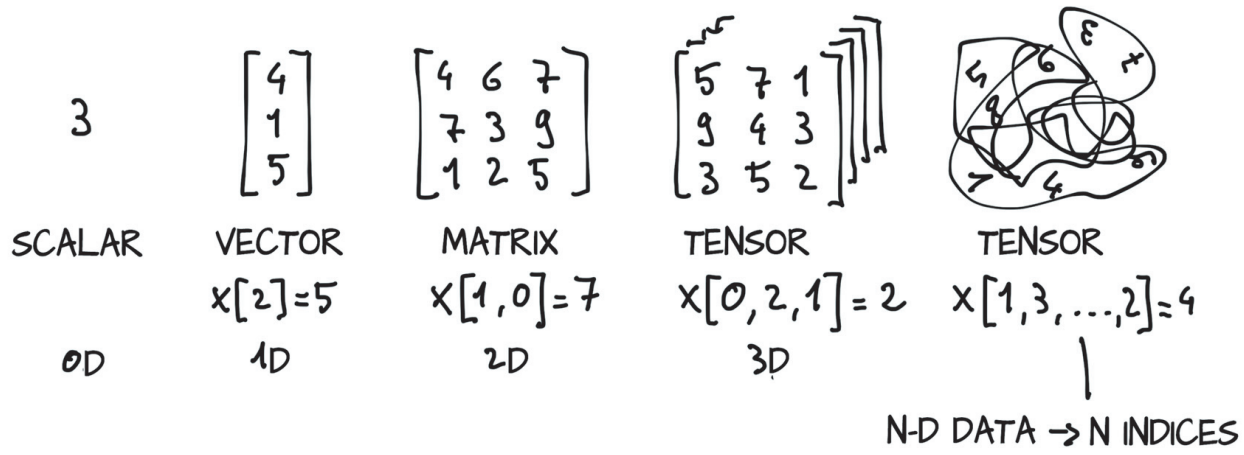


Figure 2.2 **Tensors are the building blocks for representing data in PyTorch**

PyTorch isn't 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 arguably become the *lingua franca* of data science. In fact, 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[1], Scikit-learn[2], and Pandas[3].

Compared with NumPy arrays, PyTorch tensors have a few superpowers, such as the ability to perform fast operations on graphical processing units (GPUs), to distribute operations on multiple devices or machines, and to keep track of the graph of computations that created them. All these features are important in implementing a modern deep learning library.

We start the chapter by introducing PyTorch tensors, covering the basics to set things in motion. We show you how to manipulate tensors by using the PyTorch tensor library, covering things such as how the data is stored in memory and how certain operations can be performed on arbitrarily large tensors in constant time; then we move on to the aforementioned NumPy interoperability and the GPU acceleration.

Understanding the capabilities and API of tensors is important if they're to be go-to tools in your programming toolbox.

---

[1] https://www.scipy.org
[2] https://scikit-learn.org/stable
[3] https://pandas.pydata.org

2.1 Tensor fundamentals

You've already learned that tensors are the fundamental data structures in PyTorch. A tensor is an array—that is, a data structure storing collection of numbers that are accessible individually by means of an index and that can be indexed with multiple indices.

Take a look at `list` indexing in action so that you can compare it with tensor indexing. The following listing shows a list of three numbers in Python.

Listing 2.1 code/p1ch3/1_tensors.ipynb

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

You can access the first element of the list by using the corresponding `0`-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's not unusual for simple Python programs that deal with vectors of numbers, such as the coordinates of a 2D line, to use Python lists to store the vector. This practice can be suboptimal, however, for several reasons:

- *Numbers in Python are full-fledged objects.* Whereas a floating-point number might take only 32 bits to be represented on a computer, Python *boxes* them in a full-fledged Python object with reference counting and so on. This situation isn't a problem if you need to store a small number of numbers, but allocating millions of such numbers gets inefficient.
- *Lists in Python are meant for sequential collections of objects.* No operations are defined for, say, efficiently taking the dot product of two vectors or summing vectors. Also, Python lists have no way of optimizing the layout of their content in memory, as they're indexable collections of pointers to Python objects (of any kind, not numbers alone). Finally, Python lists are one-dimensional, and although you can create lists of lists, again, this practice is inefficient.
- *The Python interpreter is slow compared with optimized, compiled code.* Performing mathematical operations on large collections of numerical data can be must 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 such as PyTorch tensors that provide efficient low-level implementations of

numerical data structures and related operations on them, wrapped in a convenient high-level API.

Many types of data—from images to time series, audio, and even sentences—can be represented by tensors. By defining operations over tensors, some of which you explore in this chapter, you 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.

Now you're ready to construct your first PyTorch tensor to see what it looks like. This tensor won't be particularly meaningful for now, being three ones in a column:

```
# In[4]:
import torch
a = torch.ones(3)
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.])
```

Now take a look at what you did here. After importing the `torch` module, you called a function that creates a (one-dimensional) tensor of size `3` filled with the value `1.0`. You can access an element by using its `0`-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. Python lists or tuples of numbers are collections of Python objects that are individually allocated in memory, as shown on the left side of figure 2.3. PyTorch tensors or NumPy arrays, on the other hand, are views over (typically) contiguous memory blocks containing unboxed C numeric types, not Python objects. In this case, 32 bits (4 bytes) float, as you see on the right side of figure 2.3. So a 1D tensor of 1 million float numbers requires 4 million contiguous bytes to be stored, plus a small overhead for the metadata (dimensions, numeric type, and so on).

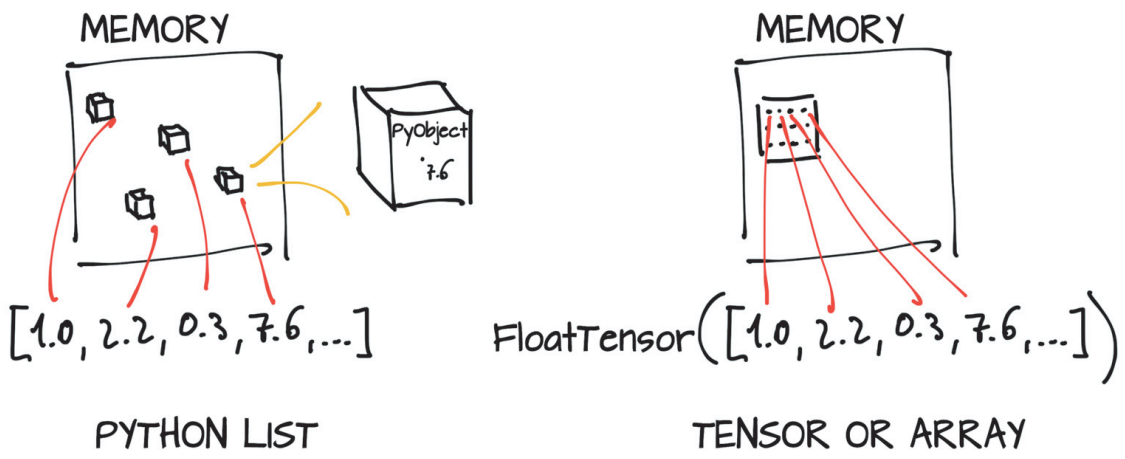


Figure 2.3 Python object (boxed) numeric values versus tensor (unboxed array) numeric values

Suppose that you have a list of 2D coordinates that you'd like to manage to represent a geometrical object, such as a triangle. The example isn't particularly pertinent to deep learning, but it's easy to follow. Instead of having coordinates as numbers in a Python list, you can use a one-dimensional tensor by storing xs in the even indices and ys in the odd indices, like so:

```
# In[8]:
points = torch.zeros(6)
points[0] = 1.0
points[1] = 4.0
points[2] = 2.0
points[3] = 1.0
points[4] = 3.0
points[5] = 5.0
```

The use of .zeros here is a way to get an appropriately sized array.

Overwrite those zeros with the values you want.

You can also pass a Python list to the constructor to the same effect

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

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

to get the coordinates of the first point:

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

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

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

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

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

Here, you passed a list of lists to the constructor. You can ask the tensor about its shape,

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

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

which informs you of the size of the tensor along each dimension. You 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 you can access an individual element in the tensor by using two indices:

```
# In[14]:
points = torch.FloatTensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])
points

# Out[14]:
tensor([[1., 4.],
        [2., 1.],
        [3., 5.]])

# In[15]:
points[0, 1]

# Out[15]:
tensor(4.)
```

This code returns the `y` coordinate of the `0`th point in your data set. You can also access the first element in the tensor as you did before to get the 2D coordinates of the first point:

```
# In[16]:
points[0]

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

Note that what you get as the output is *another tensor*, but a 1D tensor of size 2 containing the values in the first row of the `points` tensor. Does this output mean that 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 process would be inefficient, especially if you had millions of points. What you got back instead was a different *view* of the same underlying data, limited to the first row.

2.2 Tensors and storages

In this section, you start getting hints about the implementation under the hood. Values are allocated in contiguous chunks of memory, managed by `torch.Storage` instances. A *storage* is a one-dimensional array of numerical data, such as a contiguous block of memory containing numbers of a given type, perhaps a `float` or `int32`. A PyTorch `Tensor` is a view over such a `Storage` that's capable of indexing into that storage by using an offset and per-dimension strides.

Multiple tensors can index the same storage even if they index into the data differently. You can see an example in figure 2.4. In fact, when you requested `points[0]` in the last snippet, what you got back was another tensor that indexes the same storage as the `points` tensor, but not all of it and with different dimensionality (1D versus 2D). The underlying memory is allocated only once, however, so creating alternative tensor views on the data can be done quickly, regardless of the size of the data managed by the `Storage` instance.

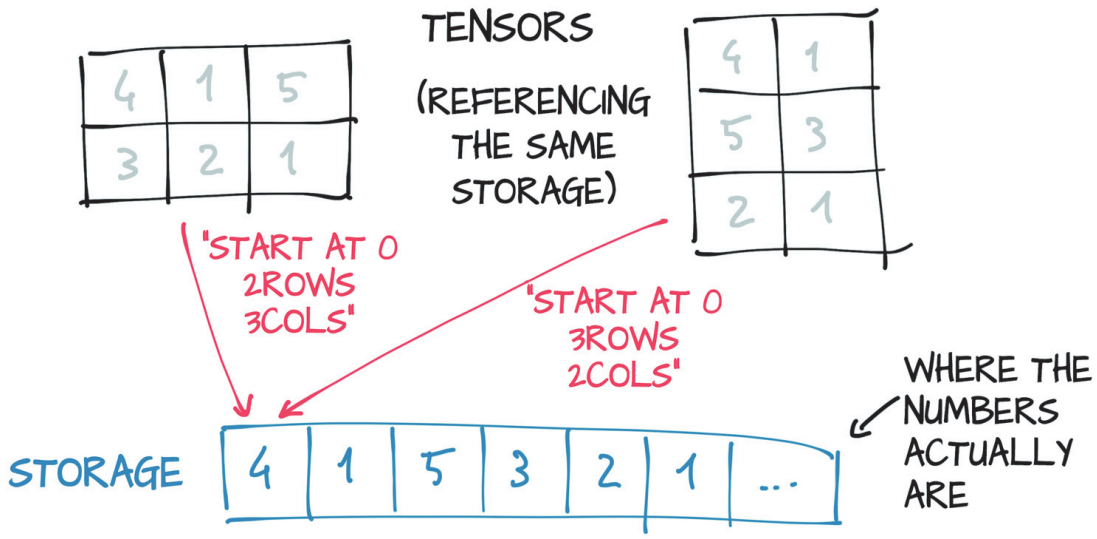


Figure 2.4 Tensors are views over a `Storage` instance

Next, you see how indexing into the storage works in practice with 2D points. You can access the storage for a given tensor by using the `.storage` property:

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

# Out[17]:
 1.0
 4.0
 2.0
 1.0
 3.0
 5.0
[torch.FloatStorage 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 knows how to translate a pair of indices into a location in the storage.

You can also index into a storage manually:

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

# Out[18]:
1.0

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

# Out[19]:
4.0
```

You can't index a storage of a 2D tensor by using two indices. The layout of a storage is always one-dimensional, irrespective of the dimensionality of any tensors that may refer to it.

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

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

# Out[20]:
tensor([[2., 4.],
        [2., 1.],
        [3., 5.]])
```

You'll seldom, if ever, use storage instances directly, but understanding the relationship between a tensor and the underlying storage is useful for understanding the cost (or lack thereof) of certain operations later. This mental model is a good one to keep in mind when you want to write effective PyTorch code.

2.3 Size, storage offset, and strides

To index into a storage, tensors rely on a few pieces of information that, together with their storage, unequivocally define them: size, storage offset, and stride (figure 2.5). 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 that corresponds to the first element in the tensor. The *stride* is the number of elements in the storage that need to be skipped to obtain the next element along each dimension.

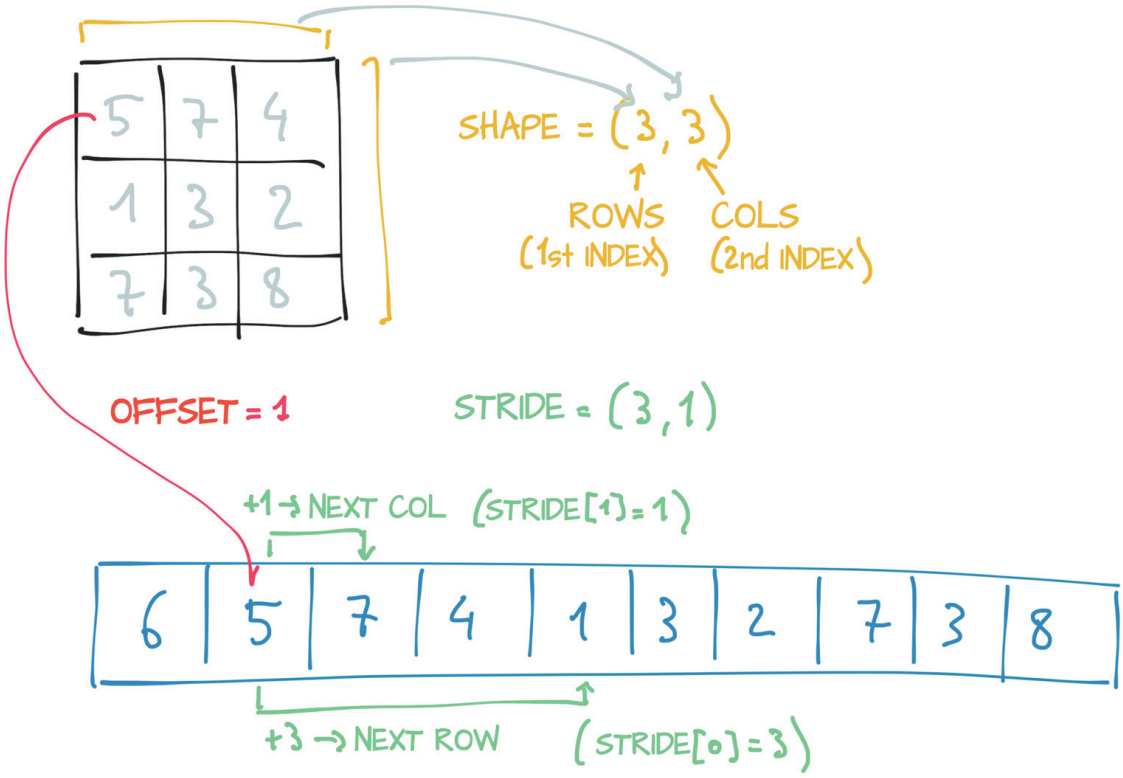


Figure 2.5 **Relationship among a tensor's offset, size, and stride**

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

```
# In[21]:
points = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])
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 (because we need to skip the first point, which has two items) and the size is an instance of the `Size` class containing one element because the tensor is one-dimensional. Important note: this information is the same information contained in the `shape` property of tensor objects:

```
# In[23]:
second_point.shape

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

Last, 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. Your `points` tensor, for example, has a stride:

```
# 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 into a storage created to hold a larger tensor the offset might be a positive value.

This indirection between `Tensor` and `Storage` leads some operations, such as transposing a tensor or extracting a subtensor, to be inexpensive, as they don't lead to memory reallocations; instead, they consist of allocating a new tensor object with a different value for size, storage offset, or stride.

You saw how to extract a subtensor when you indexed a specific point and saw the storage offset increasing. Now see what happens to size and stride:

```
# In[25]:
points = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])
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,)
```

Bottom line, the subtensor has one fewer dimension (as you'd expect) while still indexing the same storage as the original `points` tensor. Changing the subtensor has a side effect on the original tensor too:

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

# Out[28]:
tensor([[ 1.,  4.],
        [10.,  1.],
        [ 3.,  5.]])
```

This effect may not always be desirable, so you can eventually clone the subtensor into a new tensor:

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

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

Try transposing now. Take your `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 along the columns:

```
# In[30]:
points = torch.tensor([[1.0, 4.0], [2.0, 1.0], [3.0, 5.0]])
points

# Out[30]:
tensor([[1., 4.],
        [2., 1.],
        [3., 5.]])

# In[31]:
points_t = points.t()
points_t

# Out[31]:
tensor([[1., 2., 3.],
        [4., 1., 5.]])
```

You can easily verify that the two tensors share 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 result tells you that increasing the first index by 1 in `points`—that is, going from `points[0,0]` to `points[1,0]`—skips along the storage by two elements, and that increasing the second index from `points[0,0]` to `points[0,1]` skips along the storage by one. In other words, the storage holds the elements in the tensor sequentially row by row.

You can transpose `points` into `points_t` as shown in figure 2.6. You change the order of the elements in the stride. After that, increasing the row (the first index of the tensor) skips along the storage by 1, as when you were moving along columns in `points`. This is the definition of transposing. No new memory is allocated: transposing is obtained only by creating a new `Tensor` instance with different stride ordering from the original.

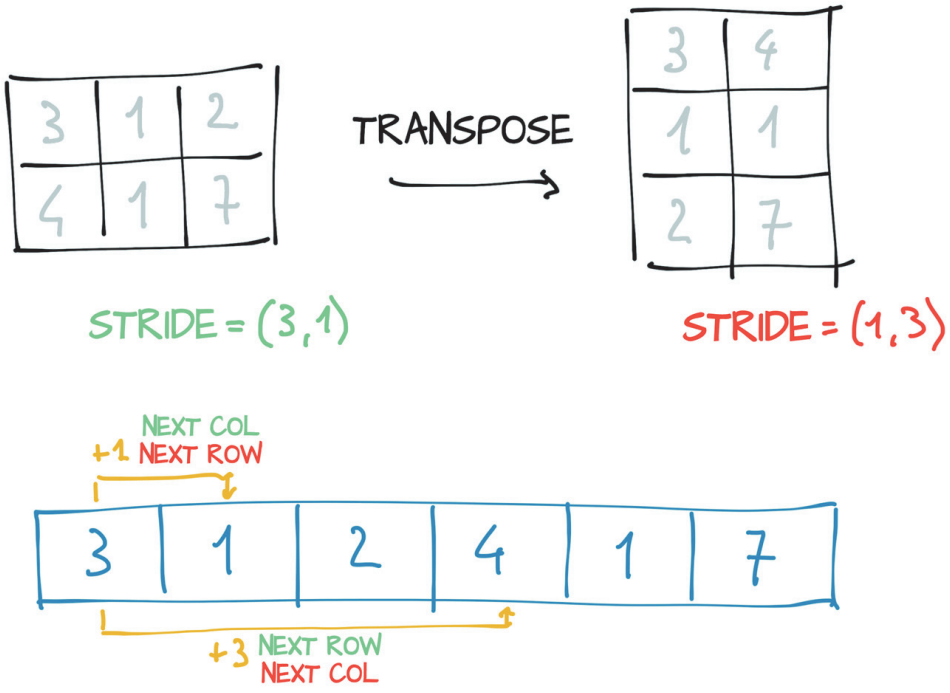


Figure 2.6 **Transpose operation applied to a tensor**

Transposing in PyTorch isn't limited to matrices. You can transpose a multidimensional array by specifying the two dimensions along which transposing (such as flipping shape and stride) should occur:

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

# Out[35]:
torch.Size([3, 4, 5])

# In[36]:
some_tensor_t.shape

# Out[36]:
torch.Size([5, 4, 3])

# In[37]:
some_tensor.stride()

# Out[37]:
(20, 5, 1)

# In[38]:
some_tensor_t.stride()

# Out[38]:
(1, 5, 20)
```

A tensor whose values are laid out in the storage starting from the rightmost dimension onward (moving along rows for a 2D tensor, for example) is defined as being *contiguous*. Contiguous tensors are convenient because you can visit them efficiently and in order without jumping around in the storage. (Improving data locality improves performance because of the way memory access works in modern CPUs.)

In this case, `points` is contiguous but its transpose is not:

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

# Out[39]:
True

# In[40]:
points_t.is_contiguous()

# Out[40]:
False
```

You can obtain a new contiguous tensor from a noncontiguous one by using the `contiguous` method. The content of the tensor stays the same, but the stride changes, as does the storage:

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

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

# In[42]:
points_t.storage()

# Out[42]:
 1.0
 4.0
 2.0
 1.0
 3.0
 5.0
[torch.FloatStorage 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([[1., 2., 3.],
        [4., 1., 5.]])

# In[45]:
points_t_cont.stride()

# Out[45]:
(3, 1)

# In[46]:
points_t_cont.storage()

# Out[46]:
 1.0
 2.0
 3.0
 4.0
 1.0
 5.0
[torch.FloatStorage of size 6]
```

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

2.4 Numeric types

All right, you know the basics of how tensors work. But we haven't touched on the numeric types you can store in a `Tensor`. The `dtype` argument to tensor constructors (that is, functions such as `tensor`, `zeros`, and `ones`) specifies the numerical data type that will be contained in the tensor. The data type specifies the possible values that the tensor can hold (integers versus floating-point numbers) and the number of bytes per value.[4] The `dtype` argument is deliberately similar to the standard NumPy argument of the same name. Here's a list of the possible values for the `dtype` argument:

- `torch.float32` or `torch.float`—32-bit floating-point
- `torch.float64` or `torch.double`—64-bit, double-precision floating-point
- `torch.float16` or `torch.half`—16-bit, half-precision floating-point
- `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

Each of `torch.float`, `torch.double`, and so on has a corresponding concrete class of `torch.FloatTensor`, `torch.DoubleTensor`, and so on. The class for `torch.int8` is `torch.CharTensor`, and the class for `torch.uint8` is `torch.ByteTensor`. `torch.Tensor` is an alias for `torch.FloatTensor`. The default data type is 32-bit floating-point.

To allocate a tensor of the right numeric type, you can specify the proper `dtype` as an argument to the constructor, as follows:

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

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

```
# In[48]:
short_points.dtype

# Out[48]:
torch.int16
```

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

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

[4] And signedness, in the case of uint8

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, `type` and `to` perform the same type check-and-convert-if-needed operation, but the `to` method can take additional arguments.

You can always cast a tensor of one type as a tensor of another type by using the `type` method:

```
# In[51]:
points = torch.randn(10, 2)
short_points = points.type(torch.short)
```

randn initializes the tensor elements to random numbers between 0 and 1.

2.5 Indexing tensors

You've seen that `points[0]` returns a tensor containing the 2D point at the first row of the tensor. What if you need to obtain a tensor that contains all points but the first? That task is easy when you use range indexing notation, the same kind that applies to standard Python lists:

All elements in the list

From element 1 inclusive to element 4 exclusive in steps of 2

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

From element 1 inclusive to element 4 exclusive

From element 1 inclusive to the end of the list

From the start of the list to element 4 exclusive

From the start of the list to one before the last element

To achieve your goal, you can use the same notation for PyTorch tensors, with the added benefit that as in NumPy and in other Python scientific libraries, we can use range indexing for each dimension of the tensor:

All rows after first, all columns

```
# In[54]:
points[1:]
points[1:, :]
points[1:, 0]
```

All rows after first, implicitly all columns

All rows after first, first column

In addition to using ranges, PyTorch features a powerful form of indexing called *advanced indexing*.

2.6 NumPy interoperability

Although we don't consider experience in NumPy to be a prerequisite for reading this book, we strongly encourage you to get familiar with NumPy due to its ubiquity in the Python data science ecosystem. PyTorch tensors can be converted to NumPy arrays and vice versa efficiently. By doing so, you can leverage 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 that works with the Python buffer protocol.[5]

To get a NumPy array out of your `points` tensor, 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 returns a NumPy multidimensional array of the right size, shape, and numerical type. Interestingly, the returned array shares an underlying buffer with the tensor storage. As a result, the `numpy` method can be executed effectively at essentially no cost as long as the data sits in CPU RAM, and modifying the NumPy array leads to a change in the originating tensor.

If the tensor is allocated on the GPU, PyTorch makes a copy of the content of the tensor into a NumPy array allocated on the CPU.

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

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

which uses the same buffer-sharing strategy.

2.7 Serializing tensors

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

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

As an alternative, you 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 your `points` back is similarly a one-liner:

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

[5] https://docs.python.org/3/c-api/buffer.html

The equivalent is

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

This technique allows you to save tensors quickly in case you only want to load them with PyTorch, but the file format itself isn't interoperable. You can't read the tensor with software other than PyTorch. Depending on the use case, this situation may not be a limitation, but you should learn how to save tensors interoperably for those times when it is. Although every use case is unique, we suspect that this one will be more common when you introduce PyTorch into existing systems that already rely on different libraries. New projects probably won't need to save tensors interoperably as often.

For those cases when you need to, however, you can use the HDF5 format and library.[6] 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[7], which accepts and returns data under the form of NumPy arrays.

You can install `h5py` by using

```
$ conda install h5py
```

At this point, you can save your `points` tensor by converting it to a NumPy array (at no cost, as 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. You can have other keys, even nested ones. One interesting thing in HDF5 is that you can index the data set while on disk and access only the elements you're interested in. Suppose that you want to load only the last two points in your data set:

```
# In[62]:
f = h5py.File('../data/p1ch3/ourpoints.hdf5', 'r')
dset = f['coords']
last_points = dset[1:]
```

Here, data wasn't loaded when the file was opened or the data set was required. Rather, data stayed on disk until you requested the second and last rows in the data set. At that point, `h5py` accessed those two columns and returned a NumPy array-like object encapsulating that region in that data set that behaves like a NumPy array and has the same API.

---

[6] https://www.hdfgroup.org/solutions/hdf5
[7] http://www.h5py.org

Owing to this fact, you 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[1:])
f.close()
>>> last_points = torch.from_numpy(dset[1:])
```

When you finish loading data, close the file.

2.8 Moving tensors to the GPU

One last point about PyTorch tensors is related to computing on the GPU. Every Torch tensor can be transferred to a GPUs to perform fast, massively parallel computations. All operations to be performed on the tensor are carried out by GPU-specific routines that come with PyTorch.

> **NOTE** As of early 2019, main PyTorch releases have acceleration only on GPUs that have support for CUDA. Proof-of-concept versions of PyTorch running on AMD's ROCm[8] platform exist, but full support hasn't been merged into PyTorch as of version 1.0. Support for Google's TPUs is a work in progress[9], with the current proof of concept available to the public in Google Colab.[10] Implementation of data structures and kernels on other GPU technology, such as OpenCL, wasn't planned at the time we wrote this chapter.

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

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

You could instead copy a tensor created on the CPU to the GPU by using the `to` method:

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

This code returns a new tensor that has the same numerical data but is stored in the RAM of the GPU rather than in regular system RAM.

Now that the data is stored locally on the GPU, you start to see speedups when performing mathematical operations on the tensor. Also, the class of this new GPU-backed tensor changes to `torch.cuda.FloatTensor`. (Given the starting type of `torch.FloatTensor`; the corresponding set of `torch.cuda.DoubleTensor` and so on exists.) In almost all cases, CPU- and GPU-based tensors expose the same user-facing API, making

[8] https://rocm.github.io
[9] https://github.com/pytorch/xla
[10] https://colab.research.google.com

it much easier to write code that is agnostic to where the heavy number-crunching process is running.

In case your machine has more than one GPU, you can decide which GPU to allocate the tensor to by passing a zero-based integer identifying the GPU on the machine:

```
# 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                                  # Multiplication performed on the CPU
points_gpu = 2 * points.to(device='cuda')            # Multiplication performed on the GPU
```

Note that the `points_gpu` tensor isn't brought back to the CPU when the result has been computed. Here's what happened:

1. The `points` tensor was copied to the GPU.
2. A new tensor was allocated on the GPU and used to store the result of the multiplication.
3. A handle to that GPU tensor was returned.

Therefore, if you 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 (except if you print or access the resulting tensor). To move the tensor back to the CPU, you need to provide a `cpu` argument to the `to` method:

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

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

```
# In[70]:
points_gpu = points.cuda()                           # Defaults to GPU index 0
points_gpu = points.cuda(0)
points_cpu = points_gpu.cpu()
```

It's worth mentioning that when you use the `to` method, you can change the placement and the data type simultaneously by providing `device` and `dtype` as arguments.

2.9 The tensor API

At this point, you know what PyTorch tensors are and how they work under the hood. Before we wrap up this chapter, we'll take a look at the tensor operations that PyTorch offers. It would be of little use to list all of them all here. Instead, we're going to give you a general feel for the API and show you 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 under the `torch` module and can also be called as methods of a tensor object. The `transpose` function that you encountered earlier, for example, can be used from the `torch` module

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

or as a method of the `a` tensor:

```
# In[72]:
a = torch.ones(3, 2)
a_t = a.transpose(0, 1)
```

No difference exists between the two forms, which can be used interchangeably. A caveat, though: a small number of operations exist only as methods of the tensor object. They're recognizable by the trailing underscore in their name, such as `zero_`, which indicates that the method operates *in-place* by modifying the input instead of creating a new output tensor and returning it. The `zero_` method, for example, zeros out all the elements of the input. Any method *without* the trailing underscore leaves the source tensor unchanged and 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.]])
```

Earlier, we mentioned the online docs[11], which are exhaustive and well organized with the tensor operations divided into groups:

- *Creation ops*—Functions for constructing a tensor, such as `ones` and `from_numpy`
- *Indexing, slicing, joining, and mutating ops*—Functions for changing the shape, stride, or content of a tensor, such as `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, such as `abs` and `cos`
  - *Reduction ops*—Functions for computing aggregate values by iterating through tensors, such as `mean`, `std`, and `norm`

[11] http://pytorch.org/docs

  - *Comparison ops*—Functions for evaluating numerical predicates over tensors, such as `equal` and `max`
  - *Spectral ops*—Functions for transforming in and operating in the frequency domain, such as `stft` and `hamming_window`
  - *Other ops*—Special functions operating on vectors, such as `cross`, or matrices, such as `trace`
  - *BLAS and LAPACK ops*—Functions that follow the BLAS (Basic Linear Algebra Subprograms) specification for scalar, vector-vector, matrix-vector, and matrix-matrix operations
- *Random sampling ops*—Functions for generating values by drawing randomly from probability distributions, such as `randn` and `normal`
- *Serialization ops*—Functions for saving and loading tensors, such as `load` and `save`
- *Parallelism ops*—Functions for controlling the number of threads for parallel CPU execution, such as `set_num_threads`

It's useful to play with the general tensor API. This chapter should provide all the prerequisites for this kind of interactive exploration.

Exercises

- Create a tensor `a` from `list(range(9))`. Predict then check what the size, offset, and strides are.
- Create a tensor `b = a.view(3, 3)`. What is the value of `b[1,1]`?
- Create a tensor `c = b[1:,1:]`. Predict then check what the size, offset, and strides are.
- Pick a mathematical operation like cosine or square root. Can you find a corresponding function in the torch library?
- Is there a version of your function that operates in-place?

Summary

- Neural networks transform floating-point representations into other floating-point representations, with the starting and ending representations typically being human-interpretable. The intermediate representations are less so.
- These floating-point representations are stored in tensors.
- Tensors are multidimensional arrays and the basic data structure in PyTorch.
- PyTorch has a comprehensive standard library for tensor creation and manipulation and for mathematical operations.
- Tensors can be serialized to disk and loaded back.
- 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 (such as `Tensor.sqrt_`).

3 Real-world data representation with tensors

This chapter covers

- Representing different types of real-world data as PyTorch tensors
- Working with range of data types, including spreadsheet, time series, text, image, and medical imaging
- Loading data from file
- Converting data to tensors
- Shaping tensors so that they can be used as inputs for neural network models

Tensors are the building blocks for data in PyTorch. Neural networks take tensors in input and produce tensors as outputs. In fact, all operations within a neural network and during optimization are operations between tensors, and all parameters (such as 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.

We can address one question at this point: how do you take a piece of data, a video, or text, and represent it with a tensor, and do that in a way that's appropriate for training a deep learning model?

The answer is what you'll learn in this chapter. We cover different types of data and show you how to get them represented as tensors. Then we show you how to load the data from the most common on-disk formats and also get a feeling for those data types structure so that you can see how to prepare them for training a neural network. Often, your raw data won't be perfectly formed for the problem you'd like to solve, so you'll have a chance to practice your tensor manipulation skills on a few more interesting tensor operations. You'll be using a lot of image and volumetric data because those data types are common and reproduce well in book format. We also cover tabular data, time series, and text, which are also of interest to many readers.

Each section of the chapter describes a data type, and each comes with its own data set. Although we've structured the chapter so that each data type builds on the preceding one, you should feel free to skip around a bit if you're so inclined.

We start with tabular data of data about wines, as you'd find in a spreadsheet. Next, we move to *ordered* tabular data, with a time-series data set from a bike-sharing program. After that, we show you how to work with text data from Jane Austen. Text data retains the ordered aspect but introduces the problem of representing words as arrays of numbers. Because a picture is worth a thousand words, we demonstrate how to work with image data. Finally, we dip into medical data with a 3D array that represents a volume containing patient anatomy.

In every section, we stop where a deep learning researcher would start: right before feeding the data to a model. We encourage you to keep these data sets around. They'll constitute excellent material when you start learning how to train neural network models.

3.1 Tabular data

The simplest form of data you'll encounter in your machine learning job is sitting in a spreadsheet, in a CSV (comma-separated values) file, or in a database. Whatever the medium, this data is a table containing one row per sample (or record), in which columns contain one piece of information about the sample.

At first, assume that there's no meaning in the order in which samples appear in the table. Such a table is a collection of independent samples, unlike a time-series, in which samples are related by a time dimension.

Columns may contain numerical values, such as temperatures at specific locations, or labels, such as a string expressing an attribute of the sample (like `"blue"`). Therefore, tabular data typically isn't homogeneous; different columns don't have the same type. You 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. Other data science packages, such as Pandas, have the concept of the *data frame*, an object representing a data set with named, heterogenous columns. By contrast, information in PyTorch is encoded

as a number, typically floating-point (though integer types are supported as well). Numeric encoding is deliberate, because 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.

Your first job as a deep learning practitioner, therefore, is to encode heterogenous, real-world data in a tensor of floating-point numbers, ready for consumption by a neural network.

A large number of tabular data sets is freely available on the internet. See https://github.com/caesar0301/awesome-public-data sets, for example.

We start with something fun: wine. The Wine Quality data set is a freely available table containing chemical characterizations of samples of *vinho verde* (a wine from northern Portugal) together with a sensory quality score. You can download the data set for white wines at https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-white.csv. For convenience, we created a copy of the data set 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; the last column contains the sensory quality score from 0 (worst) to 10 (excellent). Following are the column names in the order in which they appear in the data set:

```
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 data set is predicting the quality score from chemical characterization alone. Don't worry, though—machine learning isn't going to kill wine tasting anytime soon. We have to get the training data from somewhere!

As shown in figure 3.1, you hope to find a relationship between one of the chemical columns in your data and the quality column. Here, you're expecting to see quality increase as sulfur decreases.

Before you can get to that observation, however, you need to be able to examine the data in a more usable way than opening the file in a text editor. We'll show you how to load the data by using Python and then turn it into a PyTorch tensor.

Python offers several options for loading a CSV file quickly. Three popular options are

- The `csv` module that ships with Python
- NumPy
- Pandas

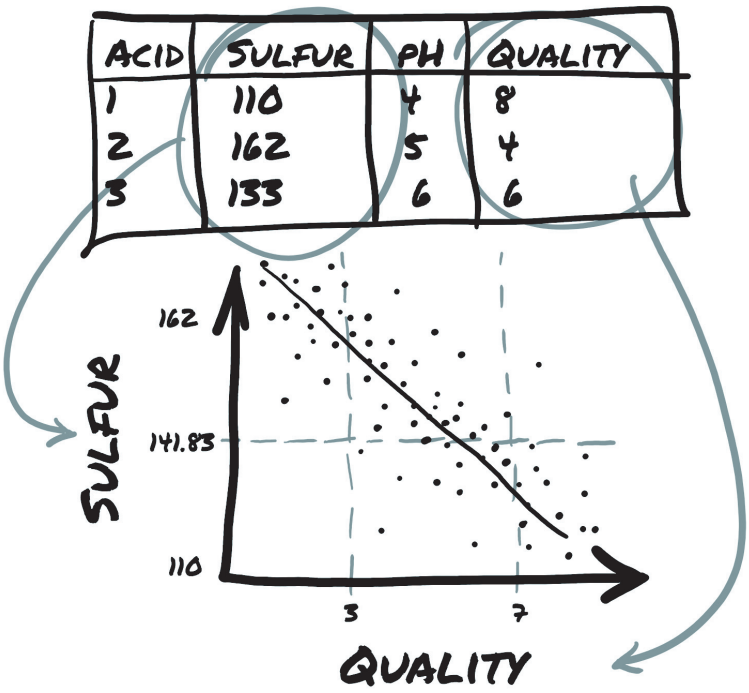


Figure 3.1 The relationship between sulfur and quality in wine

The third option is the most time- and memory-efficient, but we'll avoid introducing an additional library into your learning trajectory merely to load a file. Because we've already introduced NumPy and PyTorch has excellent NumPy interoperability, you'll go with it. Load your file and turn the resulting NumPy array into a PyTorch tensor, as shown in the following listing.

Listing 3.1 code/p1ch4/1_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=float32)
```

Here, you prescribed the type of the 2D array (32-bit floating-point) and the delimiter used to separate values in each row, and stated that the first line shouldn't be read because it contains the column names. Next, 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.type()

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

At this point, you have a `torch.FloatTensor` containing all columns, including the last, which refers to the quality score.

Interval, ordinal, and categorical values

You should be aware of three kinds of numerical values as you attempt to make sense of your data.

The first kind is *continuous* values. These values are the most intuitive when represented as numbers; they're 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 package A has a fixed meaning, no matter whether package A weighs 3 kilograms or 10, or whether B came from 200 miles away or 2,000. If you're counting or measuring something with units, the value probably is a continuous value.

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

(continued)

Finally, *categorical* values have neither ordering nor numerical meaning. These values are often enumerations of possibilities, assigned arbitrary numbers. Assigning water to `1`, coffee to `2`, soda to `3`, and milk to `4` is a good example. Placing water first and milk last has no real logic; you simply need distinct values to differentiate them. You could assign coffee to `10` and milk to `-3` with no significant change (although assigning values in the range `0..N-1` will have advantages when we discuss one-hot encoding later).

You 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 such label from the chemical analysis in a classification task. In both methods, you typically remove the score from the tensor of input data and keep it in a separate tensor, so that you can use the score as the ground truth without it being input to your model:

```
# In[5]:
data = wineq[:, :-1]
data, data.shape

# Out[5]:
(tensor([[ 7.0000,  0.2700,  ...,  0.4500,  8.8000],
         [ 6.3000,  0.3000,  ...,  0.4900,  9.5000],
         ...,
         [ 5.5000,  0.2900,  ...,  0.3800, 12.8000],
         [ 6.0000,  0.2100,  ...,  0.3200, 11.8000]]), torch.Size([4898,
     11]))
```

Select all rows and all columns except the last.

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

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

Select all rows and the last column.

If you want to transform the `target` tensor in a tensor of labels, you have two options, depending on the strategy or how you want to use the categorical data. One option is to treat a label 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 (such as *wine color*), assigning an integer number to each string would allow you to follow the same approach.

The other approach is to build a *one-hot* encoding of the scores—that is, encode each of the ten scores in a vector of ten elements, with all elements set to zero but one, at a different index for each score. This way, a score of `1` could be mapped to the vector `(1,0,0,0,0,0,0,0,0,0)`, a score of `5` to `(0,0,0,0,1,0,0,0,0,0)` and so on.

The fact that the score corresponds to the index of the nonzero element is purely incidental; you could shuffle the assignment, and nothing would change from a classification standpoint.

The two approaches have marked differences. Keeping wine-quality scores in an integer vector of scores induces an ordering of the scores, which may be appropriate in this case because a score of `1` is lower than a score of `4`. It also induces some distance between scores. (The distance between `1` and `3` is the same as the distance between `2` and `4`, for example.) If this holds for your quantity, great. If, on the other hand, scores are purely qualitative, such as color, one-hot encoding is a much better fit, as no implied ordering or distance is involved. One-hot encoding is appropriate for quantitative scores when fractional values between integer scores (such as 2.4) make no sense for the application (when score is either this or that).

You can achieve one-hot encoding by 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.]])
```

Now take a look at what `scatter_` does. First, notice that its name ends with an underscore. This convention in PyTorch indicates that the method won't return a new tensor but modify the tensor in place. The arguments for `scatter_` are

- The dimension along which the following two arguments are specified
- 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 preceding invocation reads this way: "For each row, take the index of the target label (which coincides with the score in this case), and use it as the column index to set the value `1.0`. The 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 you scatter into. Because `target_onehot` has two dimensions (4898x10), you need to add an extra dummy dimension to `target` by 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 4898 elements to a 2D tensor of size (4898x1), without changing its contents. No elements were added; you decided to use an extra index to access the elements. That is, you accessed the first element of `target` as `target[0]` and the first element of its unsqueezed counterpart as `target_unsqueezed[0,0]`.

PyTorch allows you to use class indices directly as targets while training neural networks. If you want to use the score as a categorical input to the network, however, you'd have to transform it to a *one-hot* encoded tensor.

Now go back to your `data` tensor, containing the 11 variables associated with the chemical analysis. You can use the functions in the PyTorch Tensor API to manipulate your data in tensor form. First, obtain means and standard deviations for each column:

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

# Out[10]:
tensor([6.8548e+00, 2.7824e-01, 3.3419e-01, 6.3914e+00, 4.5772e-02,
     3.5308e+01,
        1.3836e+02, 9.9403e-01, 3.1883e+00, 4.8985e-01, 1.0514e+01])
# In[11]:
data_var = torch.var(data, dim=0)
data_var

# Out[11]:
tensor([7.1211e-01, 1.0160e-02, 1.4646e-02, 2.5726e+01, 4.7733e-04,
     2.8924e+02,
        1.8061e+03, 8.9455e-06, 2.2801e-02, 1.3025e-02, 1.5144e+00])
```

In this case, `dim=0` indicates that the reduction is performed along dimension `0`. At this point, you can normalize the data by subtracting the mean and dividing by the standard deviation, which helps with the learning process.

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

# Out[12]:
tensor([[ 1.7209e-01, -8.1764e-02,  ..., -3.4914e-01, -1.3930e+00],
        [-6.5743e-01,  2.1587e-01,  ...,  1.3467e-03, -8.2418e-01],
        ...,
        [-1.6054e+00,  1.1666e-01,  ..., -9.6250e-01,  1.8574e+00],
        [-1.0129e+00, -6.7703e-01,  ..., -1.4882e+00,  1.0448e+00]])
```

Next, look at the data with an eye to finding an easy way to tell good and bad wines apart at a glance. First, use the `torch.le` function to determine which rows in `target` correspond to a score less than or equal to `3`:

```
# In[13]:
bad_indexes = torch.le(target, 3)
bad_indexes.shape, bad_indexes.dtype, bad_indexes.sum()

# Out[13]:
(torch.Size([4898]), torch.uint8, tensor(20))
```

Note that only 20 of the `bad_indexes` entries are set to 1! By leveraging a feature in PyTorch called *advanced indexing*, you can use a binary tensor to index the `data` tensor. This tensor essentially filters `data` to be only items (or rows) that correspond to `1` in the indexing tensor. The `bad_indexes` tensor has the same shape as `target`, with a value of `0` or `1` depending on the outcome of the comparison between your 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 a 1 in the `bad_indexes` tensor. It retains all 11 columns.

Now you can start to get information about wines grouped into good, middling, and bad categories. Take the `.mean()` of each column:

```
# In[15]:
bad_data = data[torch.le(target, 3)]
mid_data = data[torch.gt(target, 3) & torch.lt(target, 7)]
good_data = data[torch.ge(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
```

For numpy arrays and PyTorch tensors, the `&` operator does a logical and operation.

It looks as though you're on to something here. At first glance, the bad wines seem to have higher total sulfur dioxide, among other differences. You could use a threshold on total sulfur dioxide as a crude criterion for discriminating good wines from bad

ones. Now get the indexes in which the total sulfur dioxide column is below the midpoint you 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_threshold)

predicted_indexes.shape, predicted_indexes.dtype, predicted_indexes.sum()

# Out[16]:
(torch.Size([4898]), torch.uint8, tensor(2727))
```

Your threshold implies that slightly more than half of the wines are going to be high-quality.

Next, you need to get the indexes of the good wines:

```
# In[17]:
actual_indexes = torch.gt(target, 5)

actual_indexes.shape, actual_indexes.dtype, actual_indexes.sum()

# Out[17]:
(torch.Size([4898]), torch.uint8, tensor(3258))
```

Because you have about 500 more good wines than your threshold predicted, you already have hard evidence that the threshold isn't perfect.

Now you need to see how well your predictions line up with the actual rankings. Perform a logical `and` between your prediction indexes and the good indexes (remembering that each index is an array of `0`s and `1`s), and use that intersection of wines in agreement to determine how well you 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)
```

You got around 2,000 wines right! Because you had 2,700 wines predicted, a 74 percent chance exists that if you predict a wine to be high-quality, it is. Unfortunately, you have 3,200 good wines and identified only 61 percent of them. Well, we guess you got what you signed up for; that result is barely better than random.

This example is naïve, of course. You know for sure that multiple variables contribute to wine quality and that 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 to be more complicated than a simple threshold on a single value.

Indeed, a simple neural network would overcome all these limitations, as would a lot of other basic machine learning methods. You'll have the tools to tackle this problem after completing chapters 5 and 6, in which you build your first neural network from scratch.

3.2 Time series

In the preceding 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. Equivalently, no column encoded information on what rows came before and what rows came after.

Going back to the wine data set, you could have had a Year column that allowed you to look at how wine quality evolved year over year. (Unfortunately, we don't have such data at hand, but we're working hard on collecting the data samples manually, bottle by bottle.)

In the meantime, we'll switch to another interesting data set: data from a Washington, D.C., bike sharing system reporting the hourly count of rental bikes between 2011 and 2012 in the Capital bike-share system with the corresponding weather and seasonal information.[1]

The goal is to take a flat 2D data set and transform it into a 3D one, as shown in figure 3.2.

In the source data, each row is a separate hour of data (Figure 3.2 shows a transposed version of this to better fit on the printed page). We want to change the row-per-

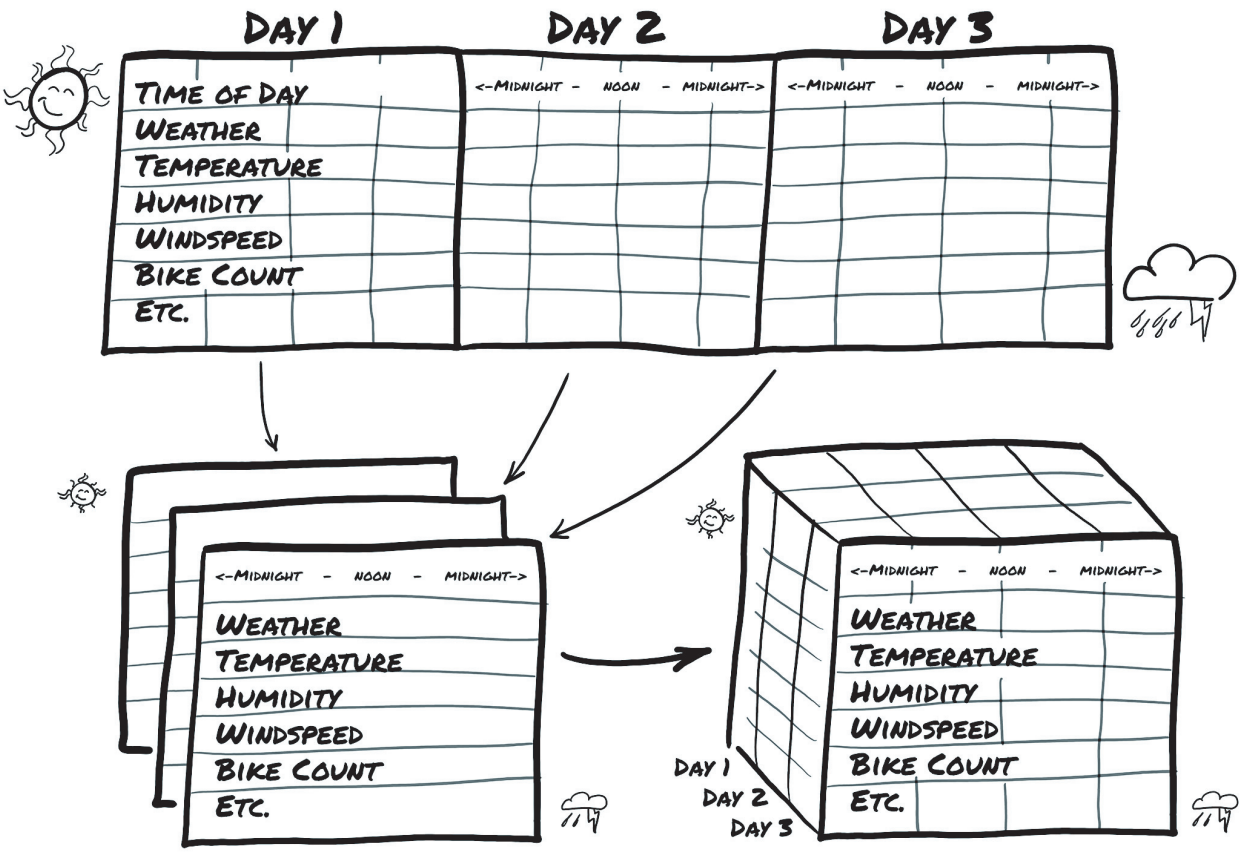


Figure 3.2 **Transforming a 1D multichannel data set into a 2D multichannel data set by separating the date and hour of each sample into separate axes**

[1] https://archive.ics.uci.edu/ml/datasets/bike+sharing+dataset

hour organization so that you have one axis that increases at a rate of one day per index increment and another axis that represents hour of day (independent of the date). The third axis is different columns of data (weather, temperature, and so on).
Load the data, as shown in the following listing.

Listing 3.2 code/p1ch4/2_time_series_bikes.ipynb

```
# In[2]:
bikes_numpy = np.loadtxt("../data/p1ch4/bike-sharing-data set/hour-
     fixed.csv",
                         dtype=np.float32,
                         delimiter=",",
                         skiprows=1,
                         converters={1: lambda x: float(x[8:10])})
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]])
```

Convert date strings to numbers corresponding to the day of the month in column 1.

For every hour, the data set reports the following variables:

```
instant      # index of record
day          # day of month
season       # season (1: spring, 2: summer, 3: fall, 4: winter)
yr           # year (0: 2011, 1: 2012)
mnth         # month (1 to 12)
hr           # hour (0 to 23)
holiday      # holiday status
weekday      # day of the week
workingday   # working day status
weathersit   # weather situation
             # (1: clear, 2:mist, 3: light rain/snow, 4: heavy rain/snow)
temp         # temperature in C
atemp        # perceived temperature in C
hum          # humidity
windspeed    # windspeed
casual       # number of causal users
registered   # number of registered users
cnt          # count of rental bikes
```

In a time-series data set such as this one, rows represent successive time points: a dimension along which they're ordered. Sure, you 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.

This existence of an ordering, however, gives you the opportunity to exploit causal relationships across time. You can predict bike rides at one time based on the fact that it was raining at an earlier time, for example. For the time being, you're going to focus

on learning how to turn your bike-sharing data set into something that your neural network can ingest in fixed-size chunks.

This neural network model needs to see sequences of values for each quantity, such as ride count, time of day, temperature, and weather conditions, so `N` parallel sequences of size `C`. `C` stands for *channel*, in neural network parlance, and is the same as *column* for 1D data like you have here. The `N` dimension represents the time axis—here, one entry per hour.

You may want to break up the 2-year data set in wider observation periods, such as days. This way, you'll have `N` (for *number of samples*) collections of `C` sequences of length `L`. In other words, your time-series data set is a tensor of dimension `3` and shape `N x C x L`. The `C` remains your 17 channels, and `L` would be `24`, one per hour of the day. 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 instead use 7*24=168 hour blocks to chunk by week instead, if we desired.

Now go back to your bike-sharing data set. The first column is the index (the global ordering of the data); the second is the date; the sixth is the time of day. You have everything you need to create a data set of daily sequences of ride counts and other exogenous variables. Your data set is already sorted, but if it weren't, you could use `torch.sort` on it to order it appropriately.

> **NOTE** The version of the file you're using here, `hour-fixed.csv`, has had some processing done to include rows that were missing from the original data set. We presumed that the missing hours had zero bikes active (typically the early-morning hours).

All you have to do to obtain your daily hours data set is view the same tensor in batches of 24 hours. Take a look at the shape and strides of your `bikes` tensor:

```
# In[3]:
bikes.shape, bikes.stride()

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

That's 17,520 hours, 17 columns. Now reshape the data to have three axes (day, hour, and then your 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, the `bikes.shape[1]` is `17`, which is the number of columns in the `bikes` tensor. But the real crux of the code is the call to `view`, which is important: it changes the way that the tensor looks at the same data as contained in storage.

Calling `view` on a tensor returns a new tensor that changes the number of dimensions and the striding information without changing the storage. As a result, you can

rearrange your tensor at zero cost because no data is copied at all. Your call to `view` requires you to provide the new shape for the returned tensor. Use the `-1` as a placeholder for "however many indexes are left, given the other dimensions and the original number of elements."

Remember that `Storage` is a contiguous, linear container for numbers—floating-point, in this case. Your `bikes` tensor has rows stored one after the other in corresponding storage, as confirmed by the output from the call to `bikes.stride()` earlier.

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

The rightmost dimension is the number of columns in the original data set. In the middle dimension, you have time split into chunks of 24 sequential hours. In other words, you now have `N` sequences of `L` hours in a day for `C` channels. To get to your desired `NxCxL` ordering, you 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))
```

We mentioned earlier that the weather-situation variable is ordinal. In fact, it has 4 levels: `1` for the best weather and `4` for the worst. You could treat this variable as categorical, with levels interpreted as labels, or continuous. If you choose categorical, you turn the variable into a one-hot encoded vector and concatenate the columns with the data set. To make rendering your data easier, limit yourself to the first day for now. First, initialize a zero-filled matrix with a number of rows equal to the number of hours in the day and a 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, 3, 2, 2, 2, 2])
```

Then scatter ones into our matrix according to the corresponding level at each row. Remember the use of `unsqueeze` to add a singleton dimension earlier:

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

# Out[7]:
```

You're decreasing the values by 1 because the weather situation ranges from 1 to 4, whereas indices are 0-based.

```
tensor([[1., 0., 0., 0.],
        [1., 0., 0., 0.],
        ...,
        [0., 1., 0., 0.],
        [0., 1., 0., 0.]])
```

The day started with weather `1` and ended with `2`, so that seems right.

Last, concatenate your matrix to your original data set, using the `cat` function. Look at the first of your 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,  0.0000,  6.0000,
          0.0000,  1.0000,  0.2400,  0.2879,  0.8100,  0.0000,  3.0000, 13.0000,
         16.0000,  1.0000,  0.0000,  0.0000,  0.0000]])
```

Here, you prescribed your original `bikes` data set and your one-hot encoded weather-situation matrix to be concatenated along the column dimension (such as `1`). In other words, the columns of the two data sets are stacked together, or the new one-hot encoded columns are appended to the original data set. For `cat` to succeed, the tensors must have the same size along the other dimensions (the row dimension, in this case).

Note that your new last four columns are `1, 0, 0, 0`—exactly what you'd expect with a weather value of `1`.

You could have done the same thing with the reshaped `daily_bikes` tensor. Remember that it's shaped `(B, C, L)`, where `L = 24`. 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 scatter the one-hot encoding into the tensor in the `C` dimension. Because operation is performed in place, only the content of the tensor changes:

```
# 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])
```

Concatenate along the `C` dimension:

```
# In[11]:
daily_bikes = torch.cat((daily_bikes, daily_weather_onehot), dim=1)
```

We mentioned earlier that this method isn't the only way to treat the weather-situation variable. Indeed, its labels have an ordinal relationship, so you could pretend that they're special values of a continuous variable. You might 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 mention in section 4.1, rescaling variables to the `[0.0, 1.0]` interval or the `[-1.0, 1.0]` interval is something that you'll want to do for all quantitative variables, such as `temperature` (column 10 in your data set). You'll see why later; for now, we'll say that it's beneficial to the training process.

You have multiple possibilities for rescaling variables. You can 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 this latter case, the variable has zero mean and unitary standard deviation. If the variable were drawn from a Gaussian distribution, 68 percent of the samples would sit in the `[-1.0, 1.0]` interval.

Great—you've built another nice data set that you'll get to use later. For now, it's important only that you got an idea of how a time series is laid out and how you can wrangle the data into a form that a network will digest.

Other kinds of data look like a time series, in that strict ordering exists. The top two in that category are text and audio.

3.3 Text

Deep learning has taken the field of natural language processing (NLP) by storm, particularly by using models that repeatedly consume a combination of new input and previous model output. These models are called *recurrent neural networks*, and they've been applied with great success to text categorization, text generation, and automated translation systems. Previous NLP workloads were characterized by sophisticated multistage pipelines that included rules encoding the grammar of a language.[2] [3] Now,

[2] Nadkarni et al., "Natural language processing: an introduction". JAMIA https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3168328

[3] Wikipedia entry for natural language processing: https://en.wikipedia.org/wiki/Natural-language_processing

state-of-the-art work trains networks end to end on large corpuses starting from scratch, letting those rules emerge from data. For the past several years, the most-used automated translation systems available as services on the internet have been based on deep learning.

Your goal in this chapter is to turn text into something that a neural network can process, which, like the previous cases, is a tensor of numbers. If you can do that and later choose the right architecture for your text processing job, you'll be in the position of doing NLP with PyTorch. You see right away how powerful this capability is: you can achieve state-of-the-art performance on tasks in different domains *with the same PyTorch tools* if you cast your problem in the right form. The first part of this job is reshaping data.

Networks operate on text at two levels: at character level, by processing one character at a time, and at word level, in which individual words are the finest-grained entities seen by the network. The technique you use to encode text information into tensor form is the same whether you operate at character level or at word level. This technique is nothing magic; you stumbled upon it earlier. It's one-hot encoding.

Start with a character-level example. First, get some text to process. An amazing resource is Project Gutenberg[4], a volunteer effort that digitizes and archives cultural work and makes it available for free in open formats, including plain-text files. If you'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. You can find several other corpora at the English Corpora website.[5]

Load Jane Austen's *Pride and Prejudice* from the Project Gutenberg website.[6] Save the file and read it in, as shown in the following listing.

Listing 3.3 code/p1ch4/3_text_jane_austin.ipynb

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

You need to take care of one more detail before you proceed: encoding. Encoding is a vast subject, so all we'll do now is touch on it. Every written character is represented by a code, a sequence of bits of appropriate length that allow each character to be uniquely identified. The simplest such encoding is ASCII (American Standard Code for Information Interchange), dating back to the 1960s. ASCII encodes 128 characters using 128 integers. Letter *a*, for example, corresponds to binary `1100001` or decimal 97; letter *b* corresponds to binary `1100010` or decimal 98, and so on. The encoding would fit 8 bits, which was a big bonus in 1965.

[4] http://www.gutenberg.org
[5] https://www.english-corpora.org
[6] http://www.gutenberg.org/files/1342/1342-0.txt

NOTE Clearly, 128 characters aren't enough to account for all the glyphs, accents, ligatures, and other features that are needed to properly represent written text in languages other than English. To this end, other encodings have been developed, using a larger number of bits as a code for a wider range of characters. That wider range of characters got standardized as Unicode, which maps all known characters to numbers, with the representation in bits of those numbers being provided by a specific encoding. Popular encodings include `UTF-8`, `UTF-16` and `UTF-32`, in which the numbers are a sequence of 8-, 16-, or 32-bit integers. Strings in Python 3.x are Unicode strings.

You're going to one-hot encode your characters to limit the one-hot encoding to a character set that's useful for the text being analyzed. In this case, because you loaded text in English, it's quite safe to use ASCII and deal with a small encoding. You could also make all characters lowercase to reduce the number of characters in your encoding. Similarly, you could screen out punctuation, numbers, and other characters that aren't relevant to the expected kinds of text, which may or may not make a practical difference to your neural network, depending on the task at hand.

At this point, you need to parse 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 characters in the encoding. This vector will contain all zeros except for a `1` at the index corresponding to the location of the character in the encoding.

First, split your `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 with him'
```

Create a tensor that can hold the total number of one-hot encoded characters for the whole line:

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

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

128 hardcoded due to the limits of ASCII

Note that `letter_tensor` holds a one-hot encoded character per row. Now set a `1` on each row in the right position so that each row represents the right character. The index where the `1` 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
    letter_tensor[i][letter_index] = 1
```

The text uses directional double quotes, which aren't valid ASCII, so screen them out here.

You've one-hot encoded your sentence into a representation that a neural network can digest. You could do word-level encoding the same way by establishing a vocabulary and one-hot encoding sentences, sequences of words, along the rows of your tensor. Because a vocabulary contains many words, this method produces wide encoded vectors that may not be practical. Later in this chapter, you see a more efficient way to represent text at word level by using embeddings. For now, stick with one-hot encodings to see what happens.

Define `clean_words`, which takes text and returns it lowercase and stripped of punctuation. When you call it on your "Impossible, Mr. Bennet" `line`, you 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 with him',
 ['impossible',
  'mr',
  'bennet',
  'impossible',
  'when',
  'i',
  'am',
  'not',
  'acquainted',
  'with',
  'him'])
```

Next, build a mapping of words to indexes in your 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 `all_words` is now a dictionary with words as keys and an integer as value. You'll use this dictionary to efficiently find the index of a word as you one-hot encode it.

Now focus on your sentence. Break it into words and one-hot encode it—that is, populate a tensor with one one-hot encoded vector per word. Create an empty vector, and assign the one-hot encoded values of the word in the sentence:

```
# In[8]:
word_tensor = torch.zeros(len(words_in_line), len(word2index_dict))
for i, word in enumerate(words_in_line):
```

```
    word_index = word2index_dict[word]
    word_tensor[i][word_index] = 1
    print('{:2} {:4} {}'.format(i, word_index, word))

print(word_tensor.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 `7261`—the number of words in your dictionary.

3.3.1 Text embeddings

One-hot encoding is a useful technique for representing categorical data in tensors. As you may have anticipated, however, one-hot encoding starts to break down when the number of items to encode is effectively unbound, as with words in a corpus. In one book, you had more than 7,000 items!

You certainly could do some work to deduplicate words, condense alternative spellings, collapse past and future tenses into a single token, and that kind of thing. Still, a general-purpose English-language encoding is going to be *huge*. Worse, every time you encounter a new word, you have to add a new column to the vector, which means adding a new set of weights to the model to account for that new vocabulary entry, which is going to be painful from a training perspective.

How can you compress your encoding to a more manageable size and put a cap on the size growth? Well, instead of using vectors of many zeros and a single `1`, you could 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 to this 100-dimensional space in a way that facilitates downstream learning. This technique is called *embedding*.

In principle, you could iterate over your vocabulary and generate a set of 100 random floating-point numbers for each word. This method would work, in that you could cram a large vocabulary into 100 numbers, but it would forgo any concept of distance between words based on meaning or context. A model that used this word embedding would have to deal with 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 map to nearby regions of the embedding.

If you were to design a solution to this problem by hand, you might decide to build your embedding space by mapping basic nouns and adjectives along the axes. You can generate a 2D space in which axes map to nouns `"fruit"` (`0.0`–`0.33`), `"flower"` (`0.33`–`0.66`), and `"dog"` (`0.66`–`1.0`), and to 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`). Your goal now is to take actual fruit, flowers, and dogs and lay them out in the embedding.

As you start embedding words, you can map `"apple"` to a number in the `"fruit"` and `"red"` quadrant. Likewise, you can easily map `"tangerine"`, `"lemon"`, `"lychee"`, and `"kiwi"` (to round out your list of colorful fruits). Then you can start on flowers, assigning `"rose"`, `"poppy"`, `"daffodil"`, `"lily"`, and . . . well, there aren't many brown flowers out there. Well, `"sunflower"` can get `"flower"`, `"yellow"`, and `"brown"`, and `"daisy"` can get `"flower"` `"white"`, and "`yellow`". Perhaps you should update `"kiwi"` to map close to `"fruit"`, `"brown"`, and `"green"`. For dogs and color, you can embed `"redbone"`, `"fox"` perhaps for `"orange"`, `"golden retriever"`, `"poodles"` for `"white"`, and . . . most kinds of dogs are `"brown"`.

Although doing this mapping manually isn't feasible for a large corpus, you should note that although you had an embedding size of `2`, you described 15 different words *besides the base 8* and probably could cram quite a few more in if you take the time to be creative.

As you've probably guessed, this kind of work can be automated. By processing a large corpus of organic text, you can generate embeddings similar to this one. The main differences are that the embedding vector has 100 to 1,000 elements and that axes don't map directly to concepts, but conceptually similar words map to neighboring regions of an embedding space whose axes are arbitrary floating-point dimensions.

Although the exact algorithms[7] used are a bit out of scope for what we wanting to focus on here, we'd like to mention that embeddings are often generated by using neural networks, trying to predict a word from nearby words (the context) in a sentence. In this case, you could start from one-hot encoded words and use a (usually rather shallow) neural network to generate the embedding. When the embedding is available, you 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 with consistent spatial relationships with other words. If you were to take the embedding vector for `"apple"` and begin to add and subtract the vectors for other words, you could begin to perform analogies such as `apple - red - sweet + yellow + sour` and end up with a vector similar to the one for `"lemon"`.

We won't be using text embeddings here, but they're essential tools when a large number of entries in a set has to be represented with numeric vectors.

---

[7] One example is https://en.wikipedia.org/wiki/Word2vec

3.4 Images

The introduction of convolutional neural networks revolutionized computer vision[8], and image-based systems have since acquired a new set of capabilities. Problems that required complex pipelines of highly tuned algorithmic building blocks became solvable at unprecedented levels of performance by training end-to-end networks with paired input-and-desired-output examples. To participate in this revolution, you need to be able to load images 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 that PyTorch expects.

An image is represented as a collection of scalars arranged in a regular grid, having a height and a width (in pixels). You 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 typically represent different colors or different *features*, such as depth from a depth camera.

Scalars representing values at individual pixels are often encoded with 8-bit integers, as in consumer cameras, for example. In medical, scientific, and industrial applications, you not infrequently find pixels with higher numerical precision, such as 12-bit and 16-bit. This precision provides a wider range or increased sensitivity in cases in which the pixel encodes information on a physical property, such as bone density, temperature, or depth.

You have several ways of encoding numbers into colors.[9] The most common is RGB, which defines a color with three numbers that represent the intensity of red, green and blue. You can think of a color channel as being a grayscale intensity map of only the color in question, similar to what you'd see if you looked at the scene in question through a pair of pure-red sunglasses. Figure 3.3 shows a rainbow in which each of the RGB channels captures a certain portion of the spectrum. (The figure is simplified, in that it elides things. The orange and yellow bands, for example, are represented as a combination of red and green.)

Images come in several file formats, but luckily, you have plenty of ways to load images in Python. Start by loading a PNG image with the `imageio` module. You'll use `imageio` throughout the chapter because it handles different data types with a uniform API. Now load an image, as in the following listing.

Listing 3.4 code/p1ch4/5_image_dog.ipynb

```
# In[2]:
import imageio

img_arr = imageio.imread('../data/p1ch4/image-dog/bobby.jpg')
img_arr.shape

# Out[2]:
(720, 1280, 3)
```

[8] https://en.wikipedia.org/wiki/Convolutional_neural_network#History

[9] Something of an understatement: https://en.wikipedia.org/wiki/Color_model

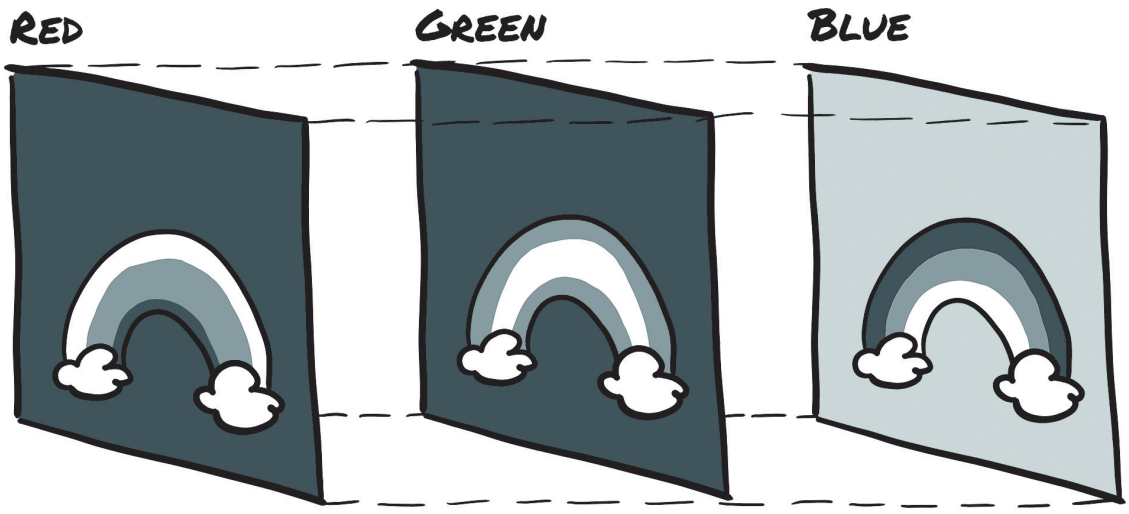


Figure 3.3 **A rainbow broken into red, green, and blue channels**

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 channels red, green, and blue. Any library that outputs a NumPy array does so to obtain a PyTorch tensor. The only thing to watch out for is the layout of dimensions. PyTorch modules that deal with image data require tensors to be laid out as `C x H x W` (channels, height, and width, respectively).

You can use the `transpose` function to get to an appropriate layout. Given an input tensor `W x H x C`, you get to a proper layout by swapping the first and last channels:

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

You've seen this example before, but note that this operation doesn't make a copy of the tensor data. Instead, `out` uses the same underlying storage as `img` and plays with the size and stride information at the tensor level. This arrangement is convenient because the operation is cheap, but (heads up) changing a pixel in `img` leads to a change in `out`.

Also note that other deep learning frameworks use different layouts. Originally, TensorFlow kept the channel dimension last, resulting in a `H x W x C` layout. (Now it supports multiple layouts.) This strategy has pros and cons from a low-level performance standpoint, but it doesn't make a difference to you as long as you reshape your tensors properly.

So far, you've described a single image. Following the same strategy that you used for earlier data types, to create a data set of multiple images to use as an input for your neural networks, you store the images in a batch along the first dimension to obtain a `N x C x H x W` tensor.

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

```
# In[4]:
batch_size = 100
batch = torch.zeros(100, 3, 256, 256, dtype=torch.uint8)
```

which indicates that your batch will consist of 100 RGB images 256 pixels in height and 256 pixels in width. Notice the type of the tensor: you're expecting each color to be represented as a 8-bit integer, as in most photographic formats from standard consumer cameras. Now you can 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)
     == '.png']
for i, filename in enumerate(filenames):
    img_arr = imageio.imread(filename)
    batch[i] = torch.transpose(torch.from_numpy(img_arr), 0, 2)
```

As we mentioned earlier, neural networks usually work with floating-point tensors as their input. As you'll also see in upcoming chapters, neural networks exhibit the best training performance when input data ranges from roughly `0` to `1` or `-1` to `1` (an effect of how their building blocks are defined).

A typical thing that you'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 you decide should lie between `0` and `1` (or `-1` and `1`). One possibility is to divide the values of pixels by 255 (the maximum representable number in 8-bit unsigned):

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

Another possibility is to compute 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
```

You can perform several other operations on inputs, including geometric transformations such as rotation, scaling, and cropping. These operations may help with training or may be required to make an arbitrary input conform to the input requirements of a network, such as the size of the image. You'll stumble onto quite a few of these strategies. For now, just remember that you have image manipulation options available.

3.5 Volumetric data

You've learned how to load and represent 2D images, like the ones you take with your camera. In contexts such as medical imaging applications involving, say, CT (Computed Tomography) scans, you typically deal with sequences of images stacked along the head-to-feet direction, each corresponding to a slice across the body. In CT scans, the intensity represents the density of the different parts of the body: lungs, fat, water, muscle, bone, in order of increasing density, mapped from dark to bright when CT scans are displayed on clinical workstations. The density at each point is computed from the amount of x-ray reaching a detector after passing through the body, with some complex math used to deconvolve the raw sensor data into the full volume.

CTs have a single intensity channel, similar to a grayscale image. Often, in native data formats, the channel dimension is left out, so the raw data typically has three dimensions. By stacking individual 2D slices into a 3D tensor, you can build volumetric data representing the 3D anatomy of a subject. Unlike figure 3.3, the extra dimension in figure 3.4 represents an offset in physical space rather than a particular band of the visible spectrum.

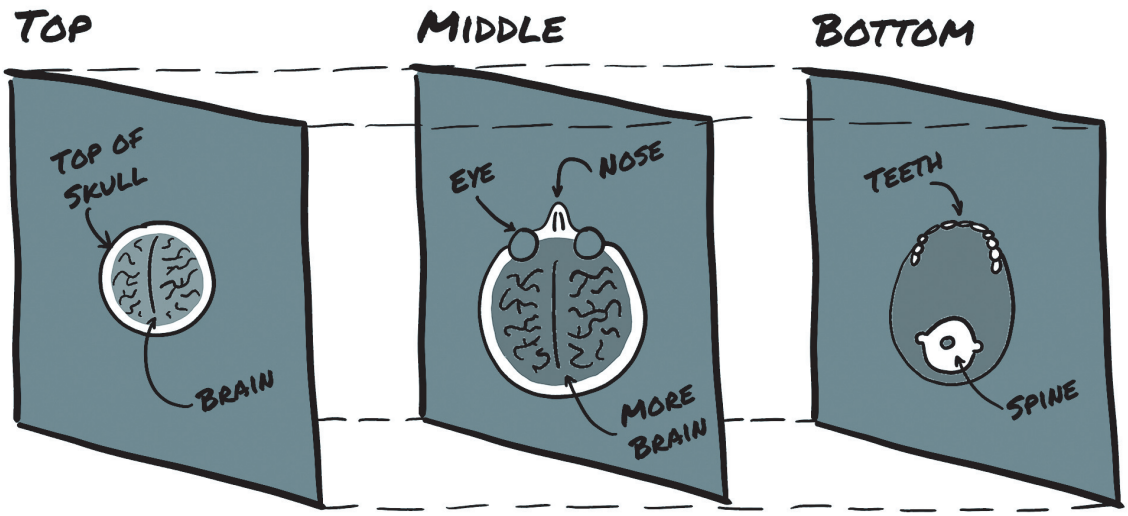


Figure 3.4 **Slices of a CT scan, from the top of the head to the jawline**

We won't go into detail here on medical imaging data formats. For now, it suffices to say that no fundamental difference exists between a tensor that stores volumetric data and one that stores image data. You have an extra dimension, *depth*, after the *channel* dimension, leading to a 5D tensor of shape `N x C x D x H x W`.

Load a sample CT scan by using the `volread` function in the `imageio` module, which takes a directory as argument and assembles all DICOM (Digital Imaging Communication and Storage) files[10] in a series in a NumPy 3D array, as shown in the following listing.

[10] https://wiki.cancerimagingarchive.net/display/Public/CPTAC-LSCC#dd4a08a246524596add33b9f8f00f288

Listing 3.5 code/p1ch4/6_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 (100.0%)
  Found 1 correct series.
Reading DICOM (loading data): 87/99  (87.999/99  (100.0%)

# Out[2]:
(99, 512, 512)
```

Also in this case, the layout is different from what PyTorch expects, due to the lack of channel information. You'll have to make room for the `channel` dimension by using `unsqueeze`:

```
# In[3]:
vol = torch.from_numpy(vol_arr).float()
vol = torch.transpose(vol, 0, 2)
vol = torch.unsqueeze(vol, 0)

vol.shape

# Out[3]:
torch.Size([1, 512, 512, 99])
```

At this point, you could assemble a 5D data set by stacking multiple volumes along the `batch` direction, as you did earlier in the chapter.

Conclusion

You covered a lot of ground in this chapter. You learned to load the most common types of data and shape them up for consumption by a neural network. There are more data formats in the wild than we could hope to describe in a single volume, of course. Some, like medical histories, are too complex to cover in this volume. For the interested reader, however, we do provide short examples of audio and video tensor creation in bonus Jupyter notebooks in our code repository[11].

Exercises

- Take several pictures of red, blue, and green items with your phone or other digital camera.[12]
  - Load each image, and convert it to a tensor.
  - For each image tensor, use the `.mean()` method to get a sense of how bright the image is.

[11] https://github.com/deep-learning-with-pytorch/dlwpt-code/tree/master/p1ch4

[12] Or download some from the internet if a camera isn't available.

  - Now take the mean of each channel of your images. Can you identify the red, green, and blue items from only the channel averages?
- Select a relatively large file containing Python source code.
  - 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 by replacing `r"[^a-zA-Z0-9_]+"` with spaces.)
  - Compare your index with the one you made for *Pride and Prejudice*. Which is larger?
  - Create the one-hot encoding for the source code file.
  - What information is lost with this encoding? How does that information compare with what's lost in the *Pride and Prejudice* encoding?

Summary

- Neural networks require data to be represented as multidimensional numerical tensors, often 32-bit floating-point.
- 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.
- In general, PyTorch expects data to be laid out along specific dimensions, according to the model architecture (such as convolutional versus recurrent). Data reshaping can be achieved effectively with the PyTorch tensor API.
- Spreadsheets can be straightforward to convert to tensors. 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.
- Images can have one or many channels. The most common are the red, green, and blue channels of typical digital photos.
- Single-channel data formats sometimes omit an explicit channel dimension.
- Volumetric data is similar to 2D image data, with the exception of adding a third dimension: depth.
- Many images have a per-channel bit depth of 8, though 12 and 16 bits per channel are not uncommon. These bit-depths can be stored in a 32-bit floating-point number without loss of precision.

4 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 from scratch
- Seeing how PyTorch supports learning with autograd

With the blooming of machine learning that has occurred over the past 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 it, or the *algorithm* behind it? From the point of view of an outer observer, a learning algorithm is presented input data that is paired with desired outputs. When learning has occurred, that algorithm is capable of producing correct outputs when it's fed new data that is similar enough to the input data on which it was trained. 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, such as an image and a sentence describing it.

As a matter of fact, models that allow you to explain input/output relationships date back centuries. When Johannes Kepler, a German mathematical astronomer

who lived between 1571 and 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, naked eye and a piece of paper). Not having Newton's Law of gravitation at his disposal (in fact, Newton used Kepler's work to figure things out), he extrapolated the simplest possible geometric model that could fit the data. By the way, it took him six years of staring at data that didn't make sense to him, as well as incremental realizations, to formulate these laws.[1] You can see this process in figure 4.1.

The 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, such as Jupiter), he could at that point 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 could possibly be during its journey in the sky. When 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.[2]

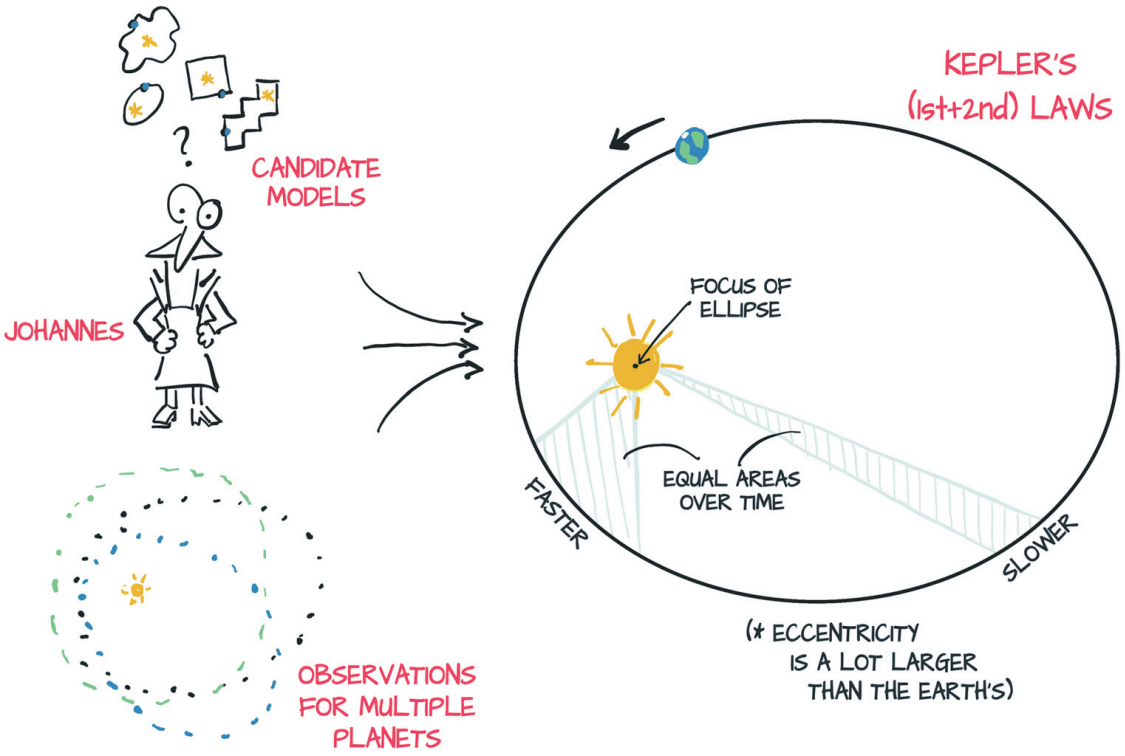


Figure 4.1 Johannes Kepler considers multiple candidate models that might fit the data at hand, settling on an ellipse.

[1] As recounted by Michael Fowler: http://galileoandeinstein.physics.virginia.edu/1995/lectures/morekepl.html

[2] Understanding the details of Kepler's laws isn't needed for understanding the chapter, but you can find more information at https://en.wikipedia.org/wiki/Kepler%27s_laws_of_planetary_motion.

How would Kepler estimate the eccentricity and size of the ellipse without computers, pocket calculators or even calculus, none of which had been invented yet? You learn the answer from Kepler's own recollection in his book *New Astronomy* or from how J.V. Field put it in his *The Origins of Proof* series:[3]

> *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.*

To sum things up, over those 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 that data because he felt that something fishy was 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, 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 constructed on these seven steps, and as scientists have learned over the centuries, deviating from them is a recipe for disaster.[4]

These steps are exactly what you'll follow to *learn* something from data. Here, there's virtually no difference between saying that you'll *fit* the data and saying that you'll make an algorithm *learn* from data. The process always involves a function with unknown parameters whose values are estimated from data—in short, a *model*.

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

You're interested here in the latter kinds of models: one that aren't engineered to solve specific narrow tasks and can be adapted automatically to specialize in solving many similar tasks, using input and output pairs—in other words, general models trained on data relevant to the 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. Don't worry if the last sentence didn't make sense; section 1.1 should clear it up for you.

---

[3] https://plus.maths.org/content/origins-proof-ii-keplers-proofs

[4] Unless you're a theoretical physicist ;)

This chapter is about how to automate this generic function-fitting, which is all you do with deep learning, deep neural networks being the generic functions, and PyTorch makes this process as simple and transparent as possible. To make sure that you get the key concepts right and to allow you to understand the mechanics of learning algorithms from first principles, we'll start with a model that's a lot simpler than a deep neural network.

4.1 Learning is parameter estimation

In this section, you learn how you can take data, choose a model, and estimate the parameters of the model so that it gives good predictions on new data. To do so, you'll leave the intricacies of planetary motion and divert your attention to the second-hardest problem in physics: calibrating instruments.

Figure 4.2 shows a high-level overview of what you'll have implemented 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 with the ground truth. To optimize the parameter of the model, its *weights*—the change in the error following a unit change in weights (the gradient of the error with respect to the parameters)—is computed by using the chain rule for the derivative of a composite function (backward pass). Then the value of the weights is 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 this sounds obscure, we've got a whole chapter to clarify things. By the time we're done, all the pieces will fall into place, and the preceding paragraph will make perfect sense to you.

Next, you take a problem with a noisy data set, build a model, and implement a learning algorithm for it. You'll start by doing everything by hand, but by the end of the chapter, you'll be letting PyTorch do all the heavy lifting. By the end of the chapter, we'll have covered many of the essential concepts that underlie training deep neural networks, even if the motivating example is simple and the model isn't a neural network (yet!).

4.1.1 A hot problem

Suppose that you took a trip to some obscure location and brought back a fancy, wall-mounted analog thermometer. It looks great, it's a perfect fit for your living room. Its only flaw is that it doesn't show units. Not to worry; you've got a plan. You'll build a data set of readings and corresponding temperature values in your favorite units, choose a model, and adjust its weights iteratively until a measure of the error is low enough, and you'll finally be able to interpret the new readings in units you understand.

Start by making a note of temperature data in good old Celsius[5] and measurements from your new thermometer.

[5] Luca is Italian, so please forgive him for using sensible units.

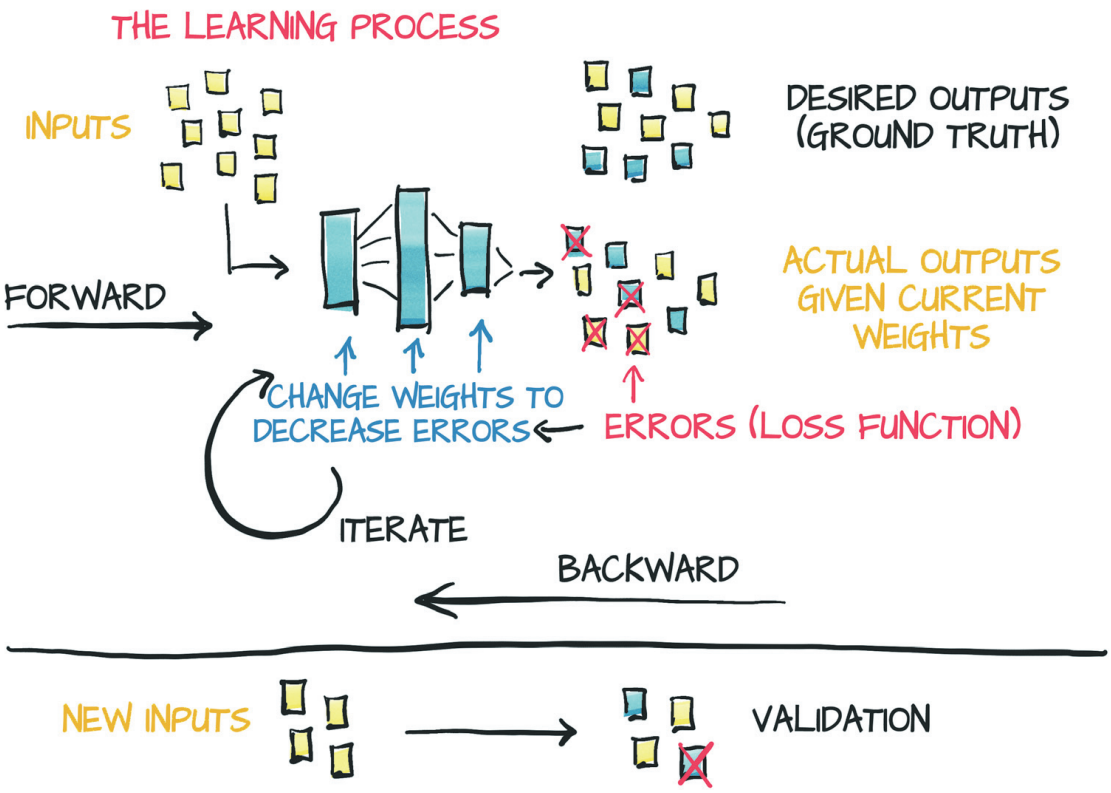


Figure 4.2 Mental model of the learning process

After a couple of weeks, here's the data:

Listing 4.1 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, 21.0]
t_u = [35.7, 55.9, 58.2, 81.9, 56.3, 48.9, 33.9, 21.8, 48.4, 60.4, 68.4]
t_c = torch.tensor(t_c)
t_u = torch.tensor(t_u)
```

`t_c` are temperatures in Celsius, and `t_u` are the unknown units. You can expect noise in both measurements coming from the devices themselves and from your approximate readings. For convenience, the data is already in tensors, which you'll use soon.

4.1.2 Choosing a linear model as a first try

In the absence of further knowledge, assume the simplest possible model for converting between the two sets of measurements, as Kepler might have done. The two sets may be linearly related. That is, multiplying `t_u` by a factor and adding a constant, you may get the temperature in Celsius:

```
t_c = w * t_u + b
```

Is this assumption reasonable? Probably; you'll see how well the final model performs. (You chose to name `w` and `b` after *weight* and *bias*, two common terms for linear scaling and the additive constant, which you'll bump into all the time.)

> **NOTE** Spoiler alert: We know that a linear model is correct because the problem and data have been fabricated, but please bear with us; this model is a useful motivating example to build your understanding of what PyTorch is doing under the hood.

Now you need to estimate `w` and `b`, the parameters in the model, based on the data you have. You must do this so that the temperatures you obtain from running the unknown temperatures `t_u` through the model are close to temperatures you measured in Celsius. If that process sounds like fitting a line through a set of measurements, that's exactly what you're doing. As you go through this simple example using PyTorch, realize that training a neural network essentially involves changing the model for a slightly more elaborate one with a few (or a metric ton) more parameters.

To flesh out the example again, you have a model with some unknown parameters, and you need to estimate those parameters so that the error between predicted outputs and measured values is as low as possible. You notice that you still need to define a measure of such error. Such 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. Your optimization process, therefore, should aim at finding `w` and `b` so that the loss function is at a minimum level.

4.1.3 Less loss is what you want

A *loss function* (or *cost function*) is a function that computes a single numerical value that the learning process attempts to minimize. The calculation of loss typically involves taking the difference between the desired outputs for some training samples and those produced by the model when fed those samples—in this case, the difference between the predicted temperatures `t_p` output by the model and the actual measurements, so `t_p - t_c`.

You need to make sure the loss function makes the loss positive both when `t_p` is above and when below the true `t_c`, because the goal is to minimize this value. (Being able to push the loss infinitely negative isn't useful.) You have a few choices, the most straightforward being `|t_p - t_c|` and `(t_p - t_c)^2`. Based on the mathematical expression you choose, you can emphasize or discount certain errors. Conceptually, a loss function is a way of prioritizing which errors to fix from your training samples, so that your 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 in zero and grow monotonically as the predicted value moves farther from the true value in either direction. For this reason, both functions are said to be *convex*. Because your model is linear, the loss as a function of `w` and `b` is also convex. Cases in which the loss is a convex function of the model parameters are usually great to deal with because you can find a minimum in an efficient way through specialized algorithms. Deep neural networks don't exhibit a convex loss, however, so those methods aren't generally useful to you.

For the two loss functions `|t_p - t_c|` and `(t_p - t_c)^2`, as shown in figure 4.3, notice that the square of 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 contrary, has an undefined derivative right where you'd like to converge. This issue is less important than it looks in practice, but stick to the square of differences for the time being.

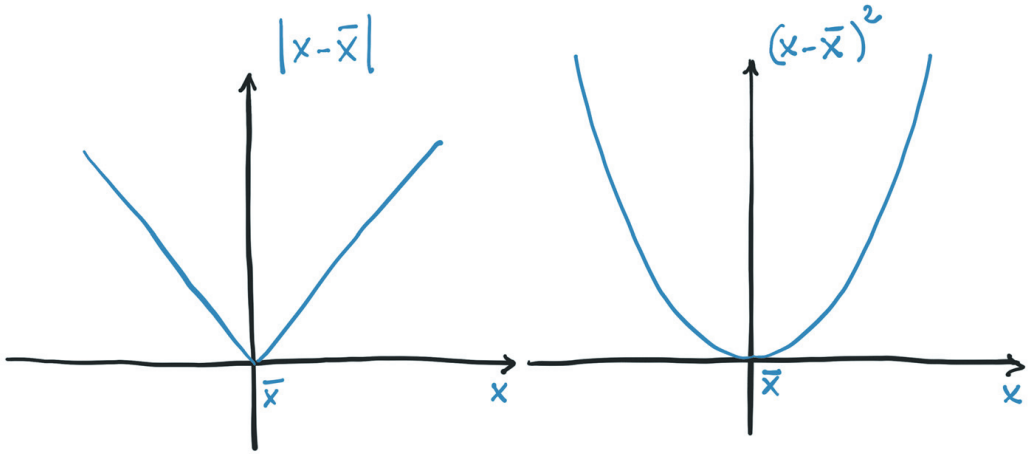


Figure 4.3 Absolute difference versus difference squared

It's worth noting that the square difference also penalizes wildly wrong results more than the absolute difference. Often, having more slightly wrong results is better than having a few wildly wrong ones, and the squared difference helps prioritize those results as desired.

4.1.4 From problem to PyTorch

You've figured out the model and the loss function, so you've already got a good part of the high-level picture figured out. Now you need to set the learning process in motion and feed it actual data. Also, enough with math notation already, so now switch to PyTorch. After all, you came here for the *fun*.

You've already created your data tensors, so write out the model as a Python function

```
# In[3]:
def model(t_u, w, b):
    return w * t_u + b
```

in which you're expecting `t_u`, `w`, and `b` to be the input tensor, weight parameter, and bias parameter, respectively. In your model, the parameters will be PyTorch scalars (aka zero-dimensional tensors), and the product operation will use broadcasting to yield the returned tensors. Now define your loss:

```
# In[4]:
def loss_fn(t_p, t_c):
    squared_diffs = (t_p - t_c)**2
    return squared_diffs.mean()
```

Note that you're building a tensor of differences, taking their square elementwise and finally producing a scalar loss function by averaging all elements in the resulting tensor. The loss is a *mean square loss*.

Now you can initialize the parameters, invoke the model,

```
# In[5]:
w = torch.ones(1)
b = torch.zeros(1)

t_p = model(t_u, w, b)
t_p

# Out[5]:
tensor([35.7000, 55.9000, 58.2000, 81.9000, 56.3000, 48.9000, 33.9000, 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)
```

In this section, you implemented the model and the loss. The meat of the section is how to estimate the `w` and `b` such that the loss reaches a minimum. First, you work things out by hand; then you learn how to leverage PyTorch superpowers to solve the same problem in a more general, off-the-shelf way.

4.1.5 Down along the gradient

In this section, you optimize the loss function with respect to the parameters by using the so-called *gradient descent* algorithm and build your intuition about how gradient descent works from first principles, which will help you a lot in the future. There are ways to solve this particular example more efficiently, but those approaches aren't applicable to most deep learning tasks. Gradient descent is a simple idea that scales up surprisingly well to large neural network models with millions of parameters.

Start with the mental image conveniently sketched out in figure 4.4. Suppose that you're in front of a machine sporting two knobs, labeled `w` and `b`. You're allowed to see the value of the loss on a screen and are told to minimize that value. Not knowing the effect of the knobs on the loss, you'd probably start fiddling with them and decide for each knob what direction makes the loss decrease. You'd probably decide to rotate both knobs in their direction of decreasing loss. If you're far from the optimal value, you're likely to see the loss decrease quickly and then slow as it gets closer to the minimum. You'd notice that at some point, the loss climbs back up again, so you'd invert the direction of rotation for one or both knobs. You'd 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, you'd converge to a minimum.

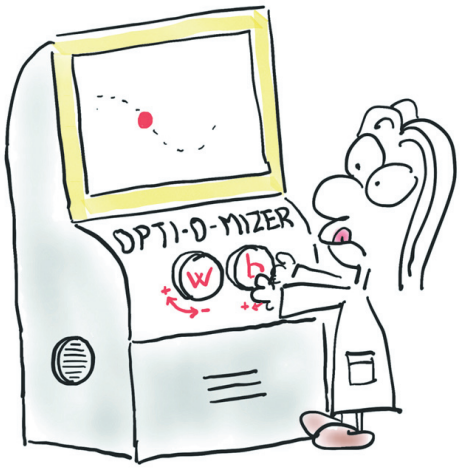


Figure 4.4 A cartoon depiction of the optimization process, in which a person with knobs for `w` and `b` searches for the direction that makes the loss decrease

Gradient descent isn't too different. The idea is to compute the rate of change of the loss with respect to each parameter and apply a change to each parameter in the direction of decreasing loss. As when you were fiddling with the knobs, you could estimate such rate of change by applying a small change to `w` and `b` to see how much the loss is changing in that neighborhood:

```
# In[7]:
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 code is saying that in a small 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, you need to increase `w` to minimize the loss, whereas if the change is positive, you need to decrease `w`. By how much? Applying a change to `w` that's proportional to the rate of change of the loss is a good idea, especially when the loss has several parameters: you'd apply a change to those that exert a significant change on the loss. It's 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, you should scale the rate of change by a typically small factor. This scaling factor has many names; the one used in machine learning is `learning_rate`.

```
# In[8]:
learning_rate = 1e-2

w = w - learning_rate * loss_rate_of_change_w
```

You can do the same with `b`:

```
# In[9]:
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 code represents the basic parameter update step for gradient descent. By reiterating these evaluations (provided that you choose a small-enough learning rate), you'd converge to an optimal value of the parameters for which the loss computed on the given data is minimal. We'll show you the complete iterative process soon, but this method of computing rates of change is rather crude and needs an upgrade. In the next section, you see why and how.

4.1.6 Getting analytical

Computing the rate of change by using repeated evaluations of 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 isn't always clear how large that neighborhood should be. You chose `delta` equal to `0.1` earlier, but everything depends on the shape of the loss as a function of `w` and `b`. If the loss changes too quickly compared with `delta`, you won't have a good idea of where downhill is.

What if you could make the neighborhood infinitesimally small, as in figure 4.5? That's exactly what happens when you take the derivative of the loss with respect to a parameter analytically. In a model with two or more parameters, you compute the individual derivatives of the loss with respect to each parameter and put them in a vector of derivatives: the *gradient*.

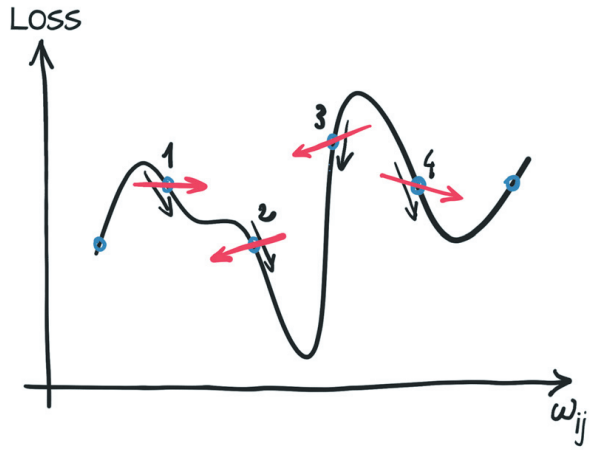


Figure 4.5 Differences in the estimated downhill directions when evaluating them at discrete locations versus analytically

To compute the derivative of the loss with respect to a parameter, you 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:

```
d loss_fn / d w = (d loss_fn / d t_p) * (d t_p / d w)
```

Recall that the model is a linear function and the loss is a sum of squares. Now 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()
```

and remembering that `d x^2 / d x = 2 x`, you get

```
# In[10]:
def dloss_fn(t_p, t_c):
    dsq_diffs = 2 * (t_p - t_c)
    return dsq_diffs
```

As for the model, recalling that the model is

```
# In[3]:
def model(t_u, w, b):
    return w * t_u + b
```

you get derivatives of

```
# In[11]:
def dmodel_dw(t_u, w, b):
    return t_u


# In[12]:
def dmodel_db(t_u, w, b):
    return 1.0
```

Putting all this together, the function that returns the gradient of the loss with respect to `w` and `b` is

```
# In[13]:
def grad_fn(t_u, t_c, t_p, w, b):
    dloss_dw = dloss_fn(t_p, t_c) * dmodel_dw(t_u, w, b)
    dloss_db = dloss_fn(t_p, t_c) * dmodel_db(t_u, w, b)
    return torch.stack([dloss_dw.mean(), dloss_db.mean()])
```

The same idea expressed in mathematical notation is shown in figure 4.6.

Again, you're averaging (summing and dividing by a constant) over all data points to get a single scalar quantity for each partial derivative of the loss.

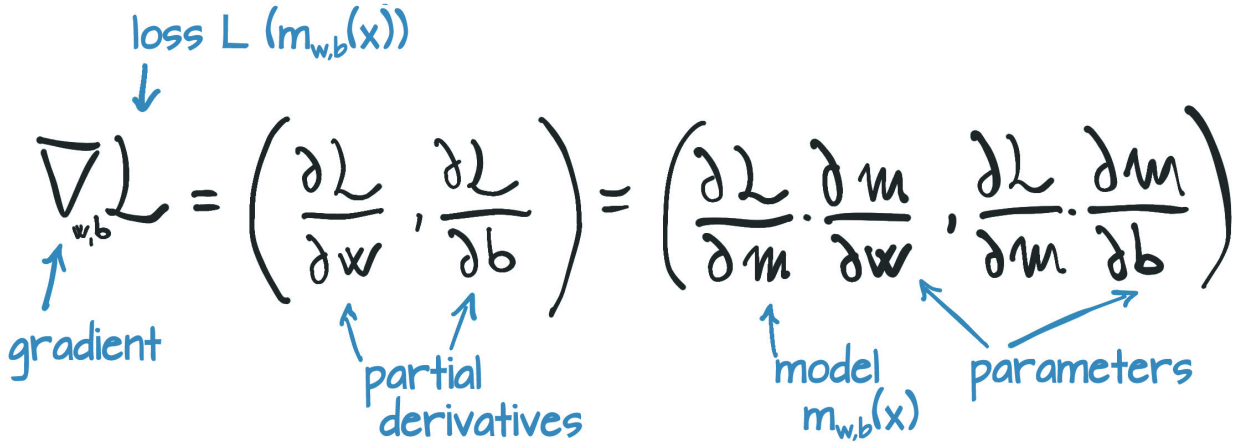


Figure 4.6 **The derivative of the loss function with respect to the weights**

4.1.7 The training loop

Now you have everything in place to optimize your parameters. Starting from a tentative value for a parameter, you can iteratively apply updates to it for a fixed number of iterations or until `w` and `b` stop changing. You can use several stopping criteria, but stick to a fixed number of iterations for now.

While we're at it, we'll introduce you to another piece of terminology. A training iteration during which you update the parameters for all your training samples is called an *epoch*.

The complete training loop looks like this:

```
# In[14]:
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)
        loss = loss_fn(t_p, t_c)
        grad = grad_fn(t_u, t_c, t_p, w, b)

        params = params - learning_rate * grad

        print('Epoch %d, Loss %f' % (epoch, float(loss)))

    return params
```

This is the forward pass.

And this is the backward pass.

This logging line can be verbose.

The actual logging logic used for the output in this text is more complicated (see cell 15 in the same notebook),[6] but the differences are unimportant for understanding the core concepts in this chapter.

Now invoke your training loop:

```
# In[16]:
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[16]:
Epoch 1, Loss 1763.884644
    Params: tensor([-44.1730,  -0.8260])
    Grad:   tensor([4517.2964,   82.6000])
Epoch 2, Loss 5802484.500000
    Params: tensor([2568.4011,   45.1637])
    Grad:   tensor([-261257.4062,   -4598.9707])
Epoch 3, Loss 19408031744.000000
    Params: tensor([-148527.7344,   -2616.3933])
    Grad:   tensor([15109615.0000,   266155.7188])
...
```

[6] https://github.com/deep-learning-with-pytorch/dlwpt-code/blob/master/p1ch5/1_parameter_estimation.ipynb

```
Epoch 10, Loss 9090107547845813096117136197786009 6.000000
    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])
```

Wait—what happened? Your training process blew up, leading to losses becoming `inf`. This result is a clear sign that `params` is receiving updates that are too large; 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. You want to see smaller and smaller updates to `params`, not larger, as shown in figure 4.7.

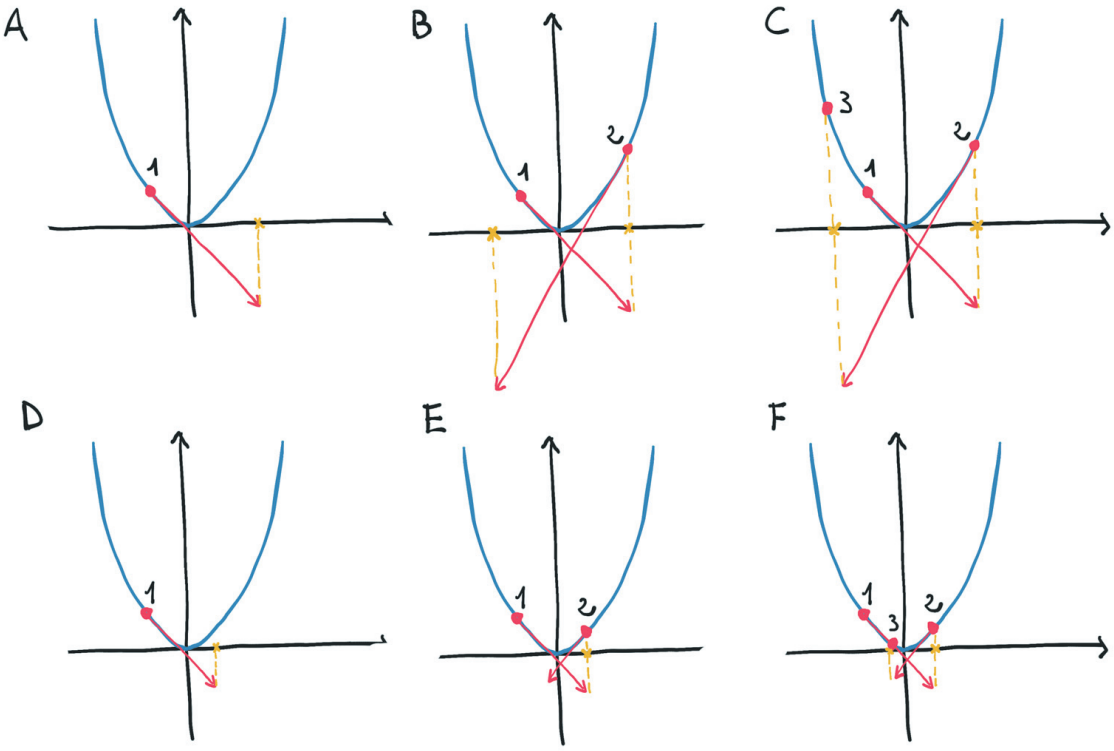


Figure 4.7 Top: Diverging optimization on convex function (parabolalike) due to large steps. Bottom: Converging optimization with small steps.

How can you limit the magnitude of the `learning_rate * grad`? Well, that process looks easy. You could simply choose a smaller `learning_rate`. You usually change learning rates by order of magnitude, so you might try `1e-3` or `1e-4`, which would decrease the magnitude of updates by orders of magnitude. Go with `1e-4` to see how it works out:

```
# In[17]:
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[17]:
Epoch 1, Loss 1763.884644
    Params: tensor([ 0.5483, -0.0083])
    Grad:   tensor([4517.2964,   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([765.4666,  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([0.5780, 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 stable now. But there's another problem: updates to parameters are small, so the loss decreases slowly and eventually stalls. You could obviate this issue by making the `learning_rate` adaptive—that is, change according to the magnitude of updates. You can use several optimization schemes for that purpose; you see one toward the end of this chapter, in section "Optimizers a-la Carte".

Another potential troublemaker exists in the update term: the gradient itself. Go back to look at `grad` at epoch 1 during optimization. You see that the first-epoch gradient for the weight is about 50 times larger than the gradient for the bias, so the weight and bias live in differently scaled spaces. In this case, a learning rate that's large enough to meaningfully update one is so large that it's unstable for the other, or a rate that's appropriate for the second one won't be large enough to change the first meaningfully. You're not going to be able to update your parameters unless you change your formulation of the problem. You could have individual learning rates for each parameter, but for models with many parameters, this approach would be too much to bother with; it's babysitting of the kind you don't like.

You have a simpler way to keep things in check: change the inputs so that the gradients aren't so different. You can make sure that the range of the input doesn't get too far from the range of -1.0 to 1.0, roughly speaking. In this case, you can achieve something close enough to that example by multiplying `t_u` by `0.1`:

```
# In[18]:
t_un = 0.1 * t_u
```

Here, you denote the normalized version of `t_u` by appending `n` to the variable name. At this point, you can run the training loop on your normalized input:

```
# In[19]:
training_loop(
    n_epochs = 100,
    learning_rate = 1e-2,
    params = torch.tensor([1.0, 0.0]),
    t_u = t_un,
    t_c = t_c)
```

You've updated t_u to your new, rescaled t_un.

```
# Out[19]:
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([-0.4445,  2.5165])

tensor([ 2.7553, -2.5162])
```

Even though you set your learning rate back to `1e-2`, parameters didn't blow up during iterative updates. Now take a look at the gradients; they were of similar magnitudes, so using a single `learning_rate` for both parameters worked fine. You probably could do a better job of normalization than rescaling by a factor of ten, but because doing so is good enough for your needs, stick it for now.

> **NOTE** The normalization here helps you get the network trained, but you could make an argument that it's not strictly needed to optimize the parameters for this problem. That's absolutely true! This problem is small enough that you have numerous ways to beat the parameters into submission. For larger, more sophisticated problems, however, normalization is an easy and effective (if not crucial!) tool to use to improve model convergence.

Next, run the loop for enough iterations to see the changes in `params` get small. Change `n_epochs` to `5000`:

```
# In[20]:
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[20]:
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. You saw the loss decrease while you were changing parameters along the direction of gradient descent. The loss didn't go to zero, which could mean that iterations weren't enough to converge to zero or that the data points aren't sitting on a line. As anticipated, your measurements weren't perfectly accurate or noise was involved in the reading.

But look: the value for `w` and `b` looks an awful lot like the numbers you need to use to convert Celsius to Fahrenheit (after accounting for the earlier normalization when you multiplied your inputs by 0.1). The exact values are `w=5.5556` and `b=-17.7778`. Your fancy thermometer was showing temperatures in Fahrenheit the whole time, which is no big discovery, but it proves that your gradient descent optimization process works.

Next, do something that you should have done right from the start: plot your data. We didn't introduce this topic until now for the sake of drama (the surprise effect). But seriously, the first thing that anyone doing data science should do is plot the heck out of the data.

```
# In[21]:
%matplotlib inline
from matplotlib import pyplot as plt

t_p = model(t_un, *params)
```

Remember that you're training on the normalized unknown units.

```
fig = plt.figure(dpi=600)
plt.xlabel("Fahrenheit")
plt.ylabel("Celsius")

plt.plot(t_u.numpy(), t_p.detach().numpy())
plt.plot(t_u.numpy(), t_c.numpy(), 'o')
```

But you're plotting the raw unknown values.

This code produces Figure 4.8.

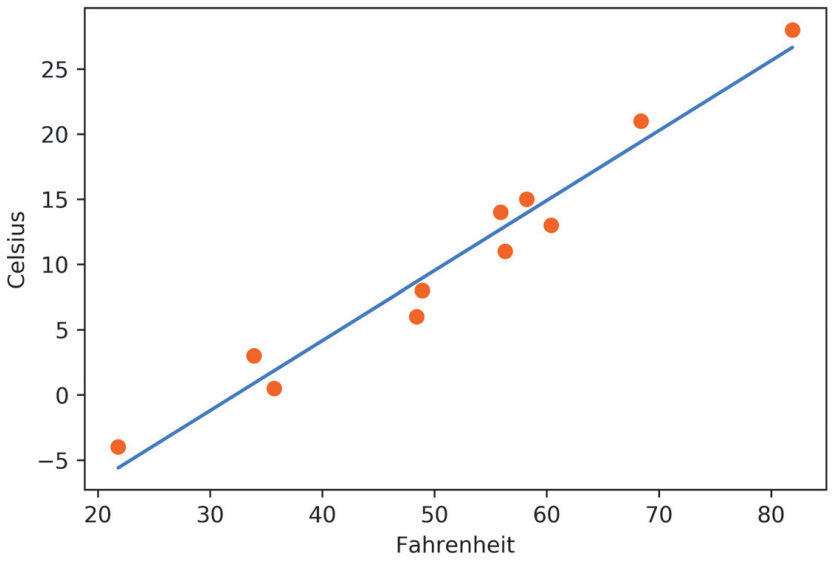


Figure 4.8 The plot of your linear-fit model (solid line) versus input data (circles)

The linear model is a good model for the data, it seems. It also seems that your measurements are somewhat erratic. You should either call your optometrist for a new pair of glasses or think about returning your fancy thermometer.

4.2 PyTorch's autograd: Backpropagate all things

In your little adventure so far, you saw a simple example of backpropagation. You 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 via the *chain rule*. The basic requirement is that all functions you're dealing with are differentiable analytically. In this case, you can compute the gradient (which we called "the rate of change of the loss" earlier) with respect to the parameters in one sweep.

Should you have a complicated model with millions of parameters, as long as the model is differentiable, computing the gradient the loss with respect to parameters amounts to writing the analytical expression for the derivatives and evaluating them once. Granted, writing the analytical expression for the derivatives of a deep composition of linear and nonlinear functions isn't a lot of fun.[7] It isn't particularly quick, either.

[7] Or maybe it is; we won't judge how you spend your weekend!

This situation is where PyTorch tensors come to the rescue, with a PyTorch component called autograd. PyTorch tensors can remember where they come from in terms of the operations and parent tensors that originated them, and they can provide the chain of derivatives of such operations with respect to their inputs automatically. You won't need to derive your model by hand;[8] given a forward expression, no matter how nested, PyTorch provides the gradient of that expression with respect to its input parameters automatically.

At this point, the best way to proceed is to rewrite the thermometer calibration code, this time using autograd, and see what happens. First, recall your model and loss function, as shown in the following listing.

Listing 4.2 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()
```

Again initialize a parameters tensor:

```
# In[5]:
params = torch.tensor([1.0, 0.0], requires_grad=True)
```

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 has `params` as an ancestor has 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 are), the value of the derivative is automatically populated as a `grad` attribute of the `params` tensor.

In general, all PyTorch tensors have an attribute named `grad`, normally `None`:

```
# In[6]:
params.grad is None

# Out[6]:
True
```

All you have to do to populate it is start with a tensor with `requires_grad` set to `True`, call the model, compute the loss, and then call `backward` on the `loss` tensor:

```
# In[7]:
loss = loss_fn(model(t_u, *params), t_c)
loss.backward()
```

[8] Bummer! What are we going to do on Saturdays now?

```
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` (figure 4.9).

You could have any number of tensors with `requires_grad` set to `True` and any composition of functions. In this case, PyTorch would compute 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 one thing that PyTorch newcomers (and a lot of more experienced folks) trip up on regularly. We wrote *accumulate*, not *store*.

WARNING Calling `backward` leads derivatives to *accumulate* at leaf nodes. You need to *zero the gradient explicitly* after using it for parameter updates.

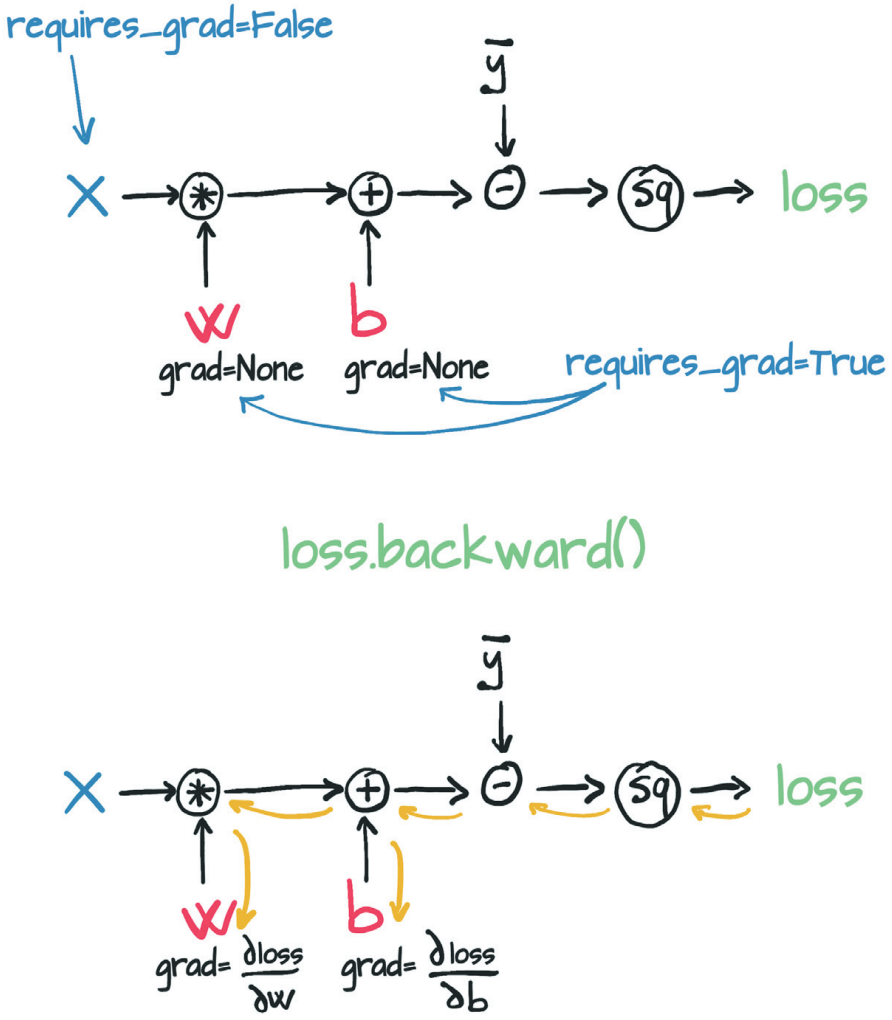


Figure 4.9 The forward graph and backward graph of model as computed with autograd

To repeat, calling `backward` leads derivatives to *accumulate* at leaf nodes. So if `backward` has been called earlier, the loss is evaluated again, and `backward` is called again (as in any training loop), the gradient at each leaf is accumulated (summed) on top of the one computed at the preceding iteration, which leads to an incorrect value for the gradient.

To prevent this situation from occurring, you need to **zero the gradient explicitly** at each iteration. You can do so easily by using the in-place `zero_` method:

```
# In[8]:
if params.grad is not None:
    params.grad.zero_()
```

NOTE You may be curious why zeroing the gradient is a required step instead of automatic whenever you call `backward`. The reason is to provide more flexibility and control for working with gradients in complicated models.

Having this reminder drilled into your head, now see how your autograd-enabled training code looks like, start to end:

```
# 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:
            params.grad.zero_()

        t_p = model(t_u, *params)
        loss = loss_fn(t_p, t_c)
        loss.backward()

        params = (params - learning_rate *
     params.grad).detach().requires_grad_()

        if epoch % 500 == 0:
            print('Epoch %d, Loss %f' % (epoch, float(loss)))

    return params
```

This could be done at any point in the loop prior to calling `loss.backward()`

It's somewhat cumbersome, but as you'll see in "Optimizers a-la Carte," it's not an issue in practice.

Notice that when you updated `params`, you also did an odd `.detach().requires_grad_()` dance. To understand why, think about the computation graph that you've built. Reformulate your `params` update line a little so that you're not reusing variable names: `p1 = (p0 * lr * p0.grad)` Here, `p0` is the random weights with which you initialized the model. `p0.grad` is computed from a combination of `p0` and your training data via the loss function.

So far, so good. Now you need to look at the second iteration of the loop: `p2 = (p1 * lr * p1.grad)`. As you've seen, the computation graph for `p1` goes back to `p0`, which is problematic because (a) you have to keep `p0` in memory (until you're done with training), and (b) it confuses the matter of where you should be assigning error via backpropagation.

Instead, detach the new `params` tensor from the computation graph associated with its update expression by calling `.detatch()`. This way, `params` effectively loses the memory of the operations that generated it. Then you can reenable tracking by calling `.requires_grad_()`, an `in_place` operation (see the trailing `_`) that reactivates autograd for the tensor. Now you can release the memory held by old versions of `params` and need to backpropagate through only your current weights.

See whether this code works:

```
# In[10]:
training_loop(
    n_epochs = 5000,
    learning_rate = 1e-2,
    params = torch.tensor([1.0, 0.0], requires_grad=True),
    t_u = t_un,
    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)
```

Note that again, you're using the normalized t_un instead of t_u.

Adding this requires_grad=True is key.

You get the same result that you got previously. Good for you! Although you're *capable* of computing derivatives by hand, you no longer need to.

4.2.1 Optimizers a la carte

This code uses vanilla gradient descent for optimization, which works fine for this simple case. Needless to say, several optimization strategies and tricks can help convergence, especially when models get complicated.

Now is the right time to introduce the way that PyTorch abstracts the optimization strategy away from user code, such as the training loop, sparing you from the boilerplate busywork of having to update every parameter in your model yourself. The `torch` module has an `optim` submodule where you can find classes that implement different optimization algorithms. Here's an abridged listing:

Listing 4.3 code/p1ch5/3_optimizers.ipynb

```
# In[5]:
import torch.optim as optim

dir(optim)

# Out[5]:
```

```
['ASGD',
 'Adadelta',
 'Adagrad',
 'Adam',
 'Adamax',
 'LBFGS',
 'Optimizer',
 '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 that the optimizer can update their values and access their `grad` attribute, as represented in figure 4.10.

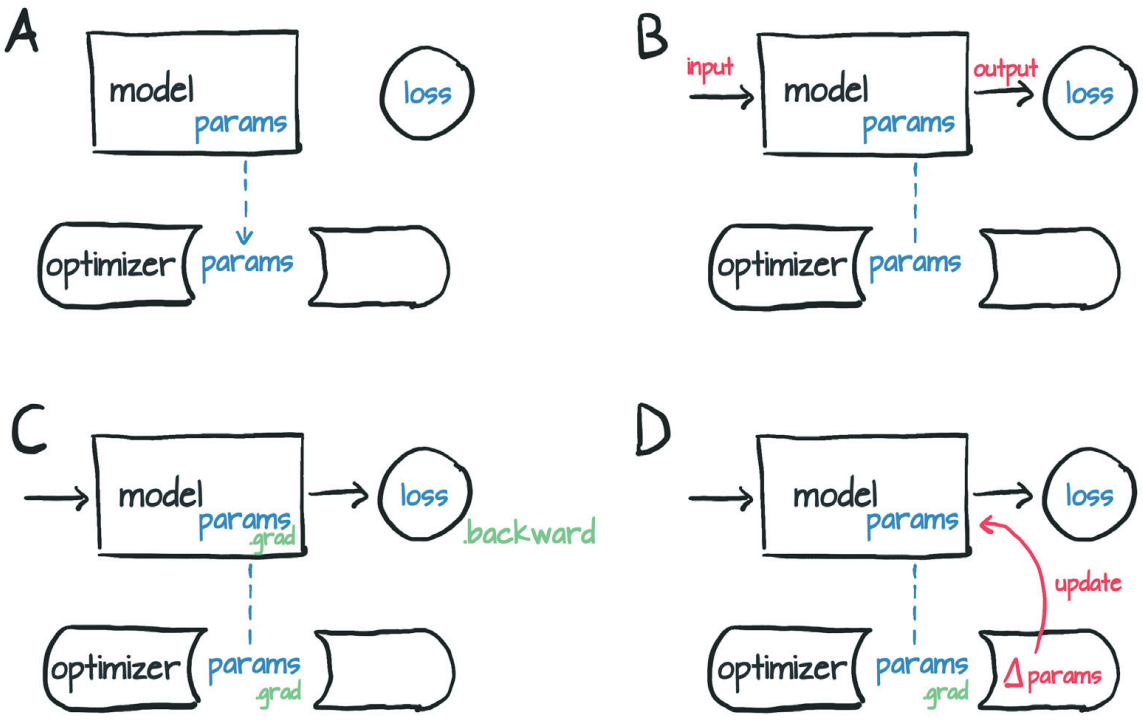


Figure 4.10 Conceptual representation of how an optimizer holds a reference to parameters (A), and after a loss is computed from inputs (B), a call to `.backward` leads to `.grad` being populated on parameter (C). At that point, the optimizer can access `.grad` and compute the parameter updates (D).

Each optimizer exposes two methods: `zero_grad` and `step`. The former zeros the `grad` attribute of all the parameters passed to the optimizer upon construction. The latter updates the value of those parameters according to the optimization strategy implemented by the specific optimizer.

Now 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*. The optimizer itself is 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*. The optimizer itself, however, doesn't know whether the loss was evaluated on all the samples (vanilla) or a random subset thereof (stochastic), so the algorithm is the same in the two cases.

Anyway, take your 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` was updated when `step` was called, and you didn't have to touch it yourself! What happened was that the optimizer looked into `params.grad` and updated `params` by subtracting `learning_rate` times `grad` from it, exactly as in your former hand-rolled code.

Are you ready to stick this code in a training loop? Nope! The big gotcha almost got you: you forgot to zero out the gradients. Had you called the preceding code in a loop, gradients would have accumulated in the leaves at every call to `backward`, and your gradient descent would have been all over the place!

Here's the loop-ready code, with the extra `zero_grad` in the right spot (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()
loss.backward()
optimizer.step()

params

# Out[8]:
tensor([1.7761, 0.1064], requires_grad=True)
```

As before, the placement of this call is somewhat arbitrary. It could be earlier in the loop as well.

Perfect! See how the `optim` module helped you abstract away the specific optimization scheme? All you have to do is provide a list of `params` to it (that list can be extremely long, as needed for deep neural network models) and then forget about the details.

Update your training loop accordingly:

```
# In[9]:
def training_loop(n_epochs, optimizer, params, t_u, t_c):
    for epoch in range(1, n_epochs + 1):
        t_p = model(t_u, *params)
        loss = loss_fn(t_p, t_c)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if epoch % 500 == 0:
            print('Epoch %d, Loss %f' % (epoch, float(loss)))

    return params

# In[10]:
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 = 5000,
    optimizer = optimizer,
    params = params,
    t_u = t_un,
    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)
```

It's important that both params here are the same object; otherwise, the optimizer won't know what parameters the model used.

Again, you get the same result as before. Great. You have further confirmation that you know how to descend a gradient by hand! To test more optimizers, all you have to do is instantiate a different optimizer, such as `Adam`, instead of `SGD`. The rest of the code stays as is. This stuff is pretty handy.

We won't go into much detail on `Adam`, but it suffices to say that it's a more sophisticated optimizer in which the learning rate is set adaptively. In addition, it's a lot less sensitive to the scaling of the parameters—so insensitive that you can go back to use

the original (non-normalized) input `t_u` and even increase the learning rate to `1e-1`. `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)

training_loop(
    n_epochs = 2000,
    optimizer = optimizer,
    params = params,
    t_u = t_u,
    t_c = t_c)
```

New optimizer class here.

Note that you're back to the original t_u as input.

```
# Out[11]:
Epoch 500, Loss 7.612901
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 isn't the only flexible part of your training loop. Turn your attention to the model. To train a neural network on the same data and the same loss, all you'd need to change is the `model` function. Doing this wouldn't make sense in this case, because you know that converting Celsius to Fahrenheit amounts to a linear transformation. Neural networks allow you to remove your arbitrary assumptions about the shape of the function you should be approximating. Even so, 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).

We've touched on a lot of the essential concepts that will enable you 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 by using gradient descent or other optimizers. We don't have a whole lot more to cover. The rest is mostly filling in the blanks, however extensive they are.

Next, we discuss how to split up samples, which sets up a perfect use case for learning to control autograd better.

4.2.2 Training, validation, and overfitting

Johannes Kepler kept a part of the data on the side so that he could validate his models on independent observations—a vital thing to do, especially when the model you adopt could potentially approximate functions of any shape, as in the case of neural networks. In other words, a highly adaptable model tends to use its many parameters to make sure that the loss is minimal *at* the data points, but you'll have no guarantee that the model behaves well *away* from or *between* the data points. After all, that's all you're asking the optimizer to do: minimize the loss *at* the data points. Sure enough, if you had independent data points that you didn't use to evaluate your loss or

descend along its negative gradient, you'd soon find out that evaluating the loss at those independent data points would yield a higher-than-expected loss. We've already mentioned this phenomenon, called *overfitting*.

The first action you can take to combat overfitting is to recognize that it might happen. To do so, as Kepler figured out in 1600, you must take a few data points out of your data set (the *validation set*) and fit our model to the remaining data points (the *training set*), as shown in figure 4.11. Then, while you're fitting the model, you can evaluate the loss once on the training set and once on the validation set. When you're trying to decide whether you've done a good job of fitting your model to the data, you must look at each data set!

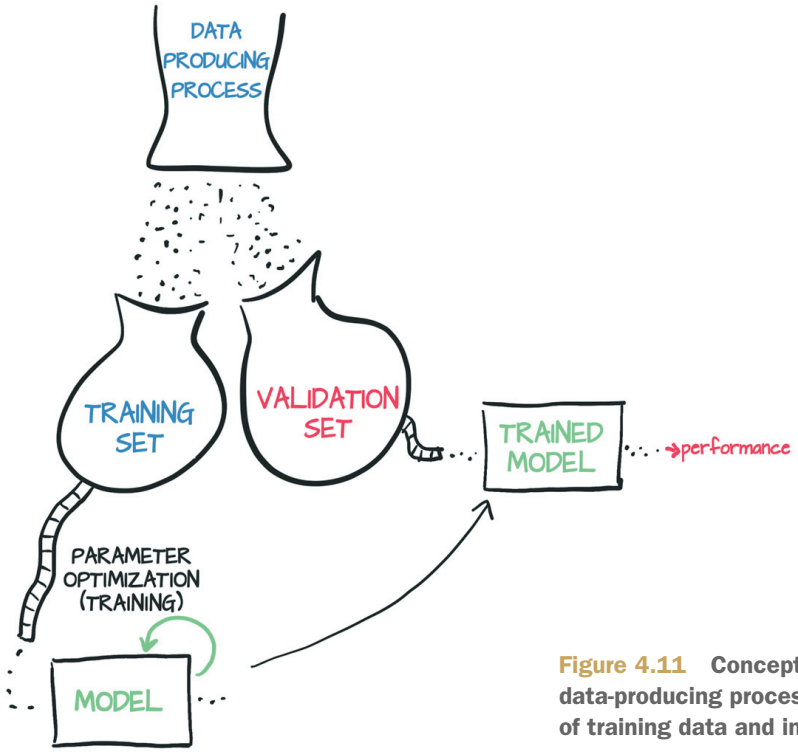


Figure 4.11 Conceptual representation of a data-producing process and the collection and use of training data and independent validation data

The first figure, the training loss, tells you whether your model can fit the training set at all—in other words, whether your model has enough *capacity* to process the relevant information in the data. If your mysterious thermometer somehow managed to measure temperatures by using a logarithmic scale, your poor linear model wouldn't have had a chance to fit those measurements and provide a sensible conversion to Celsius. In that case, your training loss (the loss you 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 parameters, the simpler the shape of the function your network will be able to approximate. So here's rule one: if the training loss isn't decreasing, chances are

that the model is too simple for the data. The other possibility is that your data doesn't contain meaningful information for it to explain the output. If the nice folks at the shop sold you a barometer instead of a thermometer, you'd have little chance to predict temperature in Celsius from pressure alone, even if you used the latest neural network architecture from Quebec.[9]

What about the validation set? Well, if the loss evaluated in the validation set doesn't decrease along with the training set, your model is improving its fit of the samples it's seeing during training, but it isn't *generalizing* to samples outside this precise set. As soon as you evaluate the model at new, previously unseen points, the values of the loss function are poor. Here's rule two: if the training loss and the validation loss diverge, you're overfitting.

We'll delve into this phenomenon a little here, going back to the thermometer example. You could have decided to fit the data with a more complicated function, such as a piecewise polynomial or a large neural network. This function could generate a model that meanders its way through the data points, as in figure 4.12, because it pushes the loss close to zero. Because the behavior of the function away from the data points doesn't increase the loss, there's nothing to keep the model away from the training data points in check.

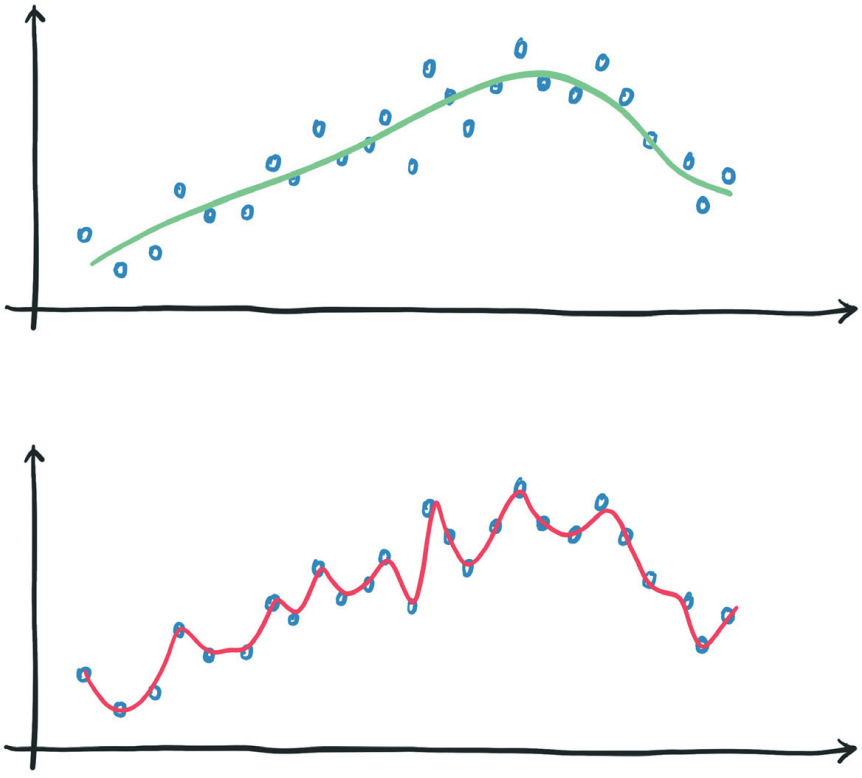


Figure 4.12 **Rather extreme example of overfitting**

[9] https://www.umontreal.ca/en/artificialintelligence

What's the cure, though? Good question. Overfitting looks like a problem of making sure that the behavior of the model *in between* data points is sensible for the process you're trying approximate. First, you should make sure that you get enough data for the process. If you collected data from a sinusoidal process by sampling it regularly at a low frequency, you'd have a hard time fitting a model to it.

Assuming that you have enough data points, you should make sure that the model that's capable of fitting the training data is as regular as possible between the data points. You have several ways to achieve this goal. One way is to add so-called *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 way is to add noise to the input samples, to artificially create new data points between training data samples and force the model to try to fit them too. Several other ways are somewhat related to the preceding ones. But the best favor you can do for yourself, at least as a first move, is to make your model simpler. From an intuitive standpoint, a simpler model may not fit the training data as perfectly as a more complicated model would do, but it will likely behave more regularly between data points.

You've got some nice tradeoffs here. On one hand, you need to model to have enough capacity for it to fit the training set. On the other hand, you need the model to avoid overfitting. Therefore, the process for choosing the right size of a neural network model, in terms of parameters, is based on two steps: increase the size until it fits and then scale it down until it stops overfitting.

Your life will be a balancing act between fitting and overfitting. You can split the data into a training set and a validation set by shuffling `t_u` and `t_c` in the same way and then splitting the resulting shuffled tensors into two parts.

Shuffling the elements of a tensor amounts to finding a permutation of its indices. The `randperm` function does 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

# Out[12]:
(tensor([ 8,  0,  3,  6,  4,  1,  2,  5, 10]), tensor([9, 7]))
```

Because these values are random, don't be surprised if your values end up being different from here on.

You get index tensors that you 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
```

Your training loop doesn't change. You want to evaluate the validation loss at every epoch to have a chance to recognize whether you'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)
        train_loss = loss_fn(train_t_p, train_t_c)

        val_t_p = model(val_t_u, *params)
        val_loss = loss_fn(val_t_p, val_t_c)

        optimizer.zero_grad()
        train_loss.backward()
        optimizer.step()

        if epoch <= 3 or epoch % 500 == 0:
            print('Epoch {}, Training loss {}, Validation loss {}'.format(
                epoch, float(train_loss), float(val_loss)))

    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,
    val_t_u = val_t_un,
    train_t_c = train_t_c,
    val_t_c = val_t_c)

# Out[15]:
Epoch 1, Training loss 88.59708404541016, Validation loss 43.31699752807617
Epoch 2, Training loss 34.42190933227539, Validation loss 35.03486633300781
Epoch 3, Training loss 27.57990264892578, Validation loss 40.214229583740234
Epoch 500, Training loss 9.516923904418945, Validation loss 9.02982234954834
Epoch 1000, Training loss 4.543173789978027, Validation loss
     2.596876621246338
Epoch 1500, Training loss 3.1108808517456055, Validation loss
     2.9066450595855713
Epoch 2000, Training loss 2.6984243392944336, Validation loss
     4.1561737060546875
Epoch 2500, Training loss 2.579646348953247, Validation loss
     5.138668537139893
Epoch 3000, Training loss 2.5454416275024414, Validation loss
     5.755766868591309

tensor([  5.6473, -18.7334], requires_grad=True)
```

These two pairs of lines are the same except for the `train_*` vs. `val_*` inputs.

Note that you have no `val_loss.backward()` here because you don't want to train the model on the validation data.

Because you're using SGD again, you're back to using normalized inputs.

Here, we're not being entirely fair to the model. The validation set is small, so the validation loss will be meaningful only up to a point. In any case, note that the validation loss is higher than your training loss, although not by an order of magnitude. The fact that a model performs better on the training set is expected since the model parameters are being shaped by the training set. Your main goal is to also see both the training loss and the validation loss decreasing. Although ideally, both losses would be roughly the same value, as long as validation loss stays reasonably close to the training loss, you know that your model is continuing to learn generalized things about your data. In figure 4.13, case C is ideal, and D is acceptable. In case A, the model isn't learning at all, and in case B, you see overfitting.

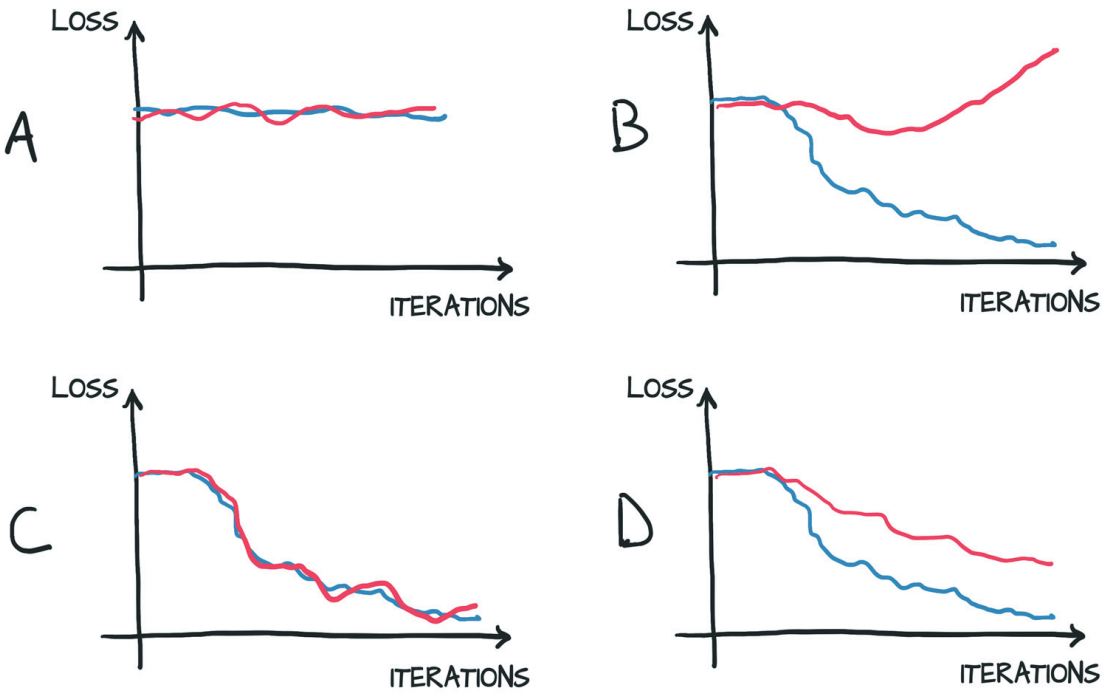


Figure 4.13 Overfitting scenarios for the training (blue) and validation (red) losses. (A) Training and validation losses don't decrease; the model isn't learning due to no information in the data or insufficient capacity of the model. (B) Training loss decreases while validation increases (overfitting). (C) Training and validation losses decrease in tandem; performance may be improved further, as the model isn't at the limit of overfitting. (D) Training and validation losses have different absolute values but similar trends; overfitting is under control.

4.2.3 Nits in autograd and switching it off

From the training loop, you can appreciate that you only ever call `backward` on the `train_loss`. Therefore, errors 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 then on `val_t_u`—after which `backward` is

called. Won't this confuse the hell out of autograd? Won't `backward` be influenced by the values generated during the pass on the validation set?

Luckily, 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`, creating 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 is 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 4.14.

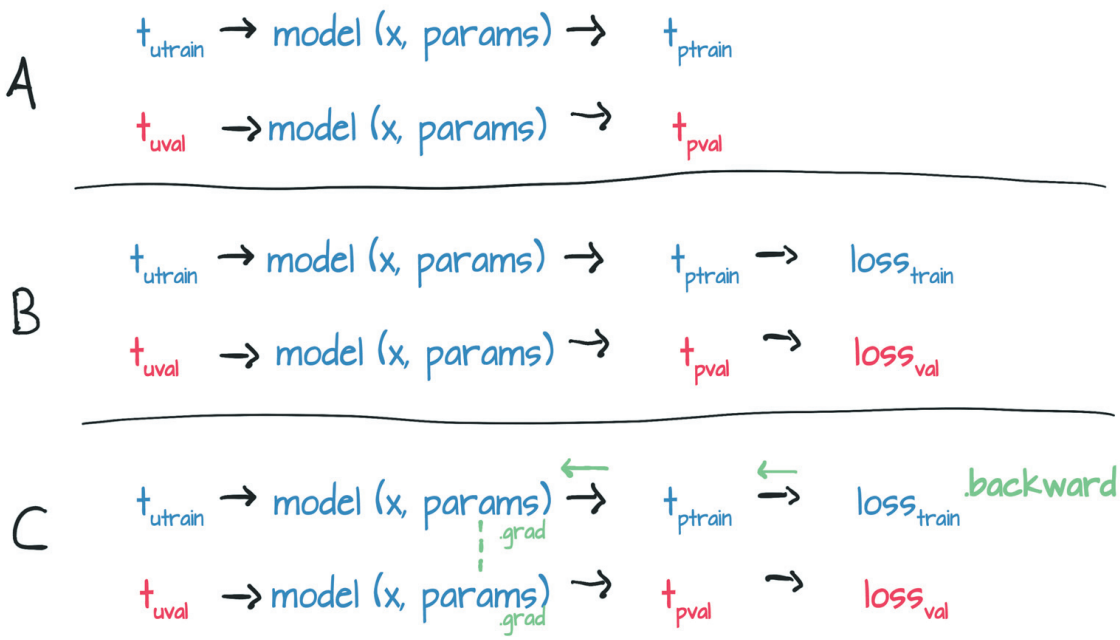


Figure 4.14 Diagram showing how gradients propagate through a graph with two losses when `.backward` is called on one of them

The only tensors that these two graphs have in common are the parameters. When you call `backward` on `train_loss`, you run the `backward` on the first graph. In other words, you accumulate the derivatives of the `train_loss` with respect to the parameters based on the computation generated from `train_t_u`.

If you (incorrectly) called `backward` on `val_loss` as well, you'd have accumulated the derivatives of the `val_loss` with respect to the parameters *on the same leaf nodes*. Remember the `zero_grad` thing, whereby gradients would be accumulated on top of each other every time you called `backward` unless you zeroed out gradients explicitly? Well, here something 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, you'd effectively train your model on the whole data set (both training and validation), because the gradient would depend on both. Pretty interesting.

Here's another element for discussion: because you're never calling `backward` on `val_loss`, why are you building the graph in the first place? You could in fact call `model` and `loss_fn` as plain functions without tracking history. However optimized, tracking history comes with additional costs that you could forgo during the validation pass, especially when the model has millions of parameters.

To address this situation, PyTorch allows you to switch off autograd when you don't need it by using the `torch.no_grad` context manager. You won't see any meaningful advantage in terms of speed or memory consumption on your small problem. But for larger models, the differences can add up. You can make sure that this context manager 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():
            val_t_p = model(val_t_u, *params)
            val_loss = loss_fn(val_t_p, val_t_c)
            assert val_loss.requires_grad == False

        optimizer.zero_grad()
        train_loss.backward()
        optimizer.step()
```

Context manager here.

All requires_grad args are forced to False inside this block.

Using the related `set_grad_enabled` context, you can also condition code to run with `autograd` enabled or disabled, according to a Boolean expression, typically indicating whether you're running in training or inference. You could define a `calc_forward` function that takes data in 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
```

Conclusion

This chapter started with a big question: how can a machine learn from examples? We spent the rest of the chapter describing the mechanism by which a model can be optimized to fit data. We chose to stick with a simple model to show all the moving parts without unneeded complications.

Exercises

- Redefine the model to be `w2 * t_u ** 2 + w1 * t_u + b`.
  - What parts of the training loop and so on must be changed to accommodate this redefinition?
    - What parts are agnostic to swapping out the model?
  - Is the resulting loss higher or lower after training?
    - Is the result better or worse?

Summary

- Linear models are the simplest reasonable model to use to fit data.
- Convex optimization techniques can be used for linear models, but they don't generalize to neural networks, so this chapter focuses on parameter estimation.
- Deep learning can be used for generic models that aren't engineered to solve a specific task but can be adapted automatically to specialize in the problem at hand.
- Learning algorithms amount to optimizing parameters of models based on observations. 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 loss function as low as possible.
- The rate of change of the loss function with respect to 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 contributed to the final output. This feature allows users to rely on the dynamic computation graph during complex forward passes.
- Context managers such as `with torch.no_grad():` can be used to control autograd behavior.
- Data is often split into separate sets of training samples and validation samples, allowing a model to be evaluated on data it wasn't 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 situation usually occurs when the model doesn't generalize and instead memorizes the desired outputs for the training set.

5 Using a neural network to fit your data

This chapter covers

- The use of nonlinear activation functions as the key difference from linear models
- The many kinds of activation functions in common use
- PyTorch's `nn` module, containing neural network building blocks
- Solving a simple linear-fit problem with a neural network

You've taken a close look at how a linear model can learn and how to make it happen in PyTorch, focusing on a simple regression problem that required a linear model with one input and one output. This simple example allowed you to dissect the mechanics of a model that learns without getting overly distracted by the implementation of the model itself. Backpropagating error to the parameters and then updating those parameters by taking the gradient with respect to the loss is going to be the same no matter what the underlying model is (figure 5.1).

In this chapter, you're going to make changes in your model architecture. You're going to implement a full artificial neural network to solve your problem.

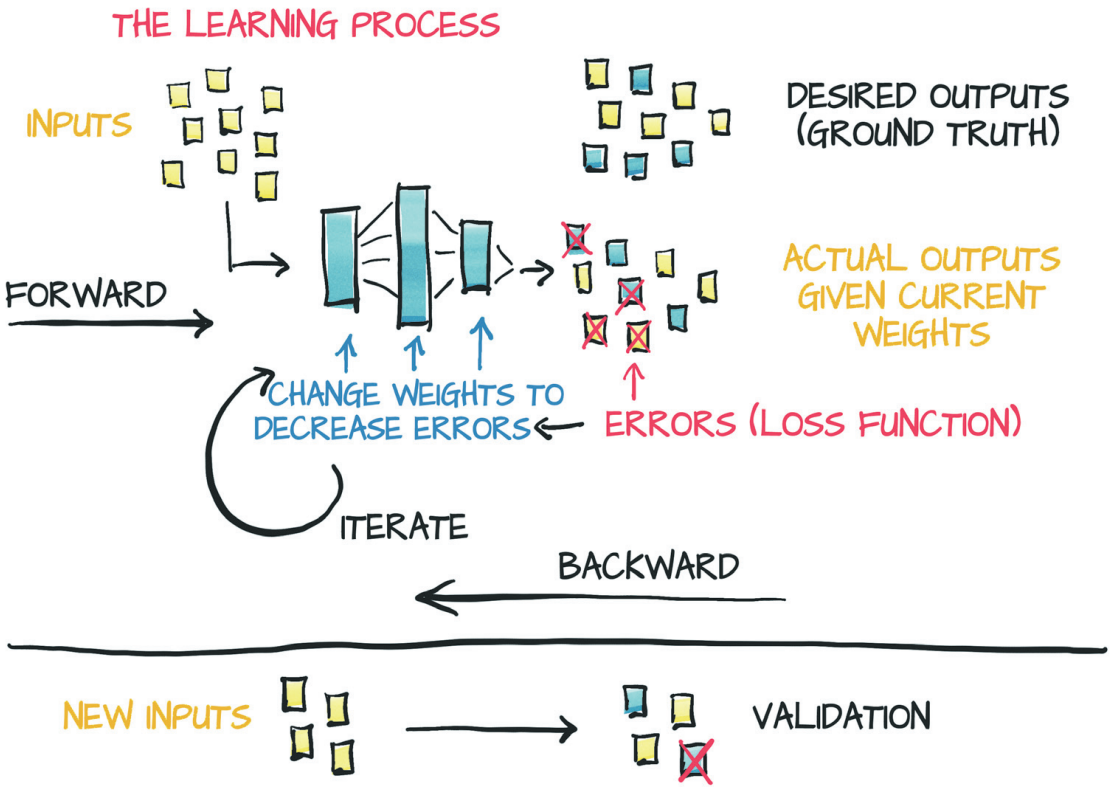


Figure 5.1 Mental model of the learning process

Your thermometer conversion training loop and Fahrenheit-to-Celsius samples split into training and validation sets remain. You could start to use a quadratic model, rewriting your `model` as a quadratic function of its input (such as `y = a * x**2 + b * x + c`). Because 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 you, though, because you'd still be fixing the shape of the function.

This chapter is where you start to hook the foundational work you've put in with the PyTorch features you'll be using day in and day out as you work on your projects. You'll have an understanding of what's going on underneath the porcelain of the PyTorch API rather than thinking that it's so much black magic.

Before we get into the implementation of the new model, though, we'll explain what we mean by *artificial neural network*.

5.1 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* obviously suggests a link to the way the human brain works. As a matter of fact, although the initial models were inspired by neuroscience,[1] modern

[1] http://psycnet.apa.org/doiLanding?doi=10.1037%2Fh0042519

artificial neural networks bear only a vague resemblance to the mechanisms of neurons in the brain. It seems likely that artificial and physiological neural networks use vaguely similar mathematical strategies for approximating complicated functions because that family of strategies works effectively.

NOTE We're going to drop *artificial* and refer to these constructs as *neural networks* from here on.

The basic building block of these complicated functions is the *neuron*, pictured in figure 5.2. At its core, a neuron is nothing but a linear transformation of the input (such as multiplication of the input by a number [the *weight*] and the addition of a constant [the *bias*]) followed by the application of a fixed nonlinear function (referred to as the *activation function*).

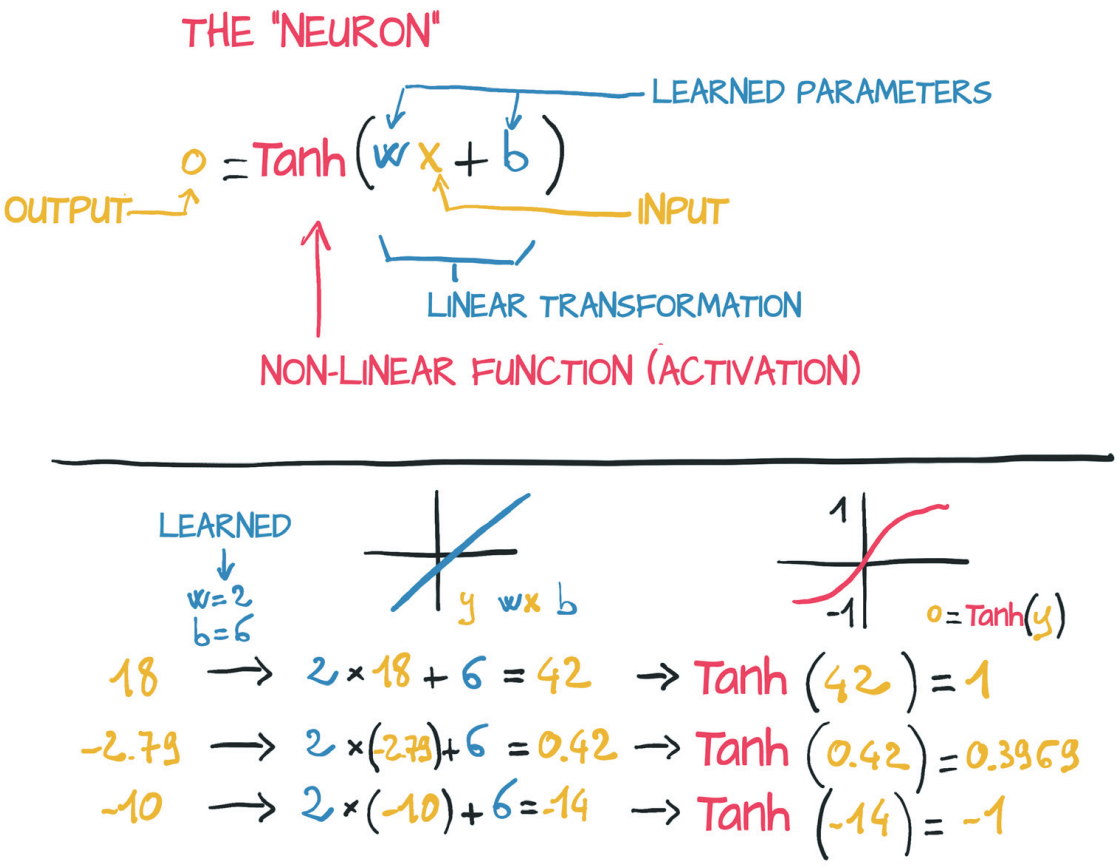


Figure 5.2 An artificial neuron: a linear transformation enclosed in a nonlinear function

Mathematically, you can write this as `o = f(w * x + b)`, with `x` as the input, `w` as the weight or scaling factor, and `b` as the bias or offset. `f` is the activation function, set to the hyperbolic tangent or `"tanh"` function here. In general, `x` and hence `o` can be simple scalars, or vector-valued (holding many scalar values). Similarly, `w` can be a single scalar, or a matrix, whereas `b` is a scalar or vector (the dimensionality of the inputs and weights must match, however). In the latter case, the expression is referred to as a

layer of neurons because it represents many neurons via the multidimensional weights and biases.

A multilayer neural network, as represented in figure 5.3, is a composition of the preceding functions:

```
x_1 = f(w_0 * x + b_0)
x_2 = f(w_1 * x_1 + b_1)
...
y = f(w_n * x_n + b_n)
```

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 here allows `w_0` to hold an entire *layer* of neurons, not just a single weight.

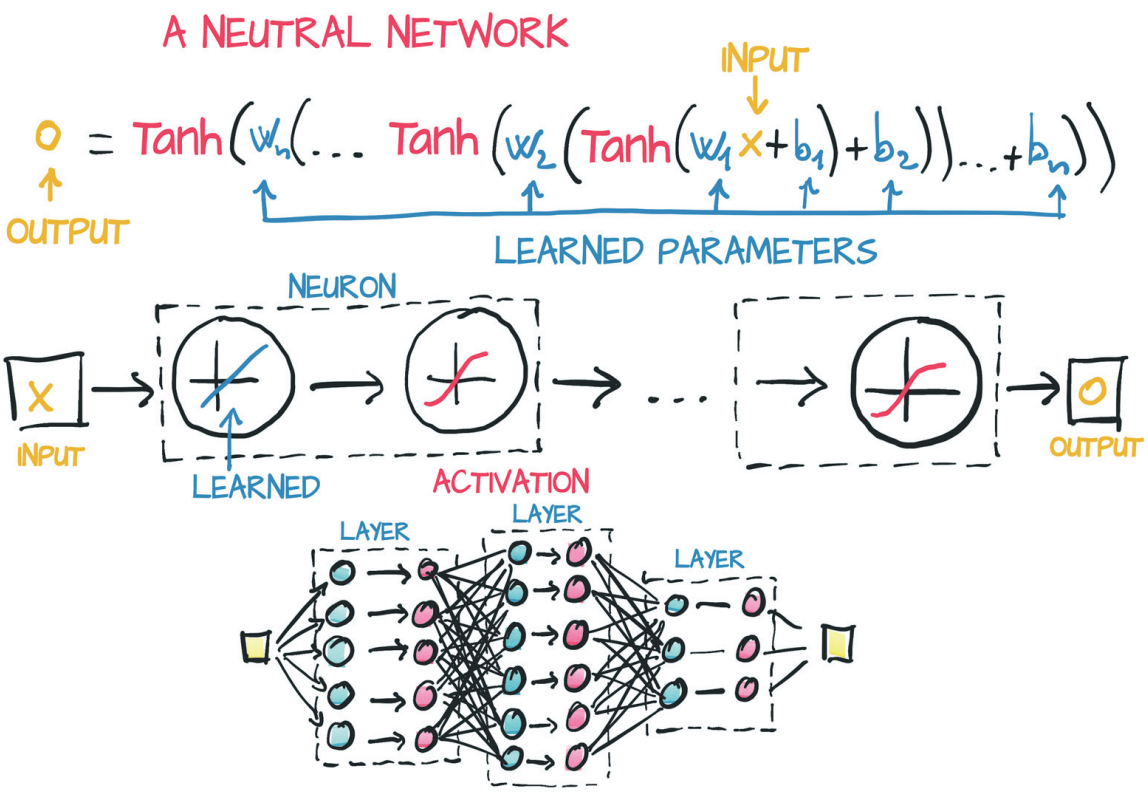


Figure 5.3 A neural network with three layers

An important difference between the earlier linear model and what you'll be using for deep learning is the shape of the error function. The linear model and error-squared loss function had a convex error curve with a singular, clearly defined minimum. If you were to use other methods, you could solve for it automatically and definitively. Your parameter updates were attempting to *estimate* that singular correct answer as best they could.

Neural networks don't 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 that you're attempting to approximate. Instead, you're trying to get all the parameters, when acting *in concert*, to produce useful output. Since that useful

output is only going to approximate the truth, there will be some level of imperfection. Where and how those imperfections manifest is somewhat arbitrary, and by implication the parameters that control the output (and hence the imperfections) are somewhat arbitrary as well. This output results in neural network training's looking much like parameter estimation from a mechanical perspective, but you must remember that the theoretical underpinnings are quite different.

A big part of the reason why neural networks have nonconvex error surfaces is due to the activation function. The ability of an ensemble of neurons to approximate a wide range of useful functions depends on the combination of the linear and nonlinear behavior inherent to each neuron.

5.1.1 All you need is activation

The simplest unit in (deep) neural networks is a linear operation (scaling + offset) followed by an activation function. You had a linear operation in your latest model; the linear operation *was* the entire model. The activation function has the role of concentrating the outputs of the preceding linear operation into a given range.

Suppose that you're assigning a "good doggo" score to images. Pictures of retrievers and spaniels should have a high score; images of airplanes and garbage trucks should have a low score. Bear pictures should have a low-ish score too, though higher than garbage trucks.

The problem is that you have to define a high score. Because you've got the entire range of `float32` to work with, you can go pretty high. Even if you say "It's a ten point scale," sometimes your model is going to produce a score of 11 out of 10. Remember that under the hood, it's all sum of `w*x+b` matrix multiplications, which won't naturally limit themselves to a specific range of outputs.

What you want to do is firmly constrain the output of your linear operation to a specific range so that the consumer of this output isn't having to handle numerical inputs of puppies at 12/10, bears at -10, and garbage trucks at -1000.

One possibility is to cap the output values. Anything below zero is set to zero, and anything above 10 is set to 10. You use a simple activation function called `torch.nn.Hardtanh`.[2]

Another family of functions that works well is `torch.nn.Sigmoid`, which is `1 / (1 + e ** -x)`, `torch.tanh`, and others that you'll see in a moment. These functions have a curve that asymptotically approaches zero or negative one as `x` goes to negative infinity, approaches one as `x` increases, and has a mostly constant slope at `x == 0`. Conceptually, functions shaped this way work well, because it means that your neuron (which, again, is a linear function followed by an activation) will be sensitive to an area in the middle of your linear function's output; everything else gets lumped up next to the boundary values. As you see in figure 5.4, a garbage truck gets a score of `-0.97`, whereas bears, foxes, and wolves may end up somewhere in the `-0.3` to `0.3` range.

[2] See https://pytorch.org/docs/stable/nn.html#hardtanh, but note that the default range is -1 to +1.

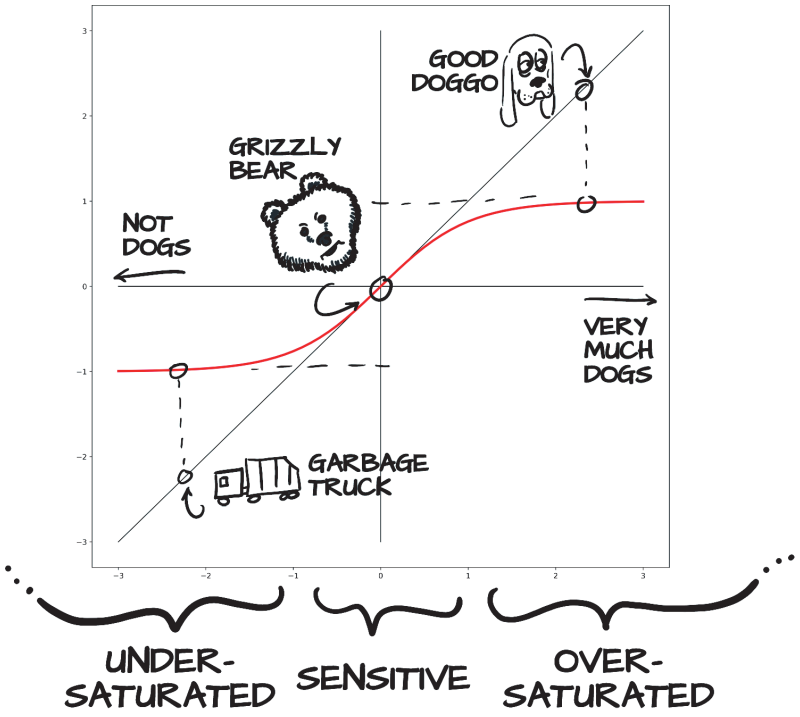


Figure 5.4 **Dogs, bears, and garbage trucks being mapped to "how doglike" via the tanh activation function**

Garbage trucks are flagged as "not dogs," the good dog maps to "clearly a dog," and the bear ends up somewhere in the middle. In code, you see the exact values:

```
>>> import math
>>> math.tanh(-2.2)
-0.9757431300314515
>>> math.tanh(0.1)
0.09966799462495582
>>> math.tanh(2.5)
0.9866142981514303
```

Garbage truck

Bear

Good doggo

With the bear in the sensitive range, small changes to the bear result in a noticeable change in the result. You could swap from a grizzly to a polar bear (which has a vaguely more traditionally canine face) and see a jump up the Y axis as you slide toward the "very much a dog" end of the graph. Conversely, a koala bear would register as less doglike and would see a drop in the activated output. There isn't much you could do to the garbage truck to make it register as doglike, though. Even with drastic changes, you might see a shift only from -0.97 to -0.8 or so.

Quite a few activation functions are available, some of which are pictured in figure 5.5. In the first column, you see the continuous functions `Tanh` and `Softplus`; the second column has "hard" versions of the activation functions to their left, `Hardtanh` and `ReLU`. `ReLU` (Rectified Linear Unit) deserves special note, as it is currently considered to be one of the best-performing general activation functions, as many state-of-the-art results have used it. The `Sigmoid` activation function, also known as the logistic func-

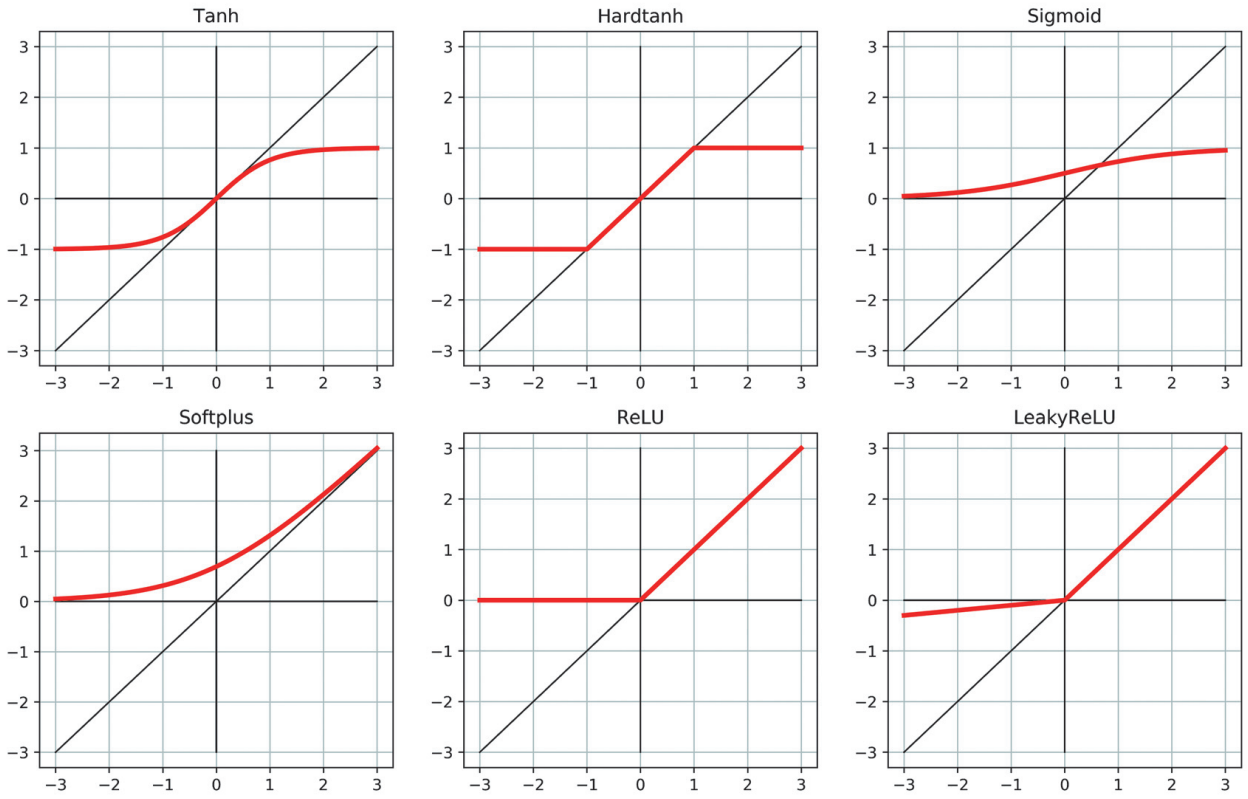


Figure 5.5 **A collection of common and not-so-common activation functions**

tion, was widely used in early deep learning work but has fallen out of common use. 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.)

Activation functions are curious, because with such a wide variety of proven-successful ones (many more than pictured in figure 5.5), 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 disproved in the specific. That said, by definition, activation functions[3]

- *Are nonlinear*—Repeated applications of `w*x+b` without an activation function results in a polynomial. The nonlinearity allows the overall network to approximate more complex functions.
- *Are differentiable*—They're differentiable so that gradients can be computed through them. Point discontinuities, as you see in `Hardtanh` or `ReLU`, are fine.

Without those functions, the network either falls back to being a complicated polynomial or becomes difficult to train.

[3] Even these statements aren't always true, of course: See https://openai.com/blog/nonlinear-computation-in-linear-networks.

Also generally true (though less so), the functions

- Have at least one sensitive range, where no-trivial changes to the input result in a corresponding nontrivial change in the output.
- Have at least one insensitive (or saturated) range, where changes to the input result in little to no change in the output.

By way of example, the `Hardtanh` function could easily be used to make piecewise-linear approximations of a function due to combining the sensitive range with different weights and biases on the input.

Often (but far from universally so), the activation function has at least one of the following:

- 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 about what you know about how backpropagation works, you can figure out that the errors will propagate backward through the activation more effectively when the inputs are in the response range, whereas errors won't greatly affect neurons for which the input is saturated (because the gradient will be close to zero due to the flat area around the output).

All put together, this mechanism is pretty powerful. What we're saying is that in a network built out of linear + activation units, when different inputs are presented to the network, (a) different units 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 one 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.

You're starting to get a deeper intuition about how joining many linear + activation units in parallel and stacking them one after the other leads to a mathematical object that is capable of approximating complicated functions. Different combinations of units respond to inputs in different ranges and for those parameters are relatively easy to optimize through gradient descent, because learning will behave a lot like that of a linear function until the output saturates.

5.1.2 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 whose parameters you can estimate surprisingly well through gradient descent. This fact remains true even when you're dealing with models with millions of parameters. What makes using deep neural networks so attractive is that it allows you not to worry too much about the exact function that represents your data, whether it's quadratic, piecewise polynomial, or something else. With a deep neural network model, you

have a universal approximator and a method to estimate its parameters. This approximator can be customized to your needs, in terms of model capacity and its ability to model complicated input/output relationships, by composing simple building blocks. Figure 5.6 shows some examples.

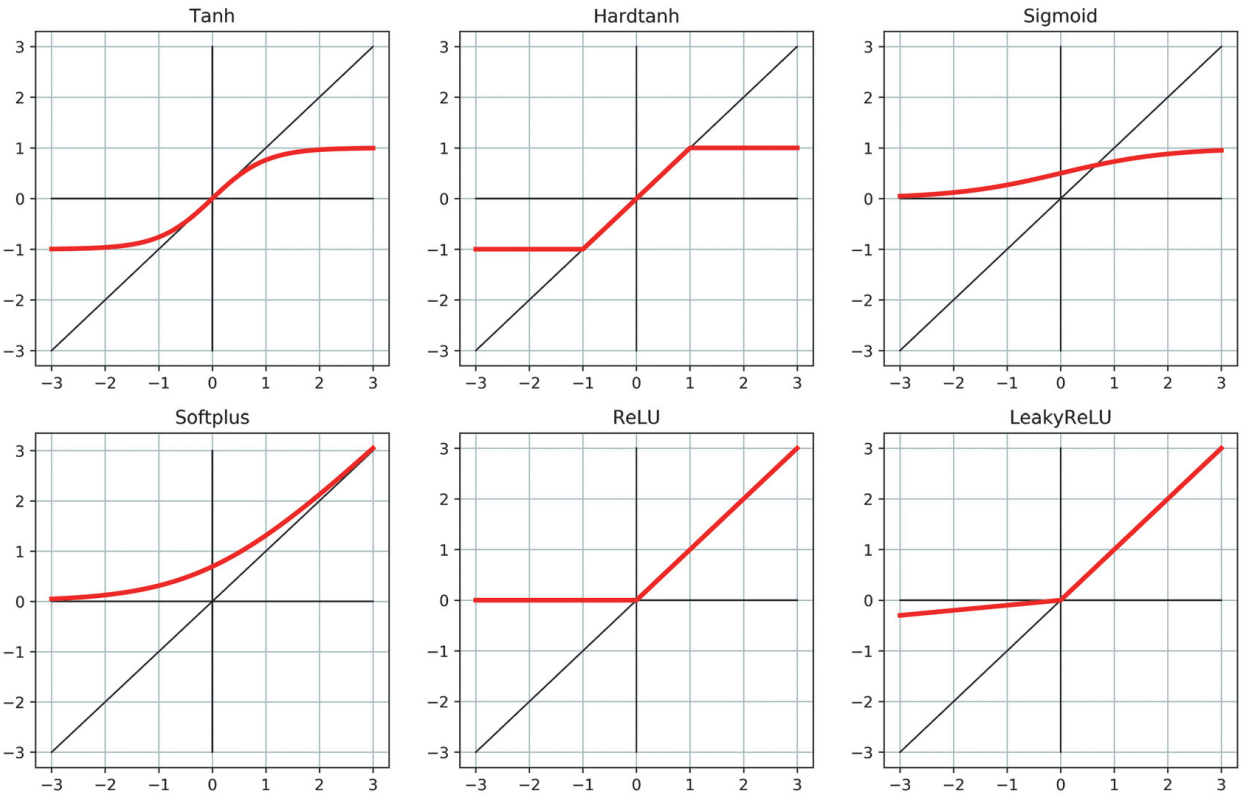


Figure 5.6 Composing multiple linear units and tanh activation functions to produce nonlinear outputs

The four top-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 minimum of -1 and a maximum of 1. The varied weights and biases move the center point and change how drastically the transition from min to max goes, but they clearly are all the same general shape. The columns to the right show both pairs of neurons added together (A+B and then C+D). Here, you start to see some interesting properties that mimic a single layer of neurons. A+B shows a slight S curve, with the extremes approaching zero, but both a positive and negative bump in the middle. Conversely, C+D has only a large positive bump, which peaks at a higher value than the single-neuron max of 1.

In the third row, you start to compose your 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* clear negative bumps and possibly a subtle second positive peak to the left of the main area of interest. All this occurs with only four neurons in two layers!

Again, these neurons' parameters were chosen only to create a visually interesting result. Training consists of finding acceptable values for these weights and biases so that the resulting network carries out a task correctly, 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 value of its weights and biases, captures the inherent structure of the data in the form of meaningful numerical representations that work correctly for previously unseen data.

Here's another step in your realization of the mechanics of learning: deep neural networks allow you to approximate highly nonlinear phenomena without having an explicit model for them. Instead, starting from a generic, untrained model, you specialize it on a task by providing it a set of inputs and outputs and a loss function to backpropagate from. Specializing a generic model to a task by 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 your thermometer experience, you assumed that both thermometers measured temperatures linearly. That assumption is where we implicitly encoded a rule for our task: we hard-coded 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 (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 it's possible to estimate the unknown parameters from measurements and get an accurate model of the world. Deep neural networks, at the other end, are families of functions that can approximate a wide range of input/output relationships without necessarily requiring one to come up with an explanatory model of a phenomenon. In a way, you're renouncing an explanation in exchange for the possibility of tackling increasingly complicated problems. In another way, you sometimes lack the ability, information, or computational resources to build an explicit model of what you're presented with, so data-driven methods are your only way forward.

5.2 The PyTorch nn module

All this talk about neural networks may be getting you curious about building one from scratch with PyTorch. The first step is replacing your linear model with a neural network unit. This step is a somewhat-useless step backward from a correctness perspective, because you've already verified that your calibration required only a linear function, but it'll still be instrumental for starting a sufficiently simple problem and scaling up later.

PyTorch has a whole submodule dedicated to neural networks: `torch.nn`. This submodule contains the building blocks needed to create all sorts of neural network

architectures. Those building blocks are called *modules* in PyTorch parlance (and *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 your linear model.) A `Module` can also have one or more submodules (subclasses of `nn.Module`) as attributes, and it can track their `Parameters` as well.

> **NOTE** The submodules must be top-level *attributes*, not buried inside `list` or `dict` instances! Otherwise, the optimizer won't be able to locate the submodules (and, hence, their parameters). For situations in which your model requires a list or dict of submodules, PyTorch provides `nn.ModuleList` and `nn.ModuleDict`.

Unsurprisingly, you 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`); it's equivalent to what you implemented earlier in your thermometer experiments. Now start precisely where you left off and convert your previous code to a form that uses `nn`.

All PyTorch-provided subclasses of `nn.Module` have their `call` method defined, which allows you to instantiate an `nn.Linear` and call it as though it were a function, as in the following listing.

Listing 5.1 code/p1ch6/1_neural_networks.ipynb

```
# In[5]:
import torch.nn as nn

linear_model = nn.Linear(1, 1)
linear_model(t_un_val)

# Out[5]:
tensor ([[-0.9852],
         [-2.6876]], grad_fn=<AddmmBackward>)
```

You look into the constructor arguments in a moment.

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 executes the forward computation; `call` does other rather important chores before and after calling `forward`. So it's technically possible to call `forward` directly, and it produces the same output as `call`, but it shouldn't be done from user code:

```
>>> y = model(x)
>>> y = model.forward(x)
```

Correct!

Silent error. Don't do it!

The following listing shows the implementation of `Module.call` (with some simplifications made for clarity).

Listing 5.2 torch/nn/modules/module.py, line 483, class: Module

```
def __call__(self, *input, **kwargs):
    for hook in self._forward_pre_hooks.values():
        hook(self, input)

    result = self.forward(*input, **kwargs)

    for hook in self._forward_hooks.values():
        hook_result = hook(self, input, result)
        # ...

    for hook in self._backward_hooks.values():
        # ...

    return result
```

As you can see, a lot of hooks won't get called properly if you use `.forward(…)` directly.

Now turn back to the 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 (defaulting to `True` here).

```
# In[5]:
import torch.nn as nn

linear_model = nn.Linear(1, 1)
linear_model(t_un_val)

# Out[5]:
tensor([[-0.9852],
        [-2.6876]], grad_fn=<AddmmBackward>)
```

The arguments are input size, output size, and bias defaulting to True.

The number of features in this case refers to the size of the input and the output tensor for the module, so 1 and 1. If you used both temperature and barometric pressure in input, for example, you'd have two features in input and one feature in output. As you'll see, for more complex models with several intermediate modules, the number of features is associated with the capacity of the model.

You have an instance of `nn.Linear` with one input and one output feature, which requires one weight and one bias:

```
# In[6]:
linear_model.weight

# Out[6]:
Parameter containing:
tensor([[-0.4992]], requires_grad=True)

# In[7]:
linear_model.bias

# Out[7]:
Parameter containing:
tensor([0.1031], requires_grad=True)
```

You can call the module with some input:

```
# In[8]:
x = torch.ones(1)
linear_model(x)

# Out[8]:
tensor([-0.3961], grad_fn=<AddBackward0>)
```

Although PyTorch let you get away with it, you didn't provide an input with the right dimensionality. You 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.

Any module in `nn` is written to produce outputs for a *batch* of multiple inputs at the same time. Thus, assuming that you need to run `nn.Linear` on 10 samples, you can create an input tensor of size `B x Nin`, where `B` is the size of the batch and `Nin` the number of input features, and run it once through the model:

```
# In[9]:
x = torch.ones(10, 1)
linear_model(x)

# Out[9]:
tensor([[-0.3961],
        [-0.3961],
        [-0.3961],
        [-0.3961],
        [-0.3961],
        [-0.3961],
        [-0.3961],
        [-0.3961],
        [-0.3961],
        [-0.3961]], grad_fn=<AddmmBackward>)
```

Figure 5.7 shows a similar situation with batched image data. The input is `BxCxHxW` with a batch size of three (say, images of a dog, bird, and then car), three channel dimensions (red, green, and blue), and an unspecified number of pixels for height and width.

As you can see, the output is a tensor of size `B x Nout`, where `Nout` is the number of output features—four, in this case.

The reason we want to do this batching is multi-faceted. One big motivation is to make sure that 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 that the batched results come back just as quickly as a single one would. Another benefit is that some advanced models will use statistical information from the entire batch, and those statistics get better with larger batch sizes.

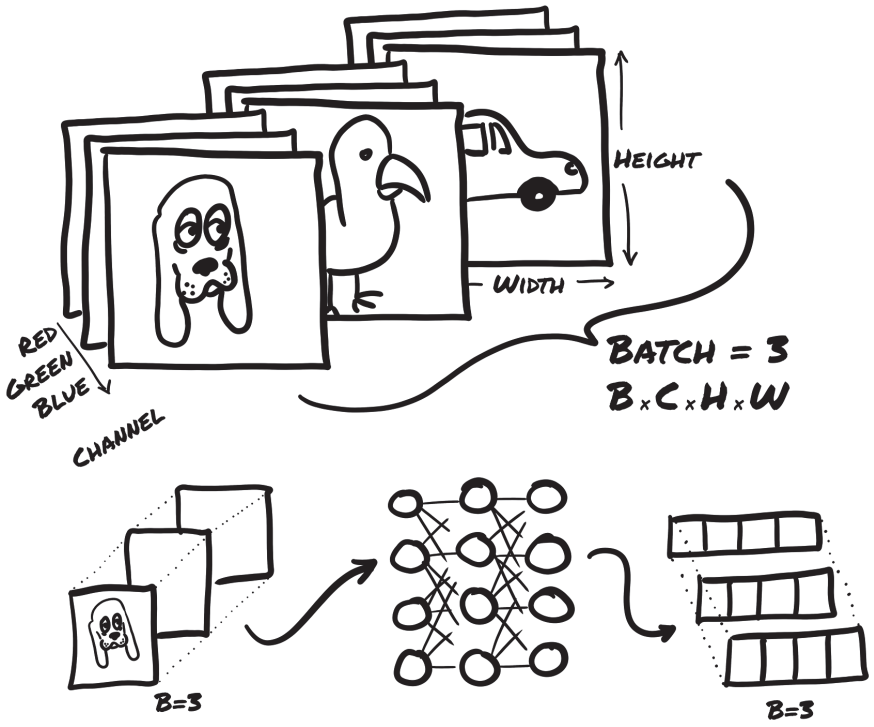


Figure 5.7 Three RGB images batched together and fed into a neural network. The output is a batch of three vectors of size 4.

Now turn back to the thermometer data. Your `t_u` and `t_c` were two 1D tensors of size `B`. Thanks to broadcasting, you could write your linear model as `w * x + b`, where `w` and `b` are two scalar parameters. This model works because you have one input feature; if you had two, you'd 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 you need to do to switch to using `nn.Linear`. You reshape your `B` inputs to `B x Nin`, where `Nin` is 1. You can easily do this 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, 21.0]
t_u = [35.7, 55.9, 58.2, 81.9, 56.3, 48.9, 33.9, 21.8, 48.4, 60.4, 68.4]
t_c = torch.tensor(t_c).unsqueeze(1)
t_u = torch.tensor(t_u).unsqueeze(1)

t_u.shape

# Out[2]:
torch.Size([11, 1])
```

Here, you add the extra dimension at axis 1.

You're done. Now update your training code. First, replace your handmade model with `nn.Linear(1,1)`; then pass the linear model parameters to the optimizer:

```
# In[10]:
linear_model = nn.Linear(1, 1)
optimizer = optim.SGD(
    linear_model.parameters(),
    lr=1e-2)
```

A redefinition from above.

You replace `[params]` with this method call.

Earlier, it was your responsibility to create parameters and pass them as the first argument to `optim.SGD`. Now you can ask any `nn.Module` for a list of parameters owned by it or any of its submodules by using the `parameters` method:

```
# In[11]:
linear_model.parameters()

# Out[11]:
<generator object Module.parameters at 0x0000020A2B022D58>

# In[12]:
list(linear_model.parameters())

# Out[12]:
[Parameter containing:
 tensor([[0.3791]], requires_grad=True), Parameter containing:
 tensor([-0.5349], 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 you can conveniently pass it to the optimizer constructor as you did earlier.

You can already figure out what happens in the training loop. The optimizer is provided a list of tensors that were defined with `requires_grad = True`. All Parameters are defined this way, by definition, because 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 iterates through each `Parameter` and changes it by an amount proportional to what is stored in its `grad` attribute, which is clean design.

Take a look at the training loop now:

```
# In[13]:
def training_loop(n_epochs, optimizer, model, loss_fn, t_u_train, t_u_val,
     t_c_train, t_c_val):
    for epoch in range(1, n_epochs + 1):
        t_p_train = model(t_un_train)
        loss_train = loss_fn(t_p_train, t_c_train)

        t_p_val = model(t_un_val)
        loss_val = loss_fn(t_p_val, t_c_val)

        optimizer.zero_grad()
        loss_train.backward()
        optimizer.step()

        if epoch == 1 or epoch % 1000 == 0:
            print('Epoch {}, Training loss {}, Validation loss {}'.format(
                epoch, float(loss_train), float(loss_val)))
```

Now the model is passed in instead of the individual params.

The loss function is also passed in. You'll use it in a moment.

The training loop hasn't changed practically except that now you don't pass `params` explicitly to `model` because the model itself holds its `Parameters` internally.

You can use one last bit from `torch.nn`: the loss. Indeed, `nn` comes with several common loss functions, among which `nn.MSELoss` (`MSE` stands for *Mean Square Error*), which is exactly what you defined earlier as your `loss_fn`. Loss functions in `nn` are still subclasses of `nn.Module`, so create an instance and call it as a function. In this case, you 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(),
    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 92.3511962890625, Validation loss 57.714385986328125
Epoch 1000, Training loss 4.910993576049805, Validation loss
     1.173926591873169
Epoch 2000, Training loss 3.014694929122925, Validation loss
     2.8020541667938232
Epoch 3000, Training loss 2.857640504837036, Validation loss
     4.464878559112549
Parameter containing:
tensor([[5.5647]], requires_grad=True)
Parameter containing:
tensor([-18.6750], requires_grad=True)
```

You're no longer using your handwritten loss function from earlier.

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.

It's been a long journey, with a lot to explore for these twenty-something lines of code. We hope that by now, the magic has vanished and left room for the mechanics. What you learn in this chapter will allow you to own the code you write instead of merely poking at a black box when things get more complicated.

You have one last step left to take: replacing your linear model with a neural network as your approximating function. As we said earlier, using a neural network won't result in a higher-quality model, because the process underlying the calibration problem is fundamentally linear. It's good to make the leap from linear to neural network in a controlled environment, however, so that you won't feel lost later on.

This section keeps everything else fixed, including the loss function, and redefines only the `model`. You'll 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, because its outputs aren't observed directly but fed into the output layer. Whereas the input and the 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 usually is larger than one. Recalling the earlier explanation on the role of activations, this situation can lead different units to respond to different ranges of the input, which increases the capacity of the model. The last linear layer takes the output of activations and combines them linearly to produce the output value.

`nn` provides a simple way to concatenate modules through the `nn.Sequential` container:

```
# In[16]:
seq_model = nn.Sequential(
            nn.Linear(1, 13),
            nn.Tanh(),
            nn.Linear(13, 1))
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)
)
```

13 was chosen arbitrarily. We wanted to pick a number that was a different size from the other various tensor shapes floating around.

This 13 must match the first size, however.

The 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.

Calling `model.parameters()` collects `weight` and `bias` from both the first and the 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]), torch.Size([1])]
```

These are the tensors that the optimizer will get. Again, after `model.backward()` is called, all parameters are populated with their `grad`, and then the optimizer updates their values accordingly during the `optimizer.step()` call, which isn't too different from the previous linear model. After all, both models are differentiable models that can be trained with gradient descent.

A few notes on parameters of `nn.Modules`: when you're inspecting parameters of a model made up of several submodules, it's handy to be able to identify parameters by their names. 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])
```

In fact, the name of each module in `Sequential` is the ordinal with which the module appeared in the arguments. Interestingly, `Sequential` also accepts an `OrderedDict`[4] in which you 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 code allows you 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])
```

You can also get to a particular `Parameter` by accessing submodules as though they were attributes:

[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.

```
# In[21]:
seq_model.output_linear.bias

# Out[21]:
Parameter containing:
tensor([-0.2194], requires_grad=True)
```

This code is useful for inspecting parameters or their gradients, such as to monitor gradients during training, as you did the beginning of this chapter. Suppose that you want to print out the gradients of `weight` of the linear portion of the hidden layer. You 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)

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 207.2268524169922, Validation loss 106.6062240600586
Epoch 1000, Training loss 6.121204376220703, Validation loss
     2.213937759399414
Epoch 2000, Training loss 5.273784637451172, Validation loss
     0.0025627268478274345
Epoch 3000, Training loss 2.4436306953430176, Validation loss
     1.9463319778442383
Epoch 4000, Training loss 1.6909029483795166, Validation loss
     4.027190685272217
Epoch 5000, Training loss 1.4900192022323608, Validation loss
     5.368413925170898
output tensor([[-1.8966],
        [11.1774]], grad_fn=<AddmmBackward>)
answer tensor([[-4.],
        [14.]])
hidden tensor([[-0.0073],
        [ 4.0584],
        [-4.5080],
        [-4.4498],
        [ 0.0127],
        [-0.0073],
        [-4.1530],
        [-0.6572]])
```

Note that the learning rate has dropped a bit to help with stability.

You can also evaluate the model on the whole data to see how different it is 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(), 'c-')
plt.plot(t_u.numpy(), seq_model(0.1 * t_u).detach().numpy(), 'kx')
```

This code produces figure 5.8.

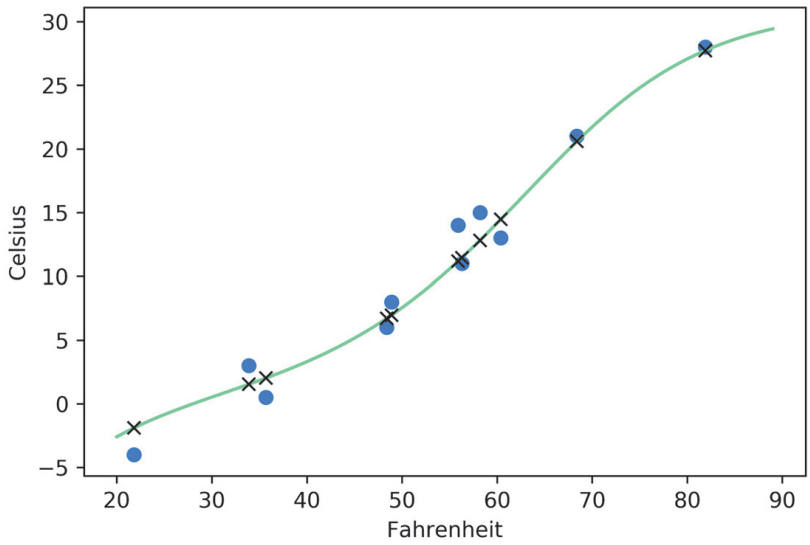


Figure 5.8 The plot of the neural network model, with input data (circles), desired output (`Xs`), and continuous line showing behavior between samples

You can appreciate that the neural network has a tendency to overfit because it tries to chase the measurements, including the noisy ones. It doesn't do a bad job overall, though.

5.3 Subclassing nn.Module

For larger and more complex projects, you need to leave `nn.Sequential` behind in favor of something that gives you more flexibility: subclassing `nn.Module`. To subclass `nn.Module`, at a minimum you need to define a `.forward(…)` function that takes the input to the module and returns the output. If you use standard `torch` operations, `autograd` takes care of the backward pass automatically.

> **NOTE** Often, your entire *model* is implemented as a subclass of `nn.Module`, which can in turn contain submodules that are also subclasses of `nn.Module`.

We'll show you three ways to implement the same network structure, using increasingly more complex PyTorch functionality to do so and varying the number of neurons in the hidden layer to make it easier to differentiate among the approaches.

The first method is a simple instance of `nn.Sequential`, as shown in the following listing.

Listing 5.3 code/p1ch6/3_nn_module_subclassing.ipynb

```
# In[2]:
seq_model = nn.Sequential(
            nn.Linear(1, 11),
            nn.Tanh(),
            nn.Linear(11, 1))
seq_model

# Out[2]:
Sequential(
  (0): Linear(in_features=1, out_features=11, bias=True)
  (1): Tanh()
  (2): Linear(in_features=11, out_features=1, bias=True)
)
```

The choice of 11 is somewhat arbitrary, but the sizes of the two layers must match.

Although this code works, you don't have any semantic information about what the various layers are intended to be. You can rectify that situation by giving each layer a label, using an ordered dictionary instead of a list as input:

```
# In[3]:
from collections import OrderedDict

namedseq_model = nn.Sequential(OrderedDict([
    ('hidden_linear', nn.Linear(1, 12)),
    ('hidden_activation', nn.Tanh()),
    ('output_linear', nn.Linear(12 , 1))
]))

namedseq_model

# Out[3]:
Sequential(
  (hidden_linear): Linear(in_features=1, out_features=12, bias=True)
  (hidden_activation): Tanh()
  (output_linear): Linear(in_features=12, out_features=1, bias=True)
)
```

Much better. You don't have any ability to control the flow of data through the network, however, aside from the purely sequential pass-through provided by the (aptly named!) `nn.Sequential` class. You can take full control of the processing of input data by subclassing `nn.Module` yourself:

```
# In[4]:
class SubclassModel(nn.Module):
    def __init__(self):
        super().__init__()
```

```
        self.hidden_linear = nn.Linear(1, 13)
        self.hidden_activation = nn.Tanh()
        self.output_linear = nn.Linear(13, 1)

    def forward(self, input):
        hidden_t = self.hidden_linear(input)
        activated_t = self.hidden_activation(hidden_t)
        output_t = self.output_linear(activated_t)

        return output_t

subclass_model = SubclassModel()
subclass_model

# Out[4]:
SubclassModel(
  (hidden_linear): Linear(in_features=1, out_features=13, bias=True)
  (hidden_activation): Tanh()
  (output_linear): Linear(in_features=13, out_features=1, bias=True)
)
```

This code ends up being somewhat more verbose, because you have to define the layers you want to have and then define how and in what order they should be applied in the `forward` function. That repetition grants you an incredible amount of flexibility in the sequential models, however, as you're now free to do all sorts of interesting things inside the `forward` function. Although this example is unlikely to make sense, you could implement `activated_t = self.hidden_activation(hidden_t) if random.random() > 0.5 else hidden_t` to apply the activation function only half the time! Because PyTorch uses a dynamic graph-based `autograd`, gradients would flow properly through the sometimes-present activation, no matter what `random.random()` returned!

Typically, you want to use the constructor of the module to define the submodules that we call in the `forward` function so that they can hold their parameters throughout the lifetime of your module. You might instantiate two instances of `nn.Linear` in the constructor and use them in `forward, for example`. Interestingly, assigning an instance of `nn.Module` to an attribute in a `nn.Module`, as you did in the constructor here, automatically registers the module as a submodule, which gives modules access to the parameters of its submodules without further action by the user.

Going back to the nonrandom `SubclassModel`, you see that the printed `output for that` class is similar to the output for the sequential model with named parameters. This makes sense, because you used the same names and intended to implement the same architecture. If you look at the parameters of all three models, you also see similarities there (except, again, for the differences in the number of hidden neurons):

```
# In[5]:
for type_str, model in [('seq', seq_model), ('namedseq', namedseq_model),
     ('subclass', subclass_model)]:
    print(type_str)
    for name_str, param in model.named_parameters():
        print("{:21} {:19} {}".format(name_str, str(param.shape), param.numel()))

    print()
```

```
# Out[5]:
seq
0.weight              torch.Size([11, 1]) 11
0.bias                torch.Size([11])    11
2.weight              torch.Size([1, 11]) 11
2.bias                torch.Size([1])     1

namedseq
hidden_linear.weight  torch.Size([12, 1]) 12
hidden_linear.bias    torch.Size([12])    12
output_linear.weight  torch.Size([1, 12]) 12
output_linear.bias    torch.Size([1])     1

subclass
hidden_linear.weight  torch.Size([13, 1]) 13
hidden_linear.bias    torch.Size([13])    13
output_linear.weight  torch.Size([1, 13]) 13
output_linear.bias    torch.Size([1])     1
```

What happens here is that the `named_parameters()` call delves into all submodules assigned as attributes in the constructor and recursively calls `named_parameters()` on them. No matter how nested the submodule is, any `nn.Module` can access the list of all child parameters. By accessing their `grad` attribute, which will have been populated by `autograd`, the optimizer knows how to change parameters so as to minimize the loss.

> **NOTE** Child modules contained inside Python `list` or `dict` instances won't be registered automatically! Subclasses can register those modules manually with the `add_module(name, module)` method of `nn.Module`[5] or can use the provided `nn.ModuleList` and `nn.ModuleDict` classes (which provide automatic registration for contained instances).

Looking back at the implementation of the `SubclassModel` class, and thinking about the utility of registering submodules in the constructor so that you can access their parameters, it appears to be a bit of a waste to also register submodules that have no parameters, such as `nn.Tanh`. Wouldn't it be easier to call them directly in the `forward` function?[6] It certainly would.

PyTorch has *functional* counterparts of every `nn` module. By *functional*, we mean "having no internal state" or "whose output value is solely and fully determined by the value input arguments." Indeed, `torch.nn.functional` provides many of the same modules you find in `nn`, but with all eventual parameters moved as an argument to the function call. The functional counterpart of `nn.Linear`, for example, is `nn.functional.linear`, which is a function that has signature `linear(input, weight, bias=None)`. The `weight` and `bias` parameters are arguments to the function.

---

[5] https://pytorch.org/docs/stable/nn.html#torch.nn.Module.add_module

[6] Aren't rhetorical questions great?

To get back to your model, it makes sense to keep using `nn` modules for `nn.Linear` so that `SubclassModel` can to manage all its `Parameter` instances during training. You can safely switch to the functional counterparts of Tanh, however, because it has no parameters:

```
# In[6]:
class SubclassFunctionalModel(nn.Module):
    def __init__(self):
        super().__init__()

        self.hidden_linear = nn.Linear(1, 14)

        self.output_linear = nn.Linear(14, 1)

    def forward(self, input):
        hidden_t = self.hidden_linear(input)
        activated_t = torch.tanh(hidden_t)
        output_t = self.output_linear(activated_t)

        return output_t

func_model = SubclassFunctionalModel()
func_model

# Out[6]:
SubclassFunctionalModel(
  (hidden_linear): Linear(in_features=1, out_features=14, bias=True)
  (output_linear): Linear(in_features=14, out_features=1, bias=True)
)
```

The `self.hidden_activation = ...` line is missing here.

That line was replaced with the equivalent functional call here.

The functional version is a bit more concise and fully equivalent to the non-functional version (as your models get more complicated, the saved lines of code start to add up!) Note that it would still make sense to instantiate modules that require arguments for their initialization in the constructor. `HardTanh`, for example, takes optional `min_val` and `max_val` arguments, and rather than repeatedly state those arguments in the body of `forward`, you could create a `HardTanh` instance and reuse it.

TIP Although general-purpose scientific functions like `tanh` still exist in `torch.nn.functional` in version 1.0, those entry points are deprecated in favor of ones in the top-level `torch` namespace. More niche functions remain in `torch.nn.functional`.

Conclusion

We covered a lot in this chapter, although we dealt with a simple problem. We dissected building differentiable models and training them by using gradient descent, using raw `autograd` first and then relying on `nn`. By now, you should have confidence in your understanding of what's going on behind the scenes.

We hope that this taste of PyTorch has given you an appetite for more!

More resources

A tremendous number of books and other resources are available to help teach deep learning. We recommend the following:

- The official PyTorch website: https://pytorch.org
- *Grokking Deep Learning*, by Andrew W. Trask[a], is a great resource for developing a strong mental model and intuition on the mechanism underlying deep neural networks.
- For a thorough introduction and reference to the field, we direct you to *Deep Learning*, by Ian Goodfellow, Yoshua Bengio, and Aaron Courville.[b]
- Last but not least, the full version of this book is available in Early Access now, with an estimated print date in late 2019: https://www.manning.com/books/deep-learning-with-pytorch.

a) https://www.manning.com/books/grokking-deep-learning
b) https://www.deeplearningbook.org

Exercises

- Experiment with the number of hidden neurons in your simple neural network model, as well as the learning rate.
  - What changes result in a more linear output from the model?
  - Can you get the model to obviously overfit the data?
- The third-hardest problem in physics is finding a proper wine to celebrate discoveries. Load the wine data from chapter 3 and create a new model with the appropriate number of input parameters.
  - How long does it take to train compared to the temperature data you've been using?
  - Can you explain what factors contribute to the training times?
  - Can you get the loss to decrease while training on this data set?
  - How would you go about graphing this data set?

Summary

- Neural networks can be automatically adapted to specialize in 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 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, provides 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 who does data science should be plotting data all the time.

index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

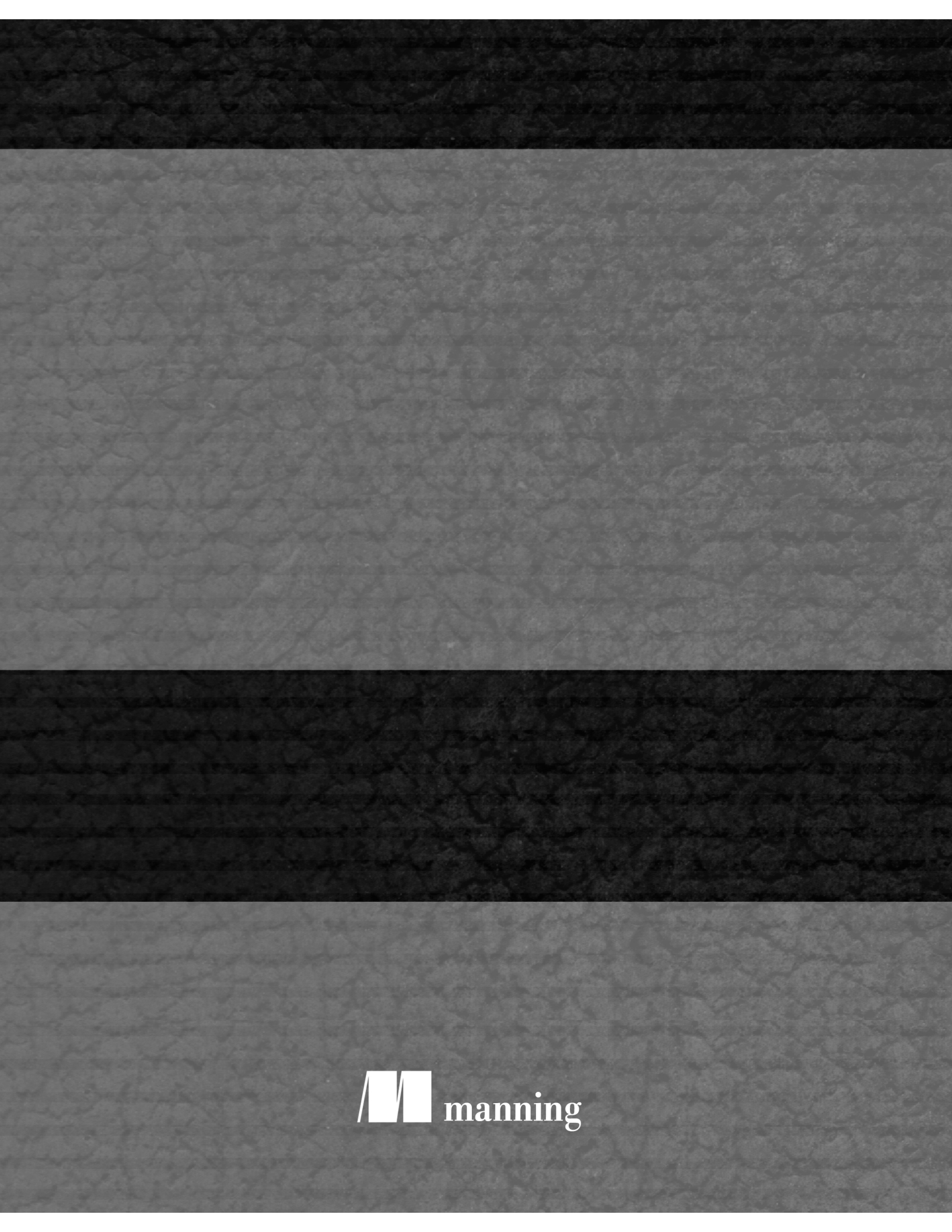
T

U

V

W

Z


manning