

Deep Learning with PyTorch

Eli Stevens
Luca Antiga
Thomas Viehmann



MANNING



MEAP Edition
Manning Early Access Program
Deep Learning with PyTorch
Version 12

Copyright 2020 Manning Publications

For more information on this and other Manning titles go to
<https://www.manning.com/>

welcome

Welcome to *Deep Learning with PyTorch!*

Eli and Luca here. We're ecstatic to have you with us. No, really — it's a big deal for us, both terrifying and exhilarating. So, thanks!

Our best wish for this book is that it'll help you develop your own intuition and stimulate your curiosity. PyTorch is an amazing library; it will give you new powers if you give it a few hours of your time.

We're having a lot of fun writing this book, but it'd be pretty lame if we are the only ones having fun. We're looking forward to being able to hear directly from you about what you like about the book, and what still needs work. We're adamant that the manuscript be as clear and of as much practical utility as possible, so please reach out. We want to know how you feel about the book, both good and bad. The good will give us fuel for the journey, while the bad keeps us out of the weeds.

One note, at the time of this writing the released version of PyTorch is 1.1. The May MEAP lockdown date was only a day or two after the PyTorch 1.1 release, so as of this version of the book, we're still using PyTorch 1.0 (we expect there won't be any changes needed, but we haven't tested that yet). Be aware that if you're trying to run the examples against a more recent PyTorch version than we've used you might run into some issues. We'll get those cleared up as soon as we can.

In the meantime, enjoy the book, say "hi!" on the [liveBook's Discussion Forum](#), and we'll chat again soon!

— Eli and Luca

brief contents

PART 1: CORE PYTORCH

1. *Introducing Deep Learning and the PyTorch Library*
2. *Pre-Trained Networks*
3. *It Starts with a Tensor*
4. *Real-World Data Representation Using Tensors*
5. *The Mechanics of Learning*
6. *Using A Neural Network To Fit the Data*
7. *Telling Birds from Airplanes: Learning from Images*
8. *Using Convolutions To Generalize*

PART II: LEARNING FROM IMAGES IN THE REAL-WORLD: EARLY DETECTION OF LUNG CANCER

9. *Using PyTorch To Fight Cancer*
10. *Ready, Dataset, Go!*
11. *Training A Classification Model To Detect Suspected Tumors*
12. *Monitoring Metrics: Precision, Recall, and Pretty Pictures*
13. *Using Segmentation To Find Suspected Nodules*
14. *Clustering and Diagnosis*

PART III: DEPLOYING PYTORCH MODELS

15. *Deploying to production*

Core PyTorch



Introducing Deep Learning and the PyTorch Library

1

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 are 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- to super-human levels of performance. For example, describing a photographic image with a sentence in idiomatic English; playing complex strategy games; and diagnosing a tumor from a radiological scan are all now approachable by a computer. Even more impressively, the ability to solve such tasks is acquired by computers *through examples*, rather than encoded by a human as a set of hand-crafted rules.

It would be disingenuous to assert that machines are learning to "think" in any human sense of the word. Rather, we've discovered a general class of algorithms that are able to approximate complicated, non-linear processes very, very effectively. In a way, we're learning that intelligence, as we subjectively perceive it, is a notion that we often conflate with self-awareness, and self-awareness is definitely not required to successfully solve or carry out these kinds of problems. In the end the question of computer intelligence might 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.

– Edsger W. Dijkstra *The threats to computing science*

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, like an input image and as output a line of text describing the input, or a written script as input and a natural-sounding voice reciting the script as output, or even more simply, associating an image of a golden retriever with a flag that tells us "yes, a golden retriever is present." This kind of capability allows us to create programs that has functionality that, until very recently, exclusively was the domain of human beings.

1.1 What is PyTorch?

PyTorch is a library for Python programs that facilitates building deep learning projects. It emphasises flexibility and allows deep learning models to be expressed in idomatic 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 across a broad range of applications.

PyTorch provides a core data structure, the `Tensor`, a multi-dimensional array that shares many similarities with Numpy arrays. From that foundation, a laundry list of features have been built that make it easy to get a project up and running, or an investigation into a new neural network architecture designed and trained. Tensors provide acceleration of mathematical operations (assuming 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 as well as 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; if it should be the last is a decision we'll leave to you.

1.2 What is this book?

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

Though we stress the practical applications, we also believe that providing an accessible

introduction to foundational deep learning tools like PyTorch is more than just a way to facilitate the acquisition of new technical skills. It is a step towards 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.

In order to get the most out of this book, readers will need two things. First, 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. Second, a willingness to dive in and get their hands dirty. We'll be starting from the basics and building up our working knowledge, and it will be much easier for you to learn it if you follow along with us.

Deep learning is a huge space. In this book we will be covering a tiny part of that space; specifically, using PyTorch for smaller-scope projects, with image processing of 2D and 3D datasets used for most of the motivating examples. This book focuses on practical PyTorch, with the aim of covering enough ground to allow the reader 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 pre-print repository, hosted at arxiv.org¹.

1.3 Why PyTorch

As we've said, deep learning allows us to carry out a very wide range of complicated tasks, like machine translation, playing strategy games or identifying objects in cluttered scenes, by exposing our model to illustrative examples. In order to do so in practice we need tools that are flexible, so they can be adapted to our specific problem; efficient, to allow training to occur over large amounts of data in reasonable times; and we need the trained network to perform correctly in the presence of uncertainty in the inputs. Let's 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 while 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 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. We'll dig into what that means more in chapter 5.

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 they don't exist until need for them arises. We'll

cover basic APIs and their use in Part 1 of this book.

A design driver for PyTorch is expressivity, allowing a developer to implement complicated models without undue complexity being imposed by the library (it's not 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 conferences². Part 2 will walk through a real-world problem and implement a solution step-by-step using PyTorch.

PyTorch also has a compelling story for the transition from research and development into production. While it was initially focused on research workflows, PyTorch has been equipped with a high-performance C{pp} runtime that can be leveraged to deploy models for inference without relying on Python. This allows us to keep most of the flexibility of PyTorch without having to pay the overhead of the Python runtime.

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

1.3.1 The Deep Learning Revolution

Let's 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 2000's, the broader class of systems that fell under the label "machine learning" heavily relied on *feature engineering*. Features are transformations on input data that result in numerical features that facilitate a downstream algorithm, like a classifier, to produce correct outcomes on new data. Feature engineering is aimed at taking the original data and coming up with *representations* of the same data that can then be fed to an algorithm to solve a problem. For instance, in order to tell 1's from 0's in images of handwritten digits, one would come up with a set of filters to estimate the direction of edges over the image, and then train a classifier to predict the correct digit given a distribution of edge directions. Another useful feature could be the number of enclosed holes, as seen in a zero, an eight, or in particularly loopy twos.

Deep learning, on the other hand, deals with finding such representations automatically, from raw data, in order to successfully perform a task. In the 1's vs 0's example, filters would be refined during training, by iteratively looking at pairs of examples and target labels. This is not to say that feature engineering has no place with deep learning; we often need to inject some form of prior knowledge in a learning system. However, the ability of a neural network to ingest data and extract useful representations on the basis of examples is what makes deep learning so powerful. The focus of deep learning practitioners is not so much on hand-crafting 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 hand-crafted! As with many disruptive technologies, this fact has led to a change in perspective.

On the left of 1.1, we see a practitioner busy defining engineering features and feeding them to a learning algorithm; the results on their task will be as good as the features he or she engineers. On the right, 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 ability of the practitioner to drive the algorithm towards its goal.

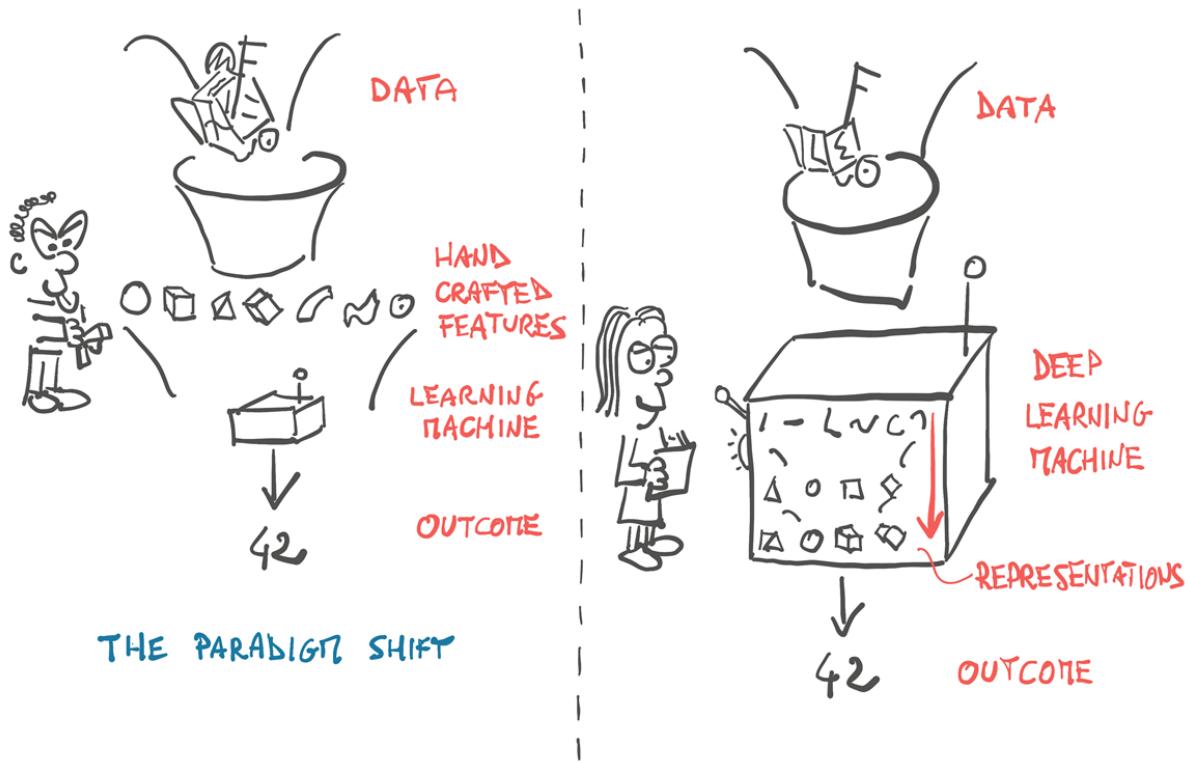


Figure 1.1 The change in perspective brought by deep learning.

1.3.2 Immediate vs. deferred execution

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

Consider the python expression `(a**2 + b**2) ** 0.5` that implements the Pythagorean theorem. If we want to execute this expression, we 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). This is useful because if there are problems 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, we could define the Pythagorean expression even before knowing what the inputs are, and then use that definition to produce the output once the inputs are available. That callable function that is defined 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, we defined a series of operations to perform, which resulted in a output *function* (`p` in this case). We didn't actually execute anything until later, when we passed in the inputs. That's deferred execution. That means that most exceptions will be raised when the function is called, not when it's defined. For normal Python, like we see here, that's fine, since the interpreter and debuggers have full access to the Python state at the time the error occurred.

Where things get tricky is when specialized classes are used that have heavy operator overloading, allowing what looks like immediate execution to actually be deferred under the hood. These 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, then taking the square root are not recorded as high-level python bytecode. Instead, the point is usually to compile the expression into a static computation graph (a graph of basic operations) that has some advantage over pure-Python (say, 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, as exceptions often lack specificity about what went wrong, and Python debugging tools don't have any visibility into the intermediate states of the data. Static graphs also don't usually mix well with standard Python flow control: they are de-facto domain-specific languages implemented on top of a host language (Python in our case).

Let's take a more concrete look at the differences between immediate and deferred execution, specifically regarding issues relevant to neural networks. In order to do so, we're going to need

to briefly cover a few topics that will have entire chapters devoted to them later in this book (chapters 5 and 6, specifically). We won't really be teaching these concepts in any depth here; this will just be a very high-level introduction to the terminology and the relationships between these concepts. Those concepts and relationships will lay the groundwork to understand how libraries like PyTorch that use immediate execution differ from deferred execution frameworks, even though the underlying math is the same for both.

The fundamental building block of a neural network is a *neuron*, which are strung together in large numbers to form the network. We can 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 different execution modes using the following figures, we'd like you to keep in mind:

- x is the input to our single-neuron computation.
- w and b are parameters or weights of the neuron, and can be changed as needed.³
- In order to update our parameters (to produce output that more closely matches what we desire), we assign error to each of the weights via back-propagation and then tweak the weights accordingly.⁴
- Back-propagation requires that we compute the gradient of the output with respect to the weights (among other things).⁵
- Automatic differentiation is used to compute the gradient automatically, saving us the trouble of writing the calculations by hand.

Back to Figure-1.2! Our neuron gets compiled into a symbolic graph where each node represents individual operations (second row), using placeholders for inputs and outputs. The graph is then evaluated numerically (third row) by plugging in concrete numbers into the placeholders (in this case, the numbers used are the values stored in w , x , and b). The gradient of the output with respect to the weights is constructed symbolically by automatic differentiation, which traverses the graph backwards and multiplies the gradients at individual nodes (fourth row). The corresponding mathematical expression is shown in the fifth row.

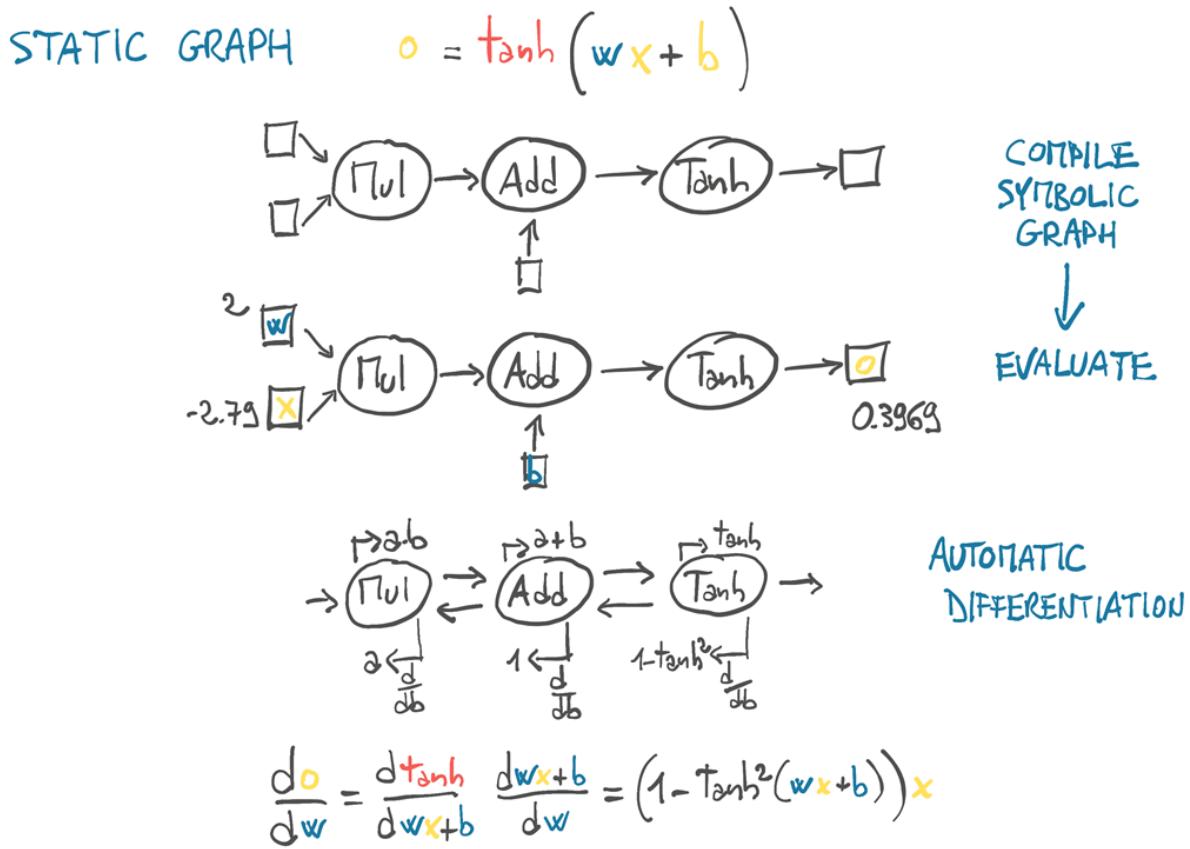


Figure 1.2 Static graph for a simple computation corresponding to a single neuron.

One of the major competing deep learning frameworks is TensorFlow, which has a "graph mode" that uses deferred execution similar to this. Graph mode is the default mode of operation in TensorFlow 1.x. In 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 upper half of Figure-1.3 shows the same calculation running under a dynamic graph engine. The computation is broken down into individual expressions, which are greedily evaluated as they are encountered. The program has no advance notion of the interconnection between computations. The lower half of the figure shows the behind-the-scenes construction of a dynamic computation graph for the same expression: the expression is still broken down into individual operations, but here they are eagerly evaluated, and the graph is built incrementally. Automatic differentiation is achieved by traversing the resulting graph backwards, similar to static computation graphs.

$$O = \tanh(wx + b)$$

$$X = -2.79$$

$$X_1 = wX = 2 \times (-2.79) = -5.58$$

$$X_2 = X_1 + b = -5.58 + 6 = 0.42$$

$$O = \tanh X_2 = \tanh 0.42 = 0.3969\dots$$

GREEDY EVALUATION
(NO GRAPH)

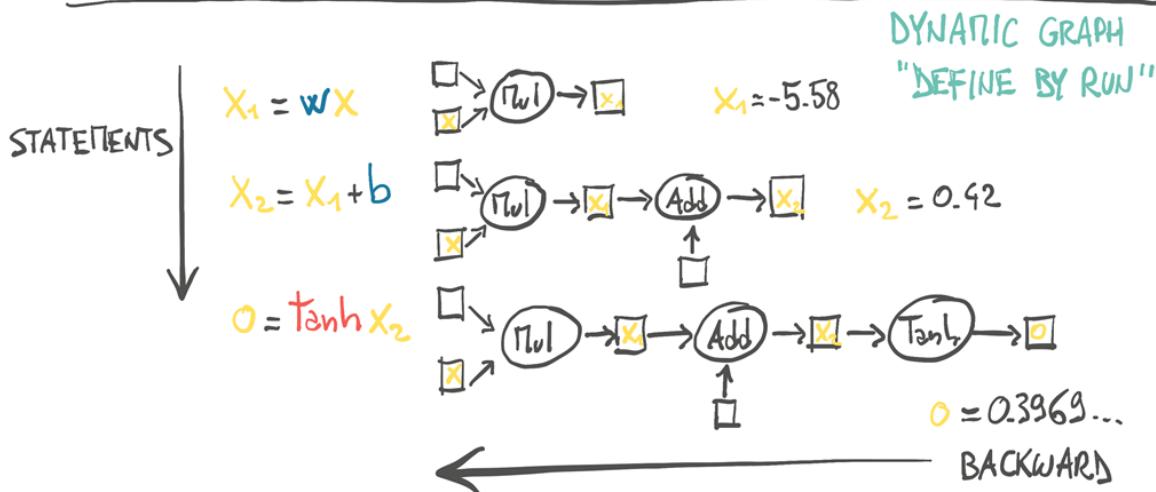


Figure 1.3 Dynamic graph for a simple computation corresponding a single neuron.

Dynamic graphs can change during successive forward passes, for instance different nodes can be invoked according to conditions on the outputs of the preceding nodes, without a need for such conditions to be represented in the graph itself. This is a distinct advantage over static graph approaches. Note that this does not mean dynamic graph libraries are inherently more capable than static graph libraries, just that it's often easier to accomplish looping or conditional behavior with dynamic graphs.

The major frameworks are converging towards supporting both modes of operation. PyTorch 1.0 gained the ability to record the execution of a model into a static computation graph or define it through a pre-compiled 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

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

NOTE

The deep learning landscape has been moving so quickly lately that by the time you read this in print, it will likely 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 additionally 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 has changed drastically. The community has largely consolidated behind either PyTorch or TensorFlow, with the adoption of other libraries dwindling or filling specific niches. In a nutshell:

- Theano, one of the first deep learning frameworks, has ceased active development.
- TensorFlow:
 - Consumed Keras entirely, promoting it to a first-class API
 - Provided an immediate execution "eager mode"
 - 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 industry-wide 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 since, as researchers and graduates train students and move to industry. Interestingly, with the advent of TorchScript and Eager mode, both have seen their feature sets start to converge with the other's.

1.4 PyTorch has the batteries included

We have already hinted at a few components in PyTorch. Let's now take some time to formalize a high-level map of the main components that form PyTorch.

First off, PyTorch has the "Py" as in Python, but there's a lot of non-Python code in it. Actually, for performance reasons, most of PyTorch is written in C{pp} and CUDA⁶, a C{pp}-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{pp}, and we'll look into those in chapter 13. One

of the main motivations for this capability is to provide a reliable strategy for deploying models in production. However, most of the time we'll interact with PyTorch from Python, building models, training them, and using the trained models to solve actual problems. Depending on a given use case's requirements for performance and scale, a pure-Python solution can be entirely sufficient to put models into production. It can be perfectly viable, for instance, to use a Flask web server to wrap a PyTorch model using the Python API.

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

At the core, PyTorch is a library that provides *multidimensional arrays*, called *tensors* in PyTorch parlance (we'll go into details on those in Chapter 3, 3) and an extensive library of operations on them, provided by the `torch` module. Both tensors and related operations can run on the CPU, or on the GPU. Running on the GPU results in massive speedups compared to CPU (especially if we're willing to pay for a top end GPU), and with PyTorch doing so doesn't require more than an additional function call or two. The second core thing that PyTorch provides is the ability of 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 back-propagation. This 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 "just" neural networks. Well, we would be correct: PyTorch can be used for physics, rendering, optimization, simulation, modeling - we're very likely to see PyTorch used in creative ways throughout the spectrum of scientific applications.

But PyTorch is first and foremost a deep learning library, and as such it provides all the building blocks needed to build neural networks and train them. Figure-1.4 shows a standard setup that loads data, trains a model, and then deploys that model to production.

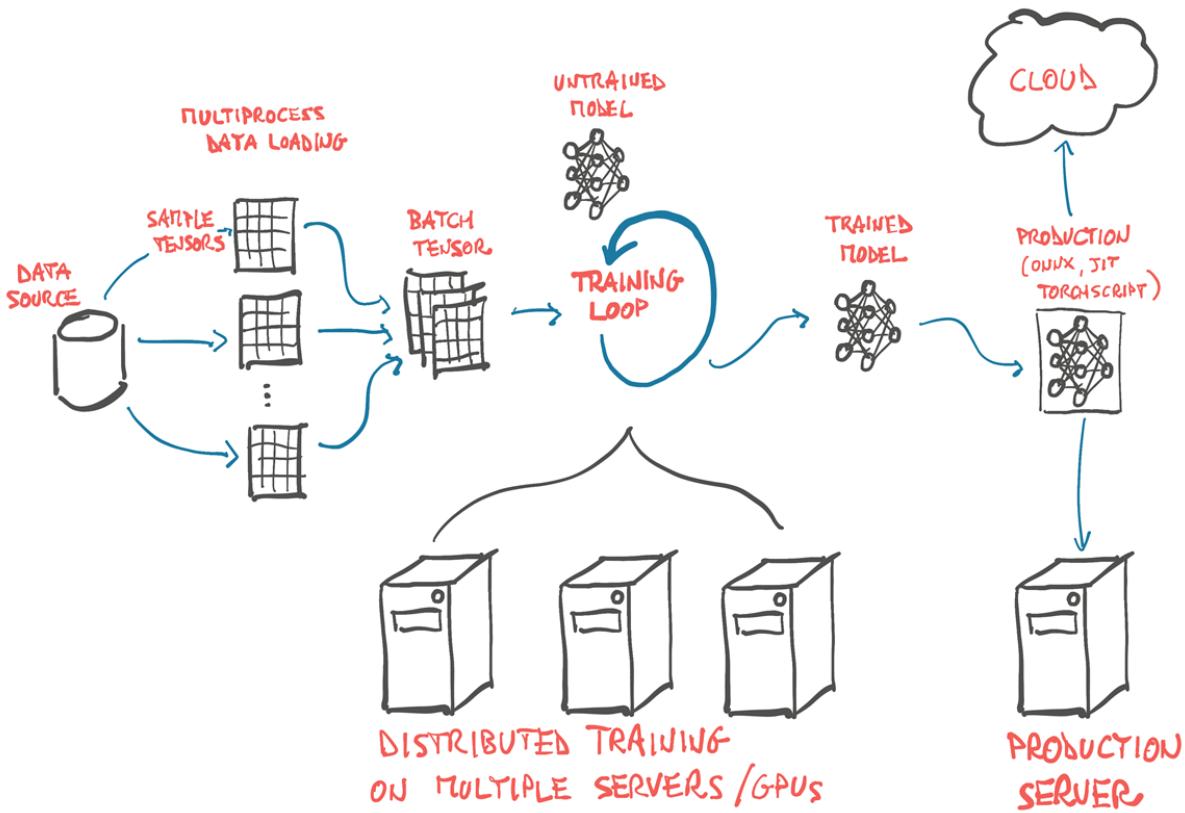


Figure 1.4 Basic, high-level structure of a PyTorch project, with data loading, training, and deployment to production.

The core PyTorch modules for building neural networks are located in `torch.nn`, which provides common neural network layers and other architectural components. Fully connected layers, convolutional layers, activation functions, and loss functions can all be found here (we'll go into more detail about what all of that means as we go through the rest of this book). These components can be used to build and initialize the untrained model we see in the center of Figure-1.4.

In order to train our model, we need a few things (besides the loop itself, which can just 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 actually 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 we will 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`. We'll go into details on tensors in chapter 3, 3. The other class we'll see a lot of is the `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 cast, the model will be running the required calculations on the local CPU or a

single GPU, and so once the training loop has the data, computation can start immediately. It's more common, however, to want to use specialized hardware like multiple GPUs or have multiple machines contribute their resources to training the model. In those cases, `torch.nn.DataParallel` and the `torch.distributed` can be employed to leverage the additional hardware available.

Once we have results from running our 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. That will be covered in chapter 5, 5.2.1.

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{pp} or CUDA implementation. As more instructions operate on tensors, more operations are executed by the backend implementation. This is typically as fast as it can be on the C{pp} side, but it incurs in the cost of calling that implementation through Python. It's a minute cost, 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. We can think about it as a virtual machine with a limited instruction set, specific to tensor operations. Besides not incurring in 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 at the basis of the production deployment capabilities of PyTorch. We'll cover this in chapter 15.

1.4.1 Hardware for deep learning

This book will require coding and running tasks that involve heavy numerical computing, such as multiplication of large matrices. As it turns out, running a pre-trained network on new data is within the capabilities of any recent laptop or personal computer. Even taking a pre-trained network and re-training a small portion of it to specialize it on a new dataset doesn't necessarily require specialized hardware. Everything we do in Part 1 of this book can be followed along using a standard personal computer or laptop. However, we anticipate that completing a full training run for the more advanced examples in Part 2 will require a CUDA-capable Graphical Processing Unit (GPU). The default parameters used in Part 2 assume a GPU with 8GB of RAM (we suggest an NVIDIA GTX 1070 or better), but those can be adjusted if your hardware has less RAM available.

To be clear: such hardware is not mandatory if you're willing to wait, but running on a GPU cuts training time by at least an order of magnitude (and usually it's 40-50x faster). Taken individually, the operations required to compute parameter updates are fast (from fractions of a

second, to a few seconds) on modern hardware like a typical laptop CPU. The issue is that training involves running these operations over and over, many, many times, incrementally updating the network parameters to minimize the training error.

Moderately large networks can take hours to days to train from scratch on large, real-world datasets on workstations equipped with a good GPU. That time can be reduced by using multiple GPUs on the same machine, and even further on clusters of machines equipped with multiple GPUs. These setups are less prohibitive to access than it sounds, thanks to the offerings of cloud computing providers. DAWN Bench⁷ is an interesting initiative from Stanford University aimed at providing benchmarks on training time and cloud computing costs related to common deep learning tasks on publicly available datasets.

So, if there's a GPU around by the time we reach Part 2, then great. Otherwise we suggest readers to check out the offerings from the various cloud platforms, many of which offer GPU-enabled Jupyter Notebooks with PyTorch pre-installed, 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. Since current Apple laptops do not include GPUs that support CUDA, the pre-compiled macOS packages for PyTorch are CPU-only. Throughout the book we will try to avoid assuming the reader is running a particular OS, though some of the scripts in Part 2 are shown as if running from a Bash prompt under Linux. Those scripts' command lines should convert to a Windows-compatible form readily. For convenience, code will be listed as if running from a Jupyter Notebook, when possible.

For installation information, please see the getting started guide on the official website⁸. We suggest Windows users install with Anaconda or Miniconda. Other operating systems like Linux typically have a wider variety of workable options, with Pip being one of the more common installers. Of course, experienced users are free to install packages in the way that is most compatible with their preferred development environment.

Part 2 has some non-trivial download bandwidth and disk space requirements as well. The raw data needed for the cancer detection project in part 2 is about 60GB to download, and when uncompressed requires about 120GB of space. The compressed data can be removed after decompressing it. In addition, due to caching some of the data for performance reasons, another 80GB will be needed while training. Readers will need a total of 200GB (at minimum) of free disk space on the system that will be used for training. While it is possible to use network storage for this, there might be training speed penalties if the network access is slower than local disk.

1.4.2 Using Jupyter notebooks

We're going to assume you've gotten PyTorch and the other dependencies installed, and have verified that things are working. Earlier we touched upon the possibilities for following along with the code in the book. We are going to be making heavy use of Jupyter Notebooks for our example code. A Jupyter Notebook shows itself as a page in the browser through which we can run code interactively. The code gets evaluated by a *kernel*, a process running on a server that is ready to receive code to execute and send back the results, which are then rendered inline on the page. A notebook maintains the state of the kernel, like variables defined during the evaluation of code, in memory until it is terminated or restarted. The fundamental unit with which we interact with a notebook is a *cell*, a box on the page where we can type code and have the kernel evaluate it (through the menu item or just using Shift-Enter). We can add multiple cells in a notebook, and the new cells will see the variables we created in the earlier cells. The value returned by the last line of a cell will be printed right below the cell after execution, and the same goes for plots. By mixing source code, results of evaluations and Markdown-formatted text cells, we can generate beautiful interactive documents. One can read everything about Jupyter Notebooks on the project website.⁹

At this point, you'll need to start the notebook server from the root directory of the code checkout from github. How exactly starting the server looks depends on the details of your operating system, and how and where you installed Jupyter. If you have questions, feel free to ask on our forums.¹⁰ Once started, your default browser will pop up, showing a list of local notebook files.

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

Full working code for all listings from the book can be found in our repository on GitHub.¹¹

1.5 Conclusion

In this chapter we introduced where the world stands with deep learning and what tools one can use to be part of the revolution. We have taken a peek into what PyTorch has to offer and why it is worth investing time and energy in it. And we have described what PyTorch looks like from a bird's-eye view, as well as touched on prerequisites for getting the most from this book.

As with any good story, wouldn't it be great to take a peek at the amazing things PyTorch will enable us to do once we've completed our journey? Hold tight, the next chapter is aimed at exactly that.

1.6 Exercises

- Start Python to get an interactive prompt
 - What Python version are you using? Is it Python 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?

1.7 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 pre-compiled deferred execution mode that can be invoked from C{pp}.
- Since the release of PyTorch in early 2017, the deep learning tooling ecosystem has consolidated significantly.
- PyTorch provides a number of utility libraries to facilitate deep learning projects

Pre-Trained Networks



This chapter covers:

- Running pre-trained image recognition models on sample data
- An introduction to GANs (generative adversarial networks) and CycleGAN
- Captioning models that can produce text descriptions of images
- Sharing models through TorchHub

We closed our first chapter promising to unveil amazing things in this chapter, and now it's time to deliver.

Computer vision is certainly one of the fields that have been most impacted by the advent of deep learning, for a variety of reasons. The need for classifying or interpreting the content of natural images existed, very large datasets became available and new constructs such as convolutional layers were invented and could be ran quickly on GPUs with unprecedented accuracies. All this combined with the motivation of the Internet giants to understand pictures shot by millions of users through their mobile devices and managed on said giants' platforms. Quite the perfect storm.

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

In this chapter we will explore three popular pre-trained models, namely a) a model that can label an image according to its content, b) another that can fabricate a new image from a real image, and c) a model that can describe the content of an image using proper English sentences. We will learn how to load and run these pre-trained models in PyTorch, and we will introduce PyTorch Hub, a set of tools through which PyTorch models like the pre-trained ones covered can be easily made available through a uniform interface. Along the way, we'll discuss data sources, define terminology like "label," and attend a zebra rodeo.

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

We hope you read through, though, since this chapter hides some important skills under the fun. Learning how to run a pre-trained model using PyTorch is a useful skill, full stop. It's especially useful if the model has been trained on a large dataset. We will need to get accustomed with the mechanics of obtaining and running a neural network on real-world data, and then visualizing and evaluating its outputs, no matter if we trained it or not.

2.1 A pre-trained network that recognizes the subject of an image

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

The pre-trained network we'll explore now has been trained on a subset of the ImageNet dataset¹². ImageNet is a very large dataset of over 14 million images maintained by Stanford University. All images are labeled with a hierarchy of nouns coming from the WordNet dataset,¹³ in turn a large lexical database of the English language.

The ImageNet dataset, like several other public datasets, has its origin in academic competitions. Competitions have traditionally been one of the main playing fields where researchers at institutions and companies regularly challenge each other. Among others, the Large Scale Visual Recognition Challenge (ILSVRC), which takes place on ImageNet, has gained popularity since its inception in 2010. This particular competition is based on a few tasks, which can vary by the

year, such as image classification (telling what object categories the image contains), object localization (identifying objects' position in images), object detection (identifying and labeling objects in images), scene classification (classifying a situation in an image), scene parsing (segmenting an image into regions associated with semantic categories, such as cow, house, cheese, hat). In particular, the image classification task consists in taking an image in input and producing a list of 5 labels out of 1000 total categories, ranked by confidence, describing the content of the image.

The training set for ILSVRC consists in 1.2 million images, labeled with one of 1000 nouns (e.g. "dog"), referred to as the *class* of the image. In this sense, we will use the terms *label* and *class* interchangeably. We can take a peek at images from ImageNet in Figure-2.2.

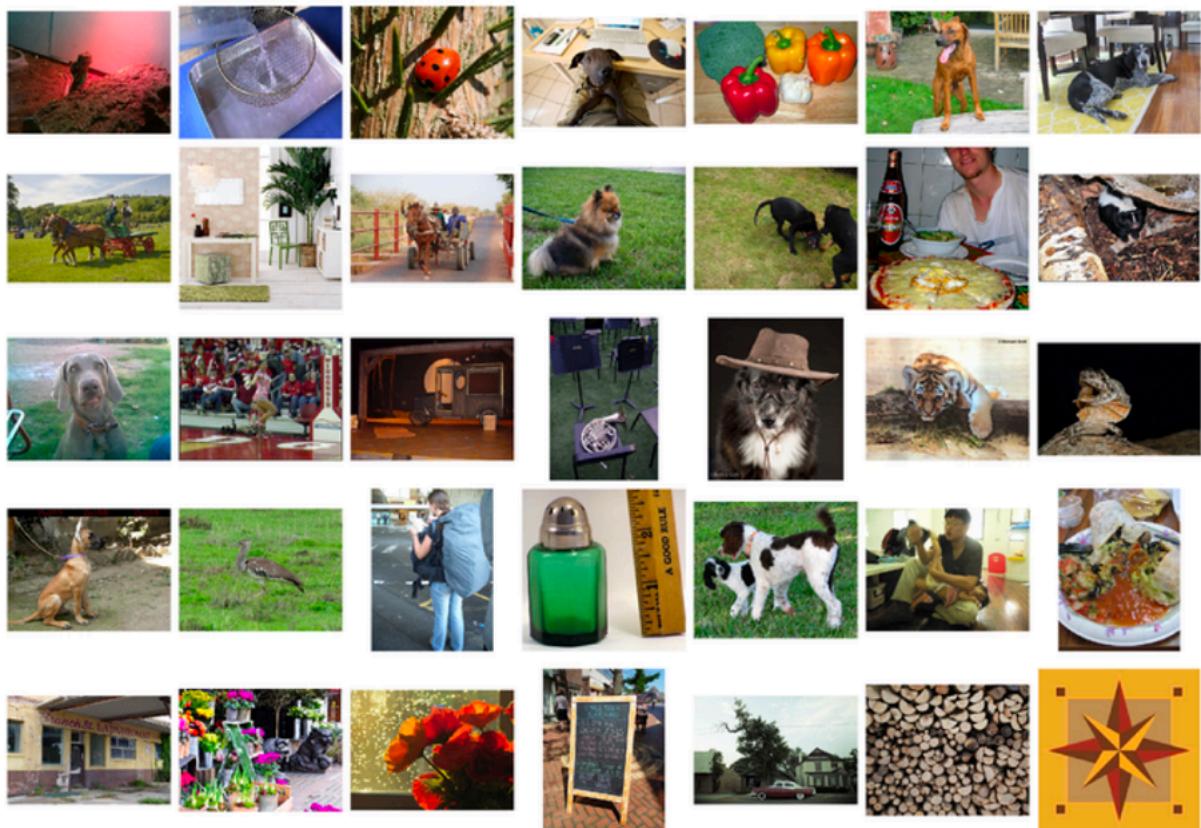


Figure 2.1 A small sample of ImageNet images

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

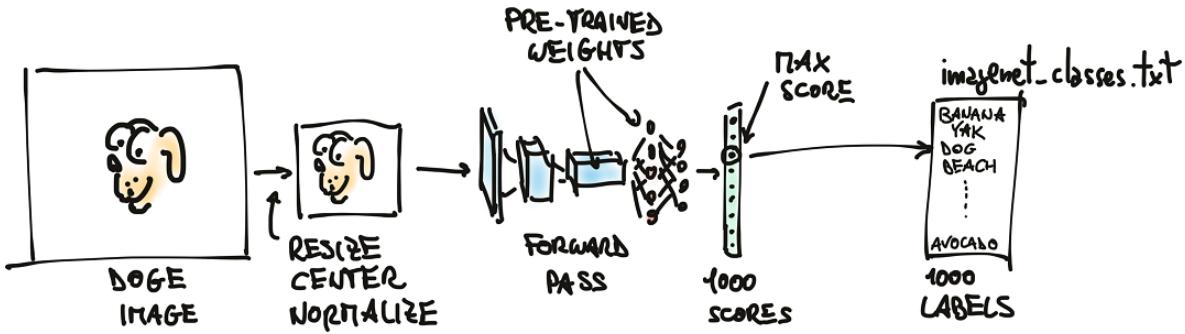


Figure 2.2 The inference process.

The input image will first be pre-processed into a instance of the multi-dimensional array class `torch.Tensor`. It is an RGB image with height and width, so this tensor will have three dimensions, the 3 color channels and two spatial image dimensions of a specific size. We'll get into the details of what a tensor is in chapter 3, 3, but for now think of it as like a vector or matrix of floating-point numbers. Our model will take that processed input image and pass it into the pre-trained network to obtain scores for each class. The highest score corresponds to the most likely class according to the weights. Each class is then mapped one-to-one onto a class label. That output is contained in a `torch.Tensor` with 1000 elements, each representing the score associated with that class.

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

2.1.1 Obtaining a pre-trained network for image recognition

As discussed, we will now equip ourselves with a network trained on ImageNet. To do so, we'll take a look at the TorchVision project,¹⁴ which contains a few of the best performing neural network architectures for computer vision, such as AlexNet¹⁵, ResNet¹⁶, and Inception v3¹⁷. It also has easy access to datasets like ImageNet and other utilities for getting up to speed with computer vision applications in PyTorch. We'll dive into what some of these are further along in the book. For now let's load up and run two networks, first AlexNet, one of early breakthrough networks for image recognition, and then a residual network, ResNet for short, which won the ImageNet classification, detection and localization competitions, among others, in 2015. If you didn't get PyTorch up and running in chapter 1, now is a good time to go back and do that.

The pre-defined models can be found in `torchvision.models`.

Listing 2.1 code/p1ch2/2_pre_trained_networks.ipynb

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

We can take a look at the actual models:

```
# In[2]:  
dir(models)  
  
# Out[2]:  
['AlexNet',  
 'DenseNet',  
 'Inception3',  
 'ResNet',  
 'SqueezeNet',  
 'VGG',  
 ...  
 'alexnet',  
 'densenet',  
 'densenet121',  
 ...  
 'resnet',  
 'resnet101',  
 'resnet152',  
 ...  
 ]
```

The capitalized names refer to Python classes that implement a number of popular models. They differ in their architecture, that is, in the arrangement of the operations occurring between the input and the output. The lowercase names are instead convenience functions that return models instantiated from those classes, sometimes with different parameter sets. For instance, `resnet101` returns an instance of `ResNet` with 101 layers, `resnet18` with 18 layers and so on. We'll now turn our attention to AlexNet.

2.1.2 AlexNet

The AlexNet architecture won the 2012 ImageNet Large-Scale Visual Recognition Challenge by a large margin, with a top 5 test error rate (i.e. correct label must be in the top 5 predictions) of 15.4%. By comparison, the second best submission, not based on a deep network, trailed at 26.2%. It was a defining moment in the history of computer vision, the moment when the community started to realize the potential of deep learning for vision tasks. That leap was followed by constant improvement, with more modern architectures and training methods getting top 5 error rates as low as 3%.

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

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

ALEXNET

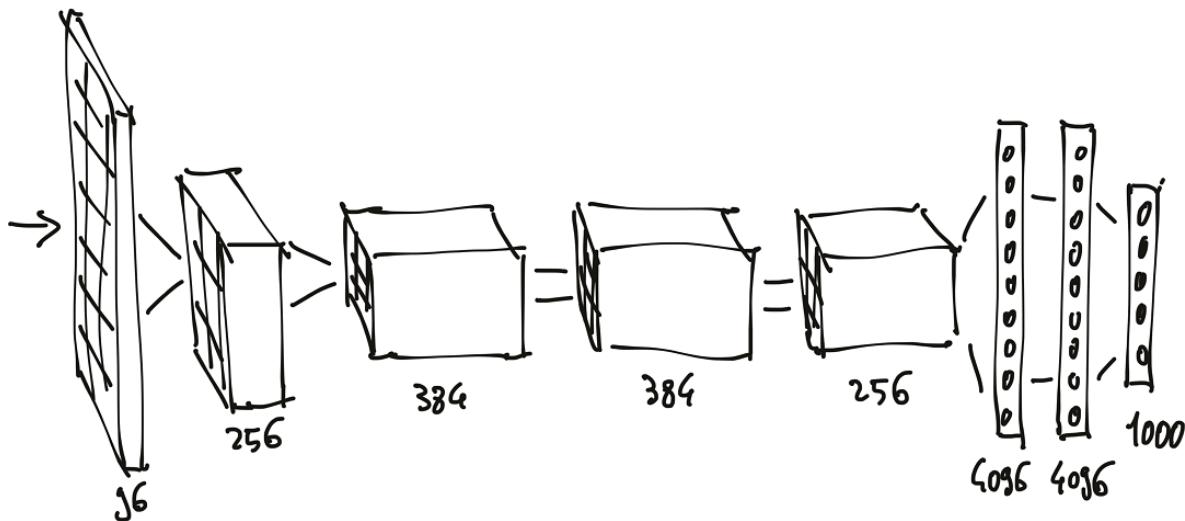


Figure 2.3 The AlexNet architecture.

Back to the diagram, input images come in from the left and go through five stacks of filters, each producing a number of output images as reported in the figure. After each filter images are reduced in size, as annotated. The images produced by the last stack of filters are layed out as a 4096-elements 1D vector and classified to produce 1000 output probabilities, one for each output class.

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

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

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

We would have just fed data through the whole network to produce... garbage! That's because the network is uninitialized: its weights, the numbers by which inputs are added and multiplied, have not been trained on anything - the network itself is a blank (or rather, random) slate. We'd need to either train it from scratch or load weights from a prior training, which we'll do now.

To this end, let's go back to the `models` module. We learned that the uppercase names correspond to classes that implement popular architectures for computer vision. The lowercase

names, on the other end, are functions that instantiate models with pre-defined number of layers and units and optionally download and load pre-trained weights into them. Note that there's nothing fundamental about using one of these functions. They just make it convenient to instantiate the model with a number of layers and units that matches how pre-trained networks were built.

2.1.3 ResNet

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

Let's create an instance of the network now. We'll pass an argument that will instruct the function to download the weights of a ResNet101 trained on the ImageNet dataset, with 1.2 million images and 1000 categories.

```
# In[4]:  
resnet = models.resnet101(pretrained=True)
```

It's downloading. While we're staring at the download progress, we can take a minute time to appreciate that ResNet101 sports 44.5 million parameters - that's a lot of parameters to optimize automatically!

2.1.4 Ready, set, almost run

Ok, what did we just get? Since we're curious, we'll take a peek at what a ResNet101 looks like. We can do so just by printing the value of the returned model. This gives us a textual representation of the same kind of information we saw in Figure-2.3, which gives us some details about the structure of the network. For now, this will be information overload, but as we progress through the book, we'll increase our ability to understand what this listing is telling us.

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

What we are seeing here is modules, one per line. Note that they have nothing in common with

Python modules: they are individual operations, the building blocks of a neural network. They are also called *layers* in other deep learning frameworks.

If we scroll down, we'll see a lot of `Bottleneck` modules repeating one after the other (101 of them!), containing convolutions and other modules. That's the anatomy of a typical deep neural network for computer vision: a more or less sequential cascade of filters and non-linear functions, ending with a layer (`fc`) producing scores for each of the 1000 output classes (`out_features`).

The `resnet` variable can be called like a function, taking in input one or more images and producing an equal number of scores for each of the 1000 ImageNet classes. Before we can do that, however, we have to pre-process any input image so that it has the right size and so that its values (its colors) sit roughly in the same numerical range. In order to do that, the `torchvision` module provides `transforms`, which allow to quickly define pipelines of basic pre-processing functions:

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

In this case, we defined a `preprocess` function that will scale the input image to 256x256, crop the image to 224x224 around the center, transform it to a tensor (a PyTorch multidimensional array, a 3D array with color, height, and width in this case), and normalize its RGB (red, green, blue) components so that they have defined means and standard deviations. These need to match what was presented to the network during training, if we wish to hope that the network will produce meaningful answers. We'll go in more in depth into `transforms` when we dive into making our own image recognition models, in 7.1.3.

We can now grab a picture of our favorite dog (say, `bobby.jpg` from the GitHub repo), preprocess it, and then see what the ResNet thinks of it. We can start by loading an image from the local filesystem using Pillow¹⁸, an image manipulation module for Python:

```
# In[7]:  
from PIL import Image  
img = Image.open("../data/plch2/bobby.jpg")
```

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

```
# In[8]:  
img
```

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

Otherwise we can invoke the `show` method, which will pop up a window with a viewer:

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



Figure 2.4 Bobby, our very special input image.

We can now pass the image through our pre-processing pipeline:

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

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

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

We're now ready to run our model.

2.1.5 Run!

The process of running a trained model on new data is called *inference* in deep learning circles. In order to do inference, we need to put the network in *eval* mode.

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

```
(avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
(fc): Linear(in_features=2048, out_features=1000, bias=True)
)
```

If we forget to do the above, several pre-trained models will not produce meaningful answers, just because the way some modules, like *batch normalization* or *dropout*, work internally. Now that `eval` has been set, we're ready for inference.

```
# In[12]:
out = resnet(batch_t)
out

# Out[12]:
tensor([[-3.4803, -1.6618, -2.4515, -3.2662, -3.2466, -1.3611,
        -2.0465, -2.5112, -1.3043, -2.8900, -1.6862, -1.3055,
        ...
        2.8674, -3.7442,  1.5085, -3.2500, -2.4894, -0.3354,
        0.1286, -1.1355,  3.3969,  4.4584]])
```

A staggering set of operations involving 44.5 million parameters has just happened, producing a vector of 1000 scores, one per ImageNet class. That didn't take long, did it?

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

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

Let's load the file containing the 1000 labels for the ImageNet dataset classes:

```
# In[13]:
with open('../data/plch2/imagenet_classes.txt') as f:
    labels = [line.strip() for line in f.readlines()]
```

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

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

We can now use the index to access the label. Here `index` is not a plain Python number, but a one-element 1-dimensional tensor (specifically `tensor([207])`), so we need to get the actual numerical value to use as an index into our `labels` list using `index[0]`. We also use `torch.nn.functional.softmax`¹⁹ to normalize our outputs to the range [0, 1], and divide by

the sum. That gives us something roughly akin to the confidence that the model has in its prediction. In this case, it's 96% certain that it knows what it's looking at is a golden retriever.

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

# Out[15]:
('golden retriever', 96.29334259033203)
```

Oh oh, who's a good boy?

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

```
# In[16]:
_, indices = torch.sort(out, descending=True)
[(labels[idx], percentage[idx].item()) for idx in indices[0][:5]]

# Out[16]:
[('golden retriever', 96.29334259033203),
 ('Labrador retriever', 2.80812406539917),
 ('cocker spaniel, English cocker spaniel, cocker', 0.28267428278923035),
 ('redbone', 0.2086310237646103),
 ('tennis ball', 0.11621569097042084)]
```

We see that the first four are dogs (redbone is a breed; who knew?), then things start to get funny. The fifth answer of "tennis ball" is probably because there are enough pictures of tennis balls with dogs nearby that the model is essentially saying "there's a 0.1% chance that I've completely misunderstood what a tennis ball is."

This is a great example of the fundamental differences in how humans and neural networks view the world, as well as how easy it is for strange, subtle biases to sneak into our data.

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

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

2.2 A pre-trained model that fakes it until it makes it

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

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

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

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

2.2.1 The GAN game

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

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

The *generator* network takes the role of the painter in our scene above, tasked with producing realistic-looking images starting from an arbitrary input. The *discriminator* network is the amoral

art inspector, needing to tell whether a given image was fabricated by the generator or it belonged in a set of real images. This two-network design is atypical for most deep learning architectures, but when used to implement a GAN game can lead to incredible results.

In Figure-2.6 there's a rough picture of what's going on. The end-goal for the generator is to fool the discriminator into mixing up real and fake images. The end-goal for the discriminator is to find out when it's being tricked, but it also helps inform the generator about the identifiable mistakes that the generated images have. At the start, the generator produces confused, three-eyed monsters that look nothing like a Rembrandt portrait. The discriminator is easily able to distinguish the muddled messes from the real paintings. As training progresses, information flows back from the discriminator, and the generator uses that to improve. By the end of training, the generator is able to produce convincing fakes, and the discriminator no longer is able to tell which is which.

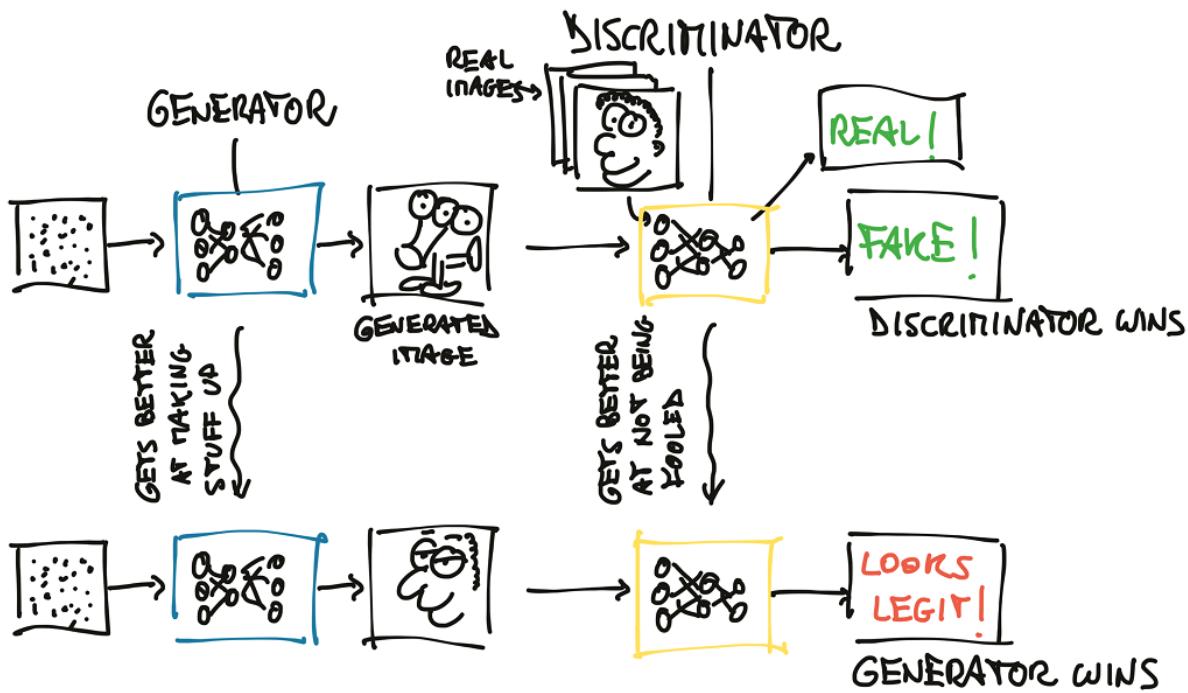


Figure 2.5 Concept of a GAN game.

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

This technique has proven itself to be able to lead to generators that produce realistic images just out of noise and a conditioning signal, like an attribute (e.g. for faces, young, female, glasses on), or another image. In other words, a well-trained generator learns a plausible model for generating realistic-looking images, even when examined by humans.

2.2.2 CycleGAN

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

In Figure-2.7 we have a CycleGAN workflow for the task of turning a photo of a horse into a zebra, and vice versa. Note that there are two separate generator networks, as well as two distinct discriminators.

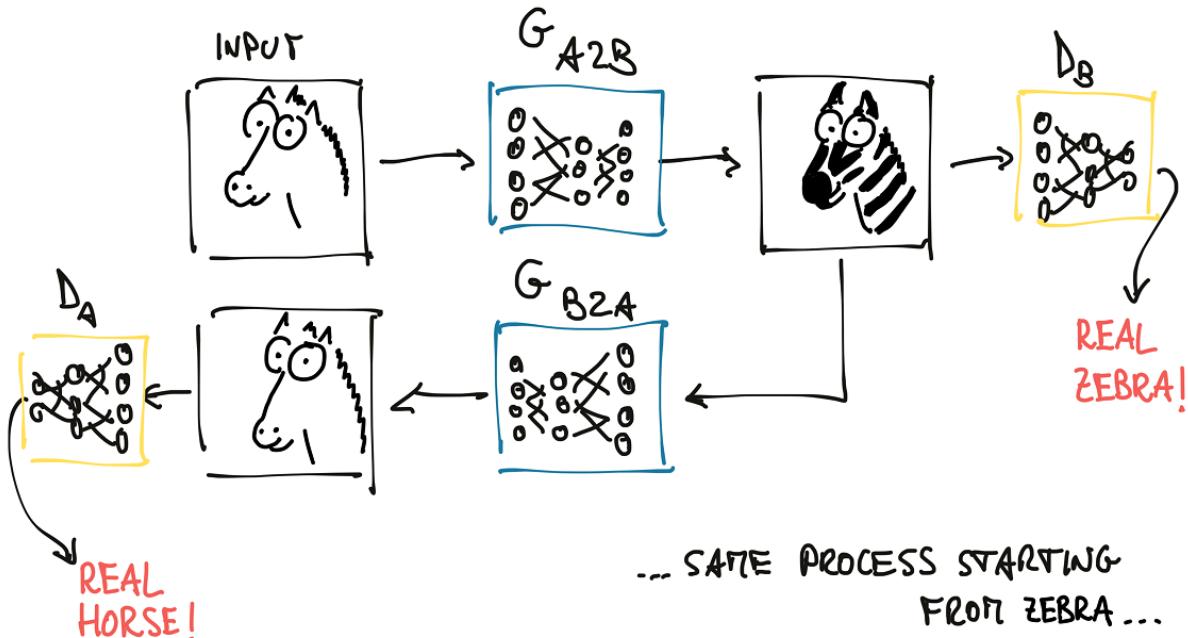


Figure 2.6 A CycleGAN trained to the point where it can fool both discriminator networks.

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

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

how to selectively change the appearance of objects in the scene without supervision on what's what. There's no signal indicating that manes are manes and legs are legs, but they get translated to something that lines up with the anatomy of the other animal.

2.2.3 A network that turns horses into zebras

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

Playing with a pre-trained CycleGAN will give us the opportunity to take a step closer and look at how a network, a generator in this case, is implemented. Let's do it right away: this is what a possible generator architecture for the horse to zebra task looks like. In our case it's our old friend ResNet. We'll define a `ResNetGenerator` class off-screen. The code is in the first cell of the `3_cyclegan.ipynb` file, but the implementation isn't relevant right now, and it's too complex to follow until we've gotten a lot more PyTorch experience. Right now, we're focused on *what* it can do, rather than *how* it does it. Let's instantiate the class with default parameters:

Listing 2.2 code/p1ch2/3_cyclegan.ipynb

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

The `netG` model has now been created, but it contains random weights. We mentioned earlier that we would run a generator model that had been pre-trained on the horse2zebra dataset. The weights of the model have been saved in a `pth` file, which is nothing but a `pickle` file of the tensor parameters of the model. We can load those into our `ResNetGenerator` using the `load_state_dict` method of the model:

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

At this point `netG` has acquired all the knowledge it had achieved during training. Note that this is fully equivalent to what happened when we loaded the `ResNet101` from `torchvision` in 2.1.1; but the `torchvision.resnet101` function hid the loading from us.

Let's put the network in `eval` mode, as we did for `ResNet101`:

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

```
netG.eval()

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

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

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

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

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

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

Let's open a horse file

```
# In[7]:
img = Image.open("../data/plch2/horse.jpg")
img
```



Figure 2.7 A man riding a horse. A horse not having it.

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

```
# In[8]:  
img_t = preprocess(img)  
batch_t = torch.unsqueeze(img_t, 0)
```

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

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

`batch_out` is now the output of the generator, that we can convert back to an image

```
# In[10]:  
out_t = (batch_out.data.squeeze() + 1.0) / 2.0  
out_img = transforms.ToPILImage()(out_t)  
# out_img.save('../data/plch2/zebra.jpg')  
out_img  
  
# Out[10]:  
<PIL.Image image mode=RGB size=316x256 at 0x23B24634F98>
```



Figure 2.8 A man riding a zebra. A zebra not having it.

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

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

On a serious note, it's hard to overstate the implications of this kind of work. Tools like the one we just downloaded are only to get higher quality and more ubiquitous. Face-swapping technology in particular has gotten considerable media attention. Searching for "deep fakes" will turn up a plethora of example content²⁰ (though we must note that there is a non-trivial amount of not safe for work content labeled as such; as with everything on the internet, click carefully).

So far we've had a chance to play with a model that sees into images and a model that generates new images. We'll end our tour with a model that involves one more, fundamental ingredient: natural language.

2.3 A pre-trained network that describes scenes

In order to get first-hand experience with a model involving natural language, we will now use a pre-trained image captioning model, generously provided by Ruotian Luo²¹. It is an implementation of the NeuralTalk2 model by Andrej Karpathy. What this kind of model does when presented with a natural image is generate a caption in English describing the scene, as shown in Figure-2.9. The model is trained on a large dataset of images along with a paired sentence description, e.g. "A Tabby cat is leaning on a wooden table, with one paw on a laser mouse and the other on a black laptop"²².

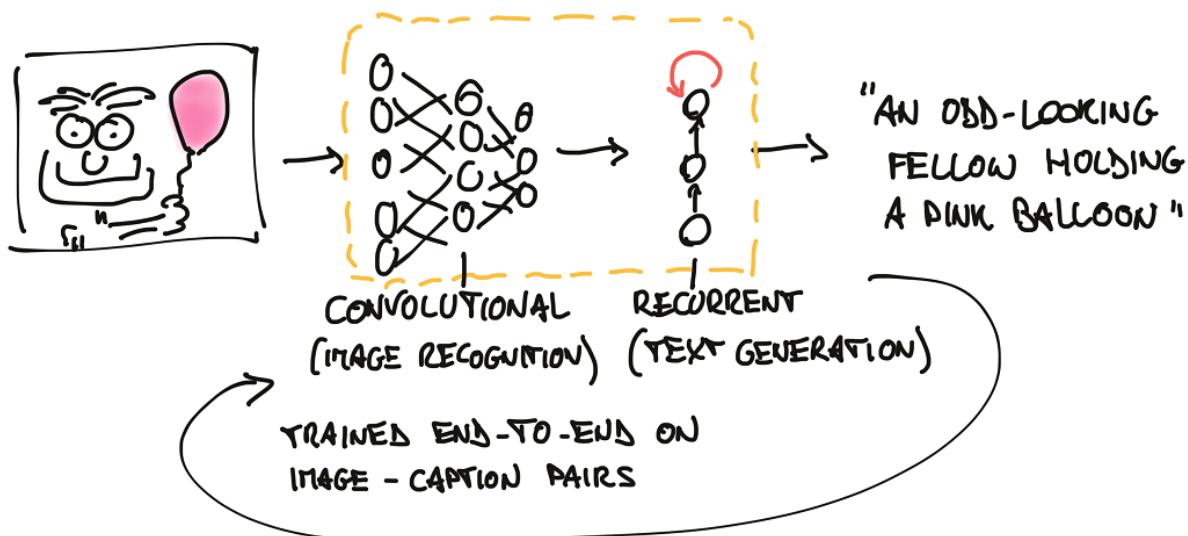


Figure 2.9 Concept of a captioning model.

This captioning model has two connected halves. The first half of the model is a network that learns to generate "descriptive" numerical representations of the scene (Tabby cat, laser mouse, paw), which are then taken as input to the second half. That second half is a so-called *recurrent neural network* which generates a coherent sentence by putting those numerical descriptions together. The two halves of the model are trained together on image-caption pairs.

The second half is called *recurrent* because it generates its outputs (individual words) in subsequent forward passes, where the input to each forward pass includes the outputs of the previous forward pass. This generates a dependency of the next word on words that have been generated earlier, as one would expect when dealing with sentences, or more in general with sequences.

2.3.1 NeuralTalk2

Back to the NeuralTalk2 model, we can find it at github.com/deep-learning-with-pytorch/ImageCaptioning.pytorch. We can just place a set of images in the data directory and run the following script

```
python eval.py --model ./data/FC/fc-model.pth --infos_path ./data/FC/fc-infos.pkl --image_folder ./data
```

Let's try with our `horse.jpg` image. It says "A person riding a horse on a beach". Quite appropriate.

Now, just for fun, let's see if our CycleGAN can also fool this NeuralTalk2 model. Let's add the `zebra.jpg` image in the data folder and rerun the model: "A group of zebras are standing in a field". Well, it got the animal right, but it saw more than one of them in the image. For sure this is not a pose that the network has ever seen in a zebra, nor it ever saw a rider on top of a zebra (with some spurious zebra patterns). In addition, it is very likely that zebras are depicted in groups in the training dataset, so there might be some bias there that one could investigate. The captioning network hasn't described the rider either. Again, it's probably for the same reason: the network hasn't been shown a rider on a zebra in the training dataset.

In any case, this is an impressive feat: we generated a fake image with an impossible situation and the captioning network was flexible enough to get the subject right.

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

The network architecture in this last case was in a way more complex than the ones we have seen earlier, as it includes two networks one of which is recurrent, but it was built out of the same building blocks, all of which are provided by PyTorch.

At the time of printing, models such as these exist more as applied research or novelty projects, rather than something we'd see with a well-defined, concrete use. The results, while promising, just aren't good enough to use... Yet. With time (and additional training data), we should expect this class of models to be able to describe the world to people with vision impairment, transcribe scenes from a video, and other similar tasks.

2.4 Torch Hub

Pre-trained models have been published since the early days of deep learning, however, until PyTorch 1.0 there was no way to ensure users would have a uniform interface to get them. TorchVision was a good example of a clean interface, as we have seen earlier in this Chapter, but other authors, as we have seen for CycleGAN and NeuralTalk2, chose different designs.

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

easy as it is loading a TorchVision model.

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

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

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

def another_entry_fn(*args, **kwargs):
    model = build_another_model(*args, **kwargs)
    return model
```

- ① an optional list of modules the code depends on;
- ② one or more functions to be exposed to users as the entry points for the repository; these functions are supposed to initialize models according to the arguments and return them.

In our quest for interesting pre-trained models, we can now search for GitHub repositories that include a `hubconf.py` and we'll know right away that we can load them using the `torch.hub` module. Let's see how this is done in practice. To do that, we'll go back to TorchVision, as it provides a clean example on how to interact with TorchHub.

First off, let's visit github.com/pytorch/vision and notice that there's a `hubconf.py` file in there. Great, that checks. The first thing to do is looking into that file to see what are the entry points for the repo - we'll need to specify them later. In case of TorchVision there are two: `resnet18` and `resnet50`. We already know what these do: return an 18-layer and 50-layer ResNet model, respectively. We also see that the entry point functions include a `pretrained` keyword argument. If `True`, the returned models will be initialized with weights learned from ImageNet, as we have seen earlier in the Chapter.

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

```
import torch
from torch import hub

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

- ① name and branch of GitHub repo
- ② name of entry point function
- ③ keyword argument

This manages to download a snapshot of the `master` branch of the `pytorch/vision` repo, along with the weights, to a local directory (defaults to `.torch/hub` in our home directory) and run the `resnet18` entry point function, which returns the instantiated model. Depending on the environment, Python could complain that there's a module missing, like `PIL`. TorchHub won't install missing dependencies, but it will report them to us so that we can take action.

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

Note that entry points are supposed to return models, but strictly speaking they are not forced to. For instance, we could have an entry point for transforming inputs and another one for turning the output probabilities into a text label. Or we could have an entry point for just the model, and another that includes the model along with the pre and post processing steps. By leaving these options open, the PyTorch developers have provided the community with just enough standardization and a lot of flexibility. We'll see what patterns will emerge from this opportunity.

TorchHub is quite new at the time of writing, and there are only a few models published this way. We can get at them by Googling "github.com hubconf.py". Hopefully the list will grow in the future, as more authors share their model through this channel.

2.5 Conclusion

This was hopefully a fun chapter, we took some time to play with models created with PyTorch, that were optimized to carry out specific tasks. In fact, the more enterprising of us could already put one of these models behind a web server and start a business, sharing the profits with the original authors!²³ Once we will learn how these models are built, we will also be able to use the knowledge we built here to download a pre-trained model and quickly fine-tune it on a slightly different task.

We will also see how building models that deal with different problems on different kinds of data will leverage on the same building blocks. One thing that PyTorch does particularly right is providing those building blocks under the form of a rather essential toolset - PyTorch is not a very large library from an API perspective, especially when compared to other deep learning frameworks.

This book will not be much about going through the complete PyTorch API or reviewing several deep learning architectures, but building hands-on knowledge of these building blocks. This way our readers will be able to consume the excellent online documentation and repositories on top of a solid foundation.

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

2.6 Exercises

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

2.7 Summary

- A pre-trained network is a model that has been already trained on a dataset. They can typically produce useful results immediately after loading the network parameters.
- By knowing how to use a pre-trained model, we can integrate a neural network into a project without having to design or train it.
- AlexNet and ResNet are two deep convolutional networks that set new benchmarks for image recognition in the years they were released.
- Generative Adversarial Networks have two parts, the generator and the discriminator, that work together to produce output indistinguishable from authentic items.
- CycleGAN uses an architecture that supports converting back and forth between two different classes of images.
- NeuralTalk2 uses a hybrid model architecture to consume an image and produce a text description of the image.
- Torch Hub is a standardized way to load models and weights from any project with an appropriate `hubconf.py` file.

It Starts with a Tensor

3

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

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

The process begins by converting our input into floating point numbers. We will cover converting image pixels to numbers, like we see in the first step of Figure-3.1, in chapter 4 (along with many other types of data). Since floating point numbers are the way a network deals with information, we need a way to encode real-world data of the kind we want to process into something digestible by a network and then decode the output back to something we can understand and use for a purpose.

The transformation from one form of data to another is typically learned by a deep neural network in stages, which means that the partially transformed data between each stage can be

thought of as a sequence of intermediate representations. For image recognition, early representations can be things like edge detection or certain textures like fur. Deeper representations can capture more complex structures like ears, noses, or eyes.

In general, such intermediate representations are collections of floating point numbers that characterize the input and capture the structure in the data, in a way that is instrumental for describing how inputs are mapped to the outputs of the neural network. Such characterization is specific to the task at hand and it is learned from relevant examples. These collections of floating point numbers and their manipulation is at the heart of modern AI - we will see several examples of this throughout the book. It's important to keep in mind that these intermediate representations (like those we see in the second step of Figure-3.1) are the results of combining the input with the weights of the previous layer of neurons. Each intermediate representation is unique to the inputs that proceeded it.

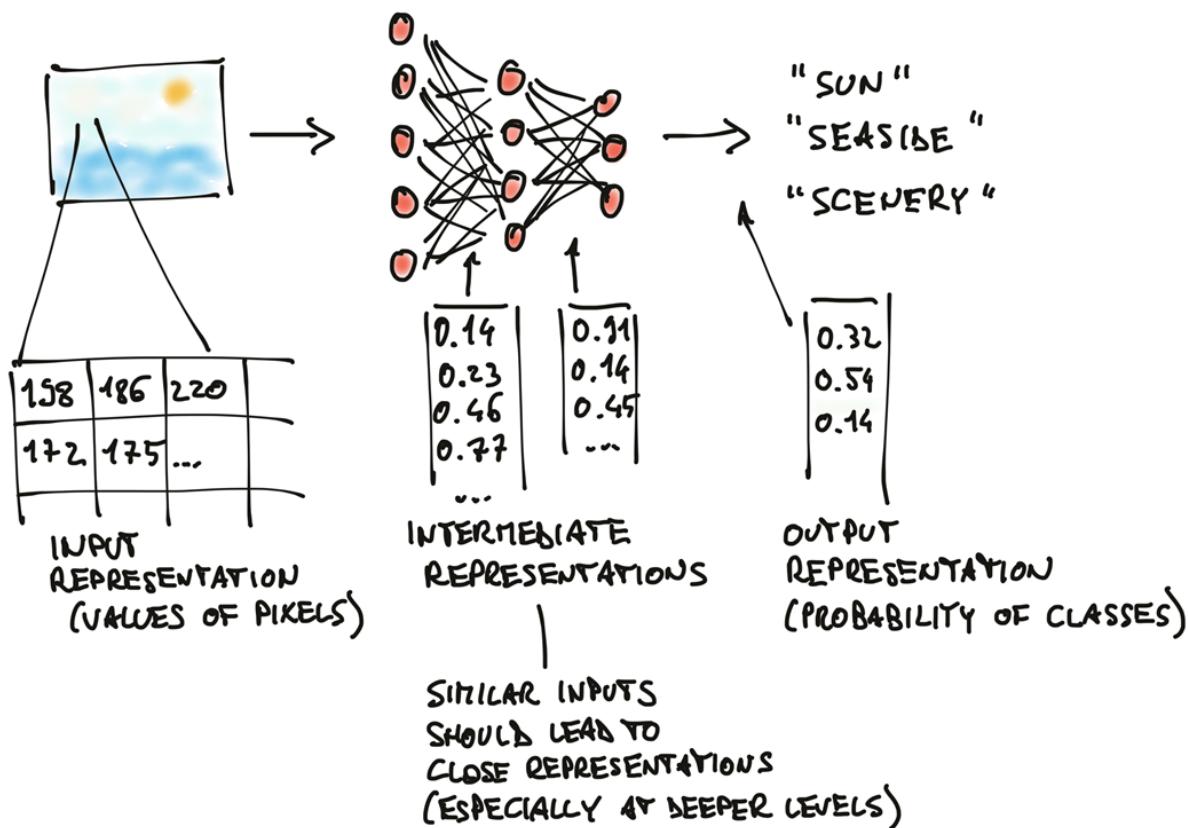


Figure 3.1 A deep neural network learns how to transform an input representation to an output representation (Note: number of neurons and outputs not to scale).

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

To this end, PyTorch introduces a fundamental data structure: the tensor. We have already bumped into tensors in Chapter 2, when we ran inference on pre-trained networks. For those who

come from mathematics, physics or engineering, the term *tensor* comes bundled with the notion of spaces, reference systems and transformations between them. For better or worse, those notions do not apply here. In the context of deep learning, tensors refer to the generalization of vectors and matrices to an arbitrary number of dimensions, as we can see in Figure-3.2. Another name for the same concept is *multidimensional arrays*. The dimensionality of a tensor coincides with the number of indexes used to refer to scalar values within the tensor.

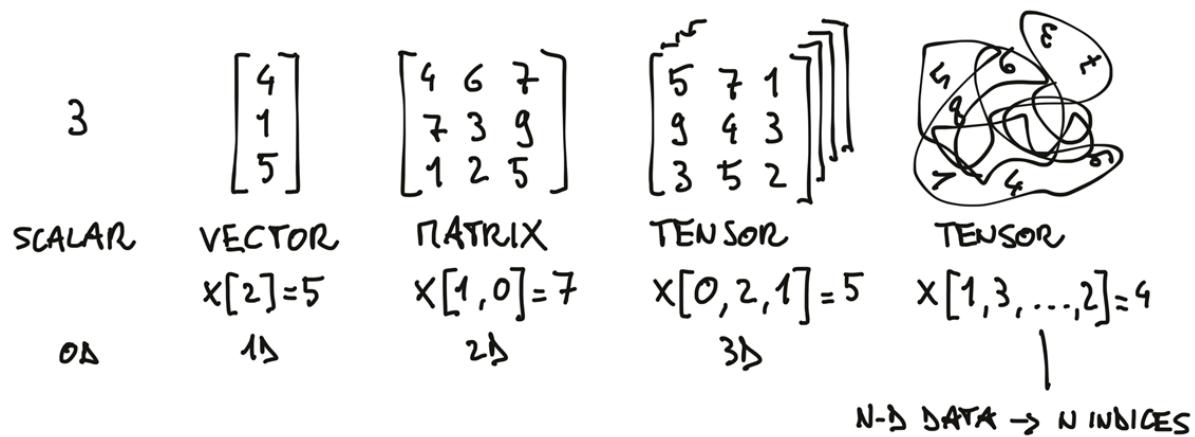


Figure 3.2 Tensors are the building blocks for representing data in PyTorch.

PyTorch is not the only library dealing with multidimensional arrays. NumPy is by far the most popular multidimensional array library, to the point that it has now arguably become the *lingua franca* of data science.

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 SciPy²⁴, Scikit-learn²⁵, and Pandas²⁶.

Compared to NumPy arrays, PyTorch tensors have a few superpowers, such as the ability to perform very fast operations on Graphical Processing Units (GPUs), to distribute operations on multiple devices or machines, or to keep track of the graph of computations that created them. These are all important features when implementing a modern deep learning library.

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

3.1 Tensors are multi-dimensional arrays

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

3.1.1 From Python lists to PyTorch tensors

Let's see list indexing in action so we can compare it to tensor indexing. Take a list of three numbers in Python:

Listing 3.1 code/p1ch3/1_tensors.ipynb

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

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

3.1.2 Constructing our first tensors

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

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

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

- ① Import the `torch` module
- ② Create a one-dimensional tensor of size 3 filled with ones

Let's see what we did here: after importing the `torch` module, we called a function that creates a (one-dimensional) tensor of size 3 filled with the value `1.0`. We can access an element using its 0-based index or assign a new value to it.

Although on the surface the previous example doesn't differ all that much from a list of number objects, under the hood things are completely different.

3.1.3 The essence of tensors

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

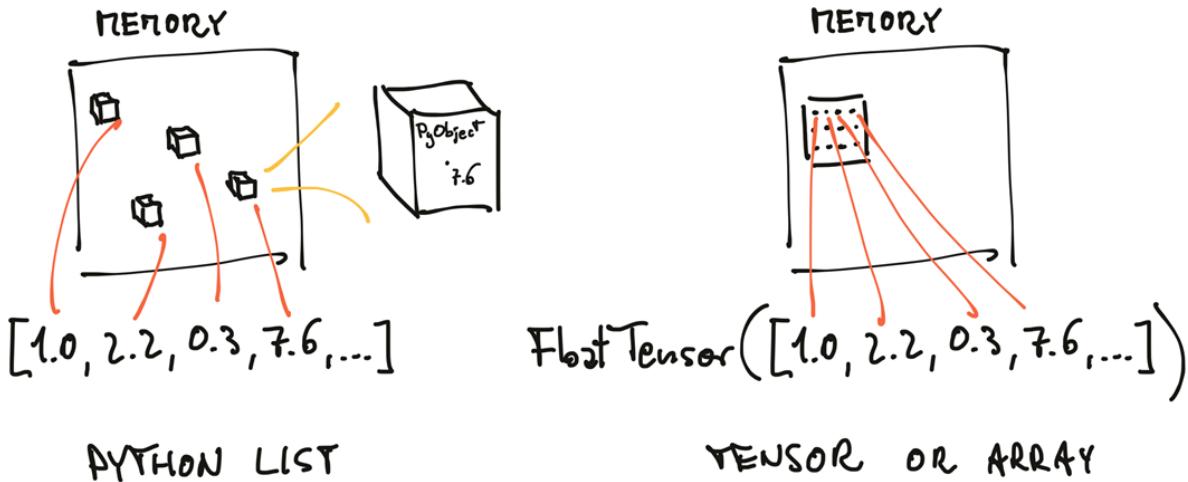


Figure 3.3 Python object (boxed) numeric values vs. tensor (unboxed array) numeric values.

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

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

- ❶ The use of `.zeros` here is just a way to get an appropriately sized array.
- ❷ We overwrite those zeros with the values we actually want.

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

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

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

To get the coordinates of the first point

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

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

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

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

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

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

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

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

which informs us on the size of the tensor along each dimension. We could also have used `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.]])
```

We can access an individual element in the tensor using two indices now, for instance

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

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



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

# Out[15]:
tensor(1.)
```

returns the y-coordinate of the 0-th point in our dataset. We can also access the first element in the tensor as we did before to get the 2D coordinates of the first point:

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

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

The output is another tensor that presents a different *view* of the same underlying data. The new

tensor is a 1D tensor of size 2 referencing the values of the first row in the `points` tensor. Does it mean that a new chunk of memory was allocated, values were copied into it, and the new memory returned wrapped in a new tensor object? No, because that would be very inefficient, especially if we had millions of points. We'll revisit how tensors are stored later in this chapter when we cover views of tensors in 3.6.

3.2 Indexing Tensors

What if we need to obtain a tensor containing all points but the first? That's easy using range indexing notation, the same that applies to standard Python lists, which we quickly recall:

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

- ① all elements in the list
- ② from element 1 inclusive to element 4 exclusive
- ③ from element 1 inclusive to the end of the list
- ④ from the start of the list to element 4 exclusive
- ⑤ from the start of the list to one before the last element
- ⑥ from element 1 inclusive to element 4 exclusive in steps of 2

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

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

- ① All rows after first, implicitly all columns
- ② All rows after first, all columns
- ③ All rows after first, first column
- ④ Add dimension of size one, just like `unsqueeze`

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

3.3 Named Tensors

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

To make things concrete, imagine that we have a 3D Tensor like `img_t` from 2.1.4 (we will use dummy data for simplicity here) and want to convert it to grayscale. We looked up typical weights for the colors to derive a single brightness value²⁷.

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

We also often want our code to generalize - for example from grayscale images represented as 2D Tensors with height and width dimensions to color images adding a third channel dimension (as in RGB) or from a single image to a batch of images. In 2.1.4 we had introduced an additional batch dimension in `batch_t`, here we pretend to have a batch of two.

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

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

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

But now we have the weight, too. PyTorch will allow us to multiply things that are of same shape, but also of shapes where one operand is of size one in a given dimensions. It also appends leading dimensions of size one automatically. This is a feature called broadcasting. We see that the our `batch_t` of shape (2, 3, 5, 5) gets multiplied with the `unsqueezed_weights` of shape (3, 1, 1) to a tensor of shape (2, 3, 5, 5), from which we can then sum the third dimension from the end (the 3 channels).

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

```
# Out[5]:  
(torch.Size([2, 3, 5, 5]), torch.Size([2, 3, 5, 5]), torch.Size([3, 1, 1]))
```

Because this gets messy quickly (and for efficiency), there even is a PyTorch function `einsum` (adapted from NumPy) that specifies an indexing mini-language²⁸ giving index names to dimensions for sums of such products. As often in Python, broadcasting — a form of summarizing unnamed things — is done using three dots '...', but don't worry too much about `einsum`, we will not use it in the following.

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

As we see, there is quite some bookkeeping involved. This is error prone, more so when the creation of tensors and their use are far apart in our code. This has caught the eye of practitioners and so it has been suggested²⁹ to give names to the dimension instead.

PyTorch 1.3 added *named tensors* as an experimental feature³⁰. Tensor factory functions such as `tensor` or `rand` take a `names` argument. The names should be a sequence of strings.

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

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

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

For operations with two inputs, in addition to the usual dimension checks, i.e. that sizes are either the same or one is 1 and can be broadcast to the other, PyTorch will now check the names for us. So far, it does not automatically align dimensions, so we need to do this explicitly. The method `align_as` returns a tensor with missing dimensions added and existing ones permuted to the right order.

```
# In[9]:
weights_aligned = weights_named.align_as(img_named)
weights_aligned.shape, weights_aligned.names

# Out[9]:
(torch.Size([3, 1, 1]), ('channels', 'rows', 'columns'))
```

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

```
# In[10]:
gray_named = (img_named * weights_aligned).sum('channels')
gray_named.shape, gray_named.names

# Out[10]:
(torch.Size([5, 5]), ('rows', 'columns'))
```

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

```
gray_named = (img_named[..., :3] * weights_named).sum('channels')

RuntimeError: Error when attempting to broadcast dims ['channels', 'rows', 'columns'] and dims
['channels']: dim 'columns' and dim 'channels' are at the same position from the
right but do not match.
```

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

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

# Out[12]:
(torch.Size([5, 5]), (None, None))
```

Given the experimental nature at the time of writing and that we do not muck around with indexing and alignment, we stick to unnamed in the remainder of the book. Named tensors have the potential of eliminating many sources of alignment errors which - if the PyTorch forum is any indication - can be a source of headaches. It will be interesting to see how widely the will be adopted.

3.4 Tensor element types

Alright, we have covered the basics on how tensors work, but we have not yet touched upon what kind of numeric types we can store in a `Tensor`. As we hinted at in 3.1.3, using the standard Python numeric types can be sub-optimal for several reasons:

- *Numbers in Python are full-fledged objects.* while a floating point number might only take, for instance, 32 bits to be represented on a computer, Python will convert them in a full-fledged Python object with reference counting, etc.. This operation, called *boxing*, is not a problem if we need to store a small number of them, but allocating millions of such numbers gets very inefficient;
- *Lists in Python are meant for sequential collections of objects.* there are no operations

defined for, say, efficiently taking the dot product of two vectors, or summing vectors together; also, Python lists have no way of optimizing the layout of their content in memory, as they are indexable collections of pointers to Python objects (of any kind, not just numbers); last, Python lists are one-dimensional, and while one can create lists of lists, this is again very inefficient;

- *The Python interpreter is slow compared to optimized, compiled code.* Performing mathematical operations on large collections of numerical data can be much faster using optimized code written in a compiled, low-level language like C.

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

3.4.1 Specifying the numeric type with `dtype`

The `dtype` argument to tensor constructors (that is, functions like `tensor`, `zeros`, `ones`) specifies the numerical data (d) type that will be contained in the tensor. The data type specifies the possible values the tensor can hold (integers vs. floating point numbers) and the number of bytes per value.³¹ 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
- `torch.bool`: boolean

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

3.4.2 A `dtype` for every occasion

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

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

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

Last, predicates on tensors, such as `points > 1.0`, produce `bool` tensors, indicating whether each individual element satisfies the condition or not. These are the numeric types in a nutshell.

3.4.3 Managing a tensor's `dtype` attribute

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

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

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

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

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

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

or the more convenient `to` method:

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

Under the hood, `to` checks and performs the conversion if needed. The `dtype`-named casting methods like `float` are shorthands for `to`, but the `to` method can take additional arguments that we'll discuss in 3.9.

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

```
# In[51]:  
points_64 = torch.rand(5, dtype=torch.double)      ①  
points_short = points_64.to(torch.short)  
points_64 * points_short # works from PyTorch 1.3 onwards  
  
# Out[51]:  
tensor([0., 0., 0., 0., 0.], dtype=torch.float64)
```

- ❶ `rand` initializes the tensor elements to random numbers between 0 and 1.

3.5 The tensor API

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

First off, 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. For instance, the `transpose` function we've encountered earlier can be used from the `torch` module

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

a.shape, a_t.shape

# Out[71]:
(torch.Size([3, 2]), torch.Size([2, 3]))
```

or as a method of the `a` tensor

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

a.shape, a_t.shape

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

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

We mentioned the online docs³² earlier. They are exhaustive and well organized, with the tensor operations divided into groups:

- *Creation ops* — functions for constructing a tensor, like `ones` and `from_numpy`
- *Indexing, slicing, joining, mutating ops* — functions for changing the shape, stride or content a tensor, like `transpose`
- *Math ops* — functions for manipulating the content of the tensor through computations
 - *Pointwise ops* — functions for obtaining a new tensor by applying a function to each element independently, like `abs` and `cos`
 - *Reduction ops* — functions for computing aggregate values by iterating through tensors, like `mean`, `std` and `norm`
 - *Comparison ops* — functions for evaluating numerical predicates over tensors, like `equal` and `max`
 - *Spectral ops* — functions for transforming in and operating in the frequency domain, like `stft` and `hamming_window`

- *Other operations* — special functions operating on vectors, like `cross`, or matrices, like `trace`
- *BLAS and LAPACK operations* — functions following the BLAS (Basic Linear Algebra Subprograms) specification for scalar, vector-vector, matrix-vector and matrix-matrix operations
- *Random sampling* — functions for generating values by drawing randomly from probability distributions, like `randn` and `normal`
- *Serialization* — functions for saving and loading tensors, like `load` and `save`
- *Parallelism* — functions for controlling the number of threads for parallel CPU execution, like `set_num_threads`

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

3.6 Tensors — scenic views on storage

It is time for us to look a bit closer at the implementation under the hood. Values in Tensors are allocated in contiguous chunks of memory, managed by `torch.Storage` instances. A storage is a one-dimensional array of numerical data, i.e. a contiguous block of memory containing numbers of a given type, such as `float`, 32-bits representing a floating point number, or `int64`, 64-bits representing an integer. A PyTorch Tensor is a view over such a `Storage` that is capable of indexing into that storage using an offset and and per-dimension strides.³³

Multiple tensors can index the same storage, even if they index into the data differently. We can see an example of this in Figure-3.4. In fact, when we requested `points[0]` in 3.2, what we got back is another tensor that indexes the same storage as the `points` tensor, just not all of it and with different dimensionality (1D vs 2D). The underlying memory is allocated only once, however, so creating alternate tensor-views on the data can be done quickly, no matter the size of the data managed by the `Storage` instance.

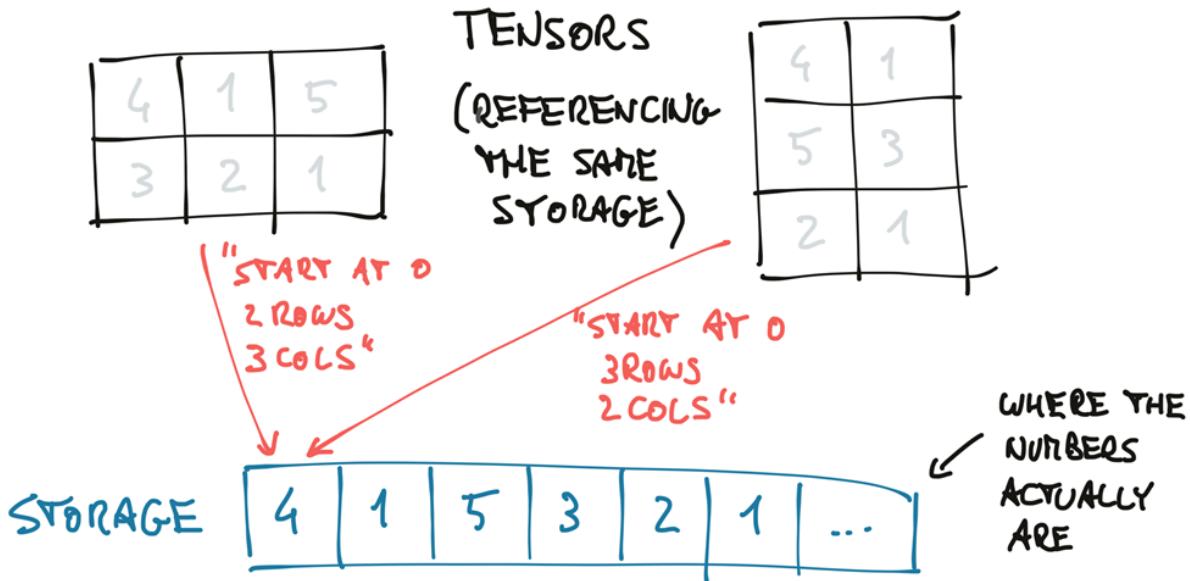


Figure 3.4 Tensors are views over a Storage instance.

3.6.1 Indexing into storage

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

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

# Out[17]:
4.0
1.0
5.0
3.0
2.0
1.0
[torch.FloatTensor of size 6]
```

Even though the tensor reports itself as having 3 rows and 2 columns, the storage under the hood is a contiguous array of size 6. In this sense, the tensor just knows how to translate a pair of indices into a location in the storage.

We can also index into a storage manually, for instance:

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

# Out[18]:
4.0

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

# Out[19]:
1.0
```

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

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

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

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

3.6.2 Modifying Stored Values — Inplace Operations

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

```
# In[73]:
a = torch.ones(3, 2)

# In[74]:
a.zero_()
a

# Out[74]:
tensor([[0., 0.],
        [0., 0.],
        [0., 0.]])
```

3.7 Tensor metadata: size, offset, stride

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

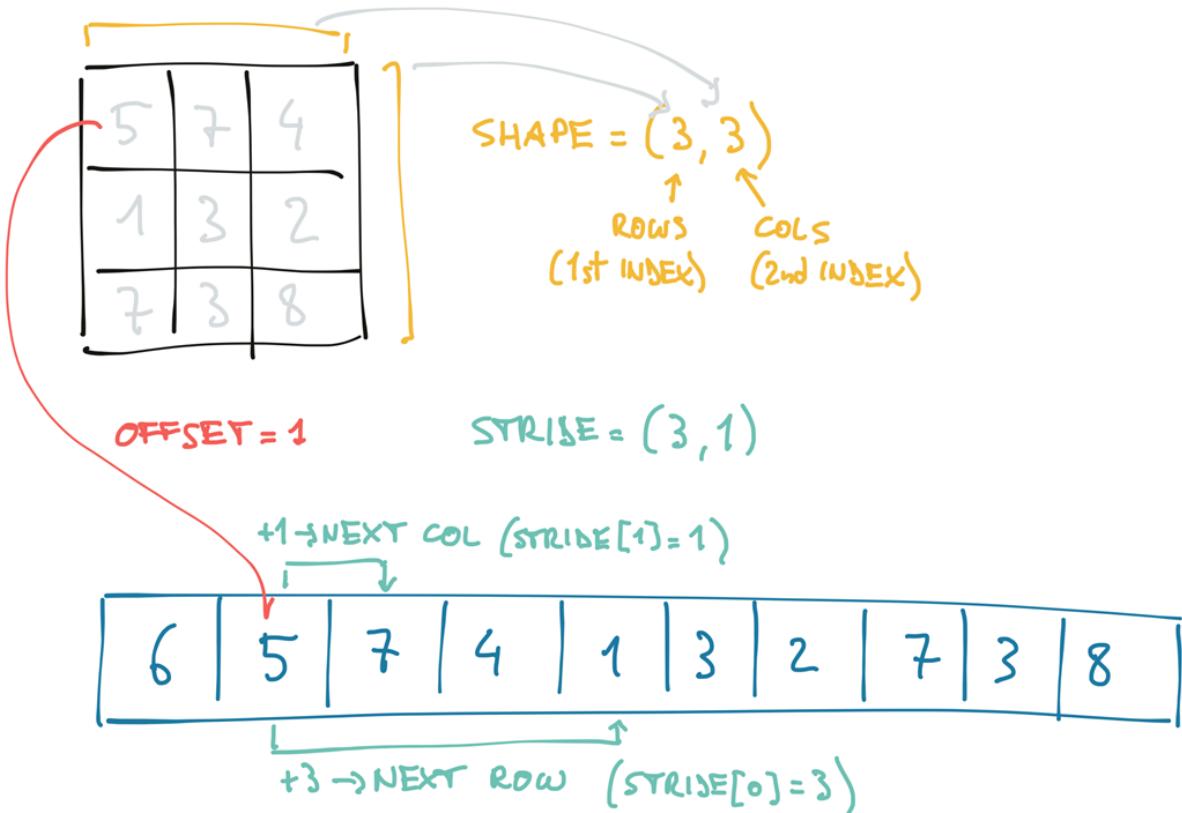


Figure 3.5 Relationship between a tensor's offset, size and stride. Here the tensor is a view over a larger storage, like one that might have been allocated when creating a larger tensor.

3.7.1 Views over another tensor's storage

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

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

# Out[21]:
2

# In[22]:
second_point.size()

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

The resulting tensor has offset 2 in the storage (since we need to skip the first point, which has two items) and the size is an instance of the `Size` class containing one element, since the tensor is one-dimensional. Important note: this is the same information as 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. For instance, our `points` tensor has a stride of `(2, 1)`:

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

Accessing an element i, j in a 2D tensor, results in accessing the `storage_offset + stride[0] * i + stride[1] * j` element in the storage. The offset will usually be zero; if this tensor is a view 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, like transposing a tensor or extracting a sub-tensor, to be inexpensive, as they do not lead to memory reallocations; instead they consist in allocating a new tensor object with a different value for size, storage offset or stride.

We've already seen extracting a sub-tensor when we indexed a specific point and saw the storage offset increasing. Let's see what happens to size and stride as well:

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

Bottomline, the sub-tensor has one fewer dimension, as one would expect, while still indexing the same storage as the original `points` tensor. This also means that changing the sub-tensor will have a side-effect on the original tensor, too

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

```
# Out[28]:  
tensor([[ 4.,  1.],  
       [10.,  3.],  
       [ 2.,  1.]])
```

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

```
# In[29]:  
points = torch.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])  
second_point = points[1].clone()  
second_point[0] = 10.0  
points  
  
# Out[29]:  
tensor([[4., 1.],  
       [5., 3.],  
       [2., 1.]])
```

3.7.2 Transposing without copying

Let's try with transposing now. Let's take our `points` tensor, that 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. We take this opportunity to introduce the `t` function, a short-hand alternative to `transpose` for 2-dimensional tensors.

TIP

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

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

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

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

and that they differ only in the shape and stride

```
# In[33]:
```

```

points.stride()

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

# Out[34]:
(1, 2)

```

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

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

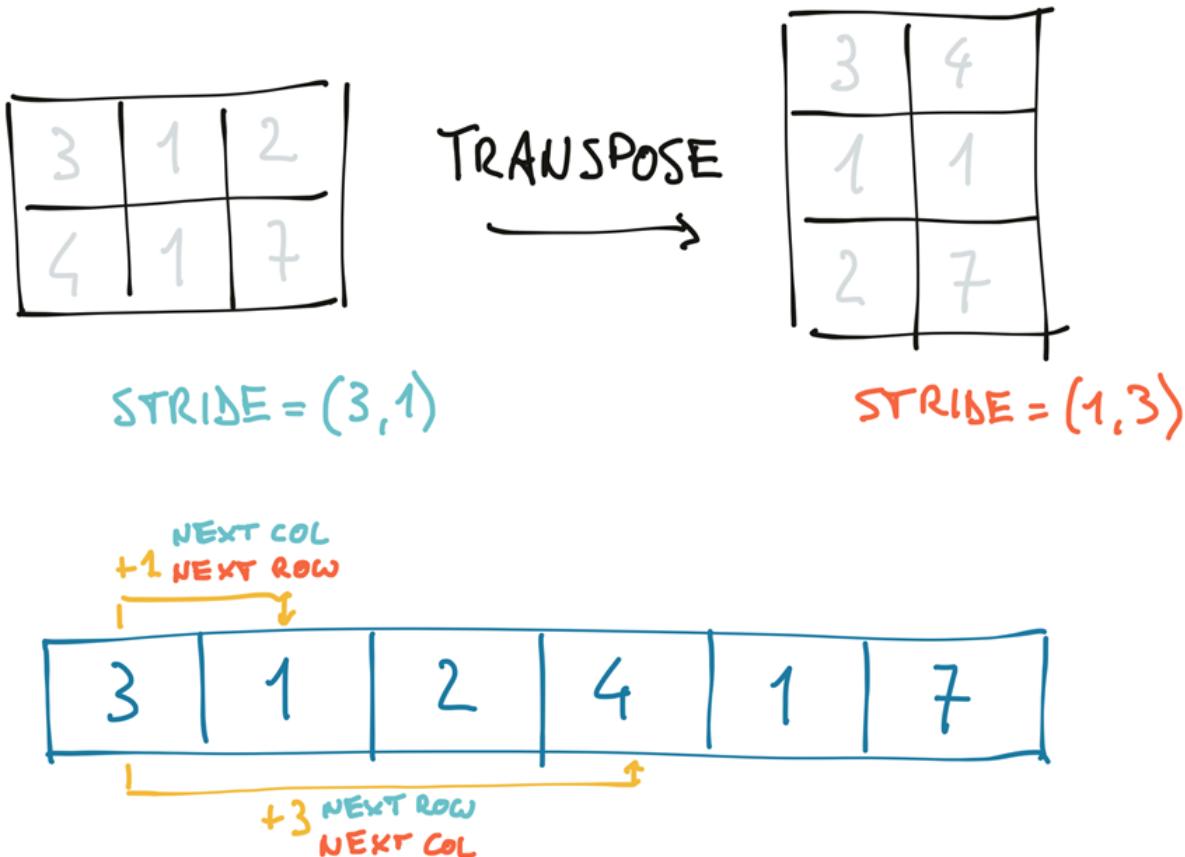


Figure 3.6 Transpose operation applied to a tensor.

3.7.3 Transposing in higher dimensions

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

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

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

# In[36]:
transpose_t.shape

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

# In[37]:
some_t.stride()

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

# In[38]:
transpose_t.stride()

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

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

3.7.4 Contiguous tensors

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

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

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

# Out[39]:
True

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

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

We can obtain a new contiguous tensor from a non-contiguous one using the `contiguous` method. The content of the tensor will be the same, but the stride will change, as will the storage. As a refresher, here's our diagram again. Hopefully it will all make sense now that we've taken a good peek at how tensors are built.

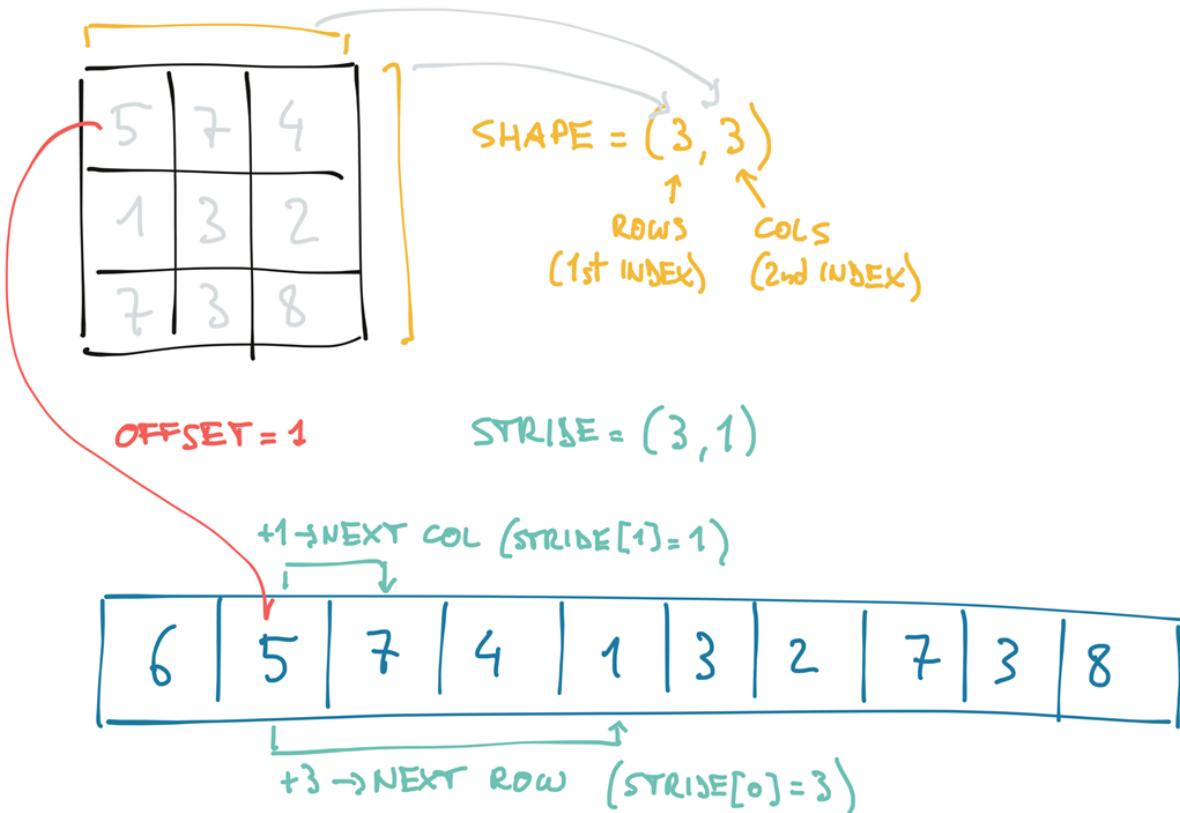


Figure 3.7 Relationship between a tensor's offset, size and stride. Here the tensor is a view over a larger storage, like one that might have been allocated when creating a larger tensor.

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

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

# In[42]:
points_t.storage()

# Out[42]:
4.0
1.0
5.0
3.0
2.0
1.0
```

```
[torch.FloatTensor of size 6]

# In[43]:
points_t.stride()

# Out[43]:
(1, 2)

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

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

# In[45]:
points_t_cont.stride()

# Out[45]:
(3, 1)

# In[46]:
points_t_cont.storage()

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

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

3.8 NumPy interoperability

We mentioned NumPy here and there. While we do not consider NumPy a prerequisite for reading this book, we strongly encourage the reader 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 very efficiently. By doing so, we 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 working with the Python buffer protocol³⁴.

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

```
# In[55]:
points = torch.ones(3, 4)
points_np = points.numpy()
points_np

# Out[55]:
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]], dtype=float32)
```

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

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

Vice-versa, we can obtain a PyTorch tensor from a NumPy array this way:

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

which will use the same buffer sharing strategy as just described.

While the default numeric type in PyTorch is 32 bit floating point, for the one for `numpy` it is 64 bit. As discussed in [Numeric types](#) we usually want to use 32 bit floating points so we want to make sure that we have tensors of `dtype torch.float` after converting.

3.9 Moving tensors to the GPU

One last point we are going to cover about PyTorch tensor implementations is related to computing on the GPU. Every Torch tensor can be transferred to (one of) the GPU(s) in order to perform massively parallel, fast computations. All operations that will be performed on the tensor will be carried out using GPU-specific routines that come with PyTorch.

NOTE

As of mid 2019, main PyTorch releases only have acceleration on GPUs that have support for CUDA. PyTorch can run on AMD's ROCm ³⁵ and the master repository provides support, but so far you need to compile it yourself. ³⁶ Support for Google's TPUs is a work in progress ³⁷, with the current proof of concept available to the public in Google Colab ³⁸. Implementation of data structures and kernels on other GPU technology, such as OpenCL, are not planned at the time of the writing of this chapter.

3.9.1 Managing a tensor's device attribute

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

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

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

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

Doing so returns a new tensor that has the same numerical data, but stored in the RAM of the GPU, rather than in regular system RAM. Now that the data is stored locally on the GPU, we'll start to see the speedups mentioned earlier when performing mathematical operations on the tensor. In almost all cases, CPU- and GPU-based tensors expose the same user-facing API, making it much easier to write code that is agnostic to where, exactly, the heavy number crunching is running.

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

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

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

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

- ① Multiplication performed on the CPU.
- ② Multiplication performed on the GPU.

Note that the `points_gpu` tensor is not brought back to the CPU once the result has been computed. What happened in the line above is that

1) the `points` tensor has been copied to the GPU; 2) a new tensor has been allocated on the GPU and used to store the result of the multiplication; 3) a handle to that GPU tensor is returned.

Therefore, if we also add a constant to the result

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

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

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

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

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

- ① Defaults to GPU index 0.

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

3.10 Generalized Tensors are Tensors, too

For the purposes of this book and for the vast majority of applications in general, Tensors are multi-dimensional arrays just as we've seen in this chapter. If we risk a peek under the hood of PyTorch, there is a twist: how the data is stored under the hood is separate from the Tensor API of 3.5. Any implementation that meets the contract of that API can be considered a Tensor!

PyTorch will cause the right computation functions to be called no matter whether our Tensor is on the CPU or the GPU. This is accomplished through a *dispatching* mechanism, and that mechanism can cater to other Tensor types by hooking up the user facing API to the right backend functions. Sure enough, there are other kinds of Tensors: some are specific to certain classes of hardware device (like Google TPUs) and others have different data representation strategies from the dense array-style we've seen so far. For example, sparse tensors will store only non-zero entries, along with index information. The PyTorch dispatcher on the left of 3.8 is designed to be extensible, with the subsequent switching done to accommodate the various numeric types of Numeric types shown on the right being a fixed aspect of the implementation coded into each backend.

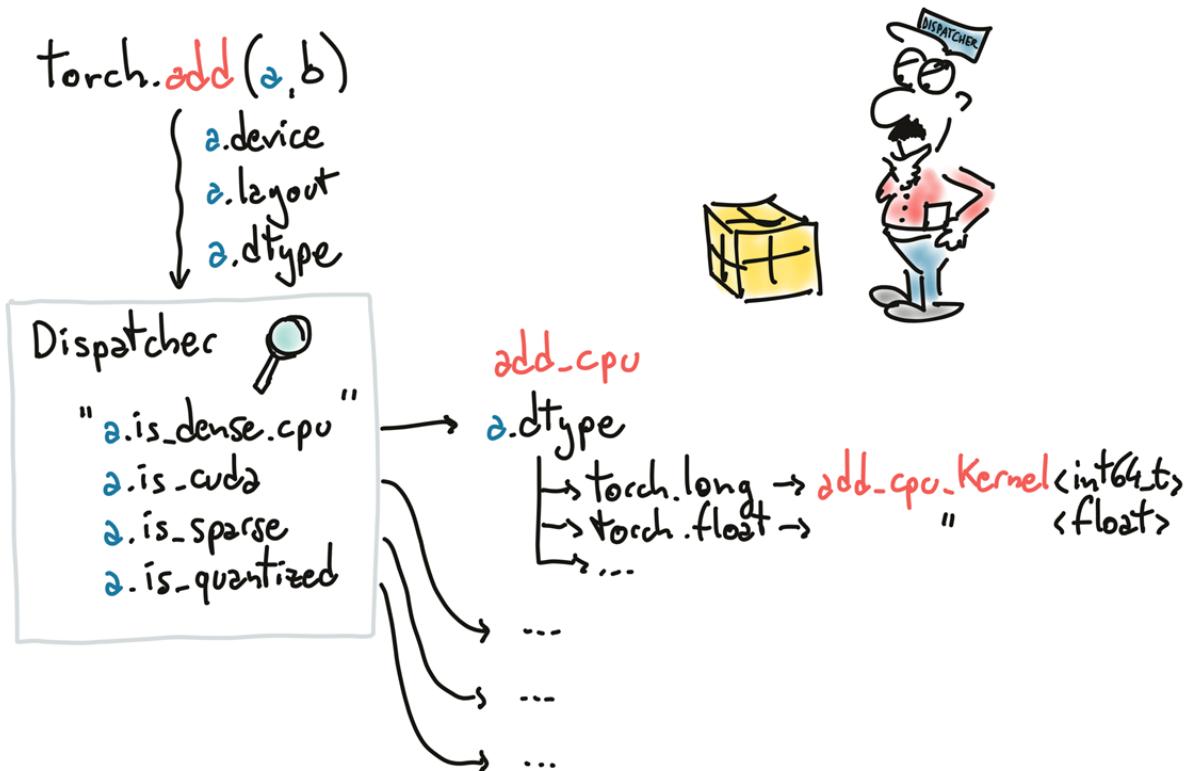


Figure 3.8 The dispatcher in PyTorch is one of its key infrastructure bits.

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

As with many things, the number of Tensor kinds has grown as PyTorch supported a broader range of hardware and applications, and we may expect new kinds to arise as people explore new ways to express and do computation with PyTorch.

3.11 Serializing tensors

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

```
# In[57]:  
torch.save(points, '../data/plch3/ourpoints.t')
```

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

```
# In[58]:  
with open('../data/plch3/ourpoints.t', 'wb') as f:  
    torch.save(points, f)
```

Loading our points back is similarly a one-liner:

```
# In[59]:  
points = torch.load('../data/plch3/ourpoints.t')
```

or equivalently

```
# In[60]:  
with open('../data/plch3/ourpoints.t','rb') as f:  
    points = torch.load(f)
```

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

3.11.1 Serializing to HDF5 with h5py

For those cases when you need to, however, you can use the HDF5 format and library ³⁹. HDF5 is a portable and widely supported format for representing serialized multidimensional arrays, organized in a nested key-value dictionary. Python supports HDF5 through the `h5py` library ⁴⁰, which accepts and returns data under the form of NumPy arrays.

We can install `h5py` using

```
$ conda install h5py
```

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

```
# In[61]:  
import h5py  
  
f = h5py.File('../data/plch3/ourpoints.hdf5', 'w')  
dset = f.create_dataset('coords', data=points.numpy())  
f.close()
```

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

```
# In[62]:  
f = h5py.File('../data/plch3/ourpoints.hdf5', 'r')  
dset = f['coords']  
last_points = dset[-2:]
```

What happened here is that data has not been loaded when the file was opened or the dataset was required. Rather, data stayed on disk until we requested the second and last rows in the dataset. At that point, `h5py` has accessed those two columns and returned a NumPy array-like object encapsulating that region in that dataset that behaves like a NumPy array and has the same API.

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

```
# In[63]:  
last_points = torch.from_numpy(dset[-2:])  
f.close()  
  
>>> last_points = torch.from_numpy(dset[1:])
```

Once we're finished loading data, we close the file.

3.12 Conclusion

Now we have covered everything we need to get started. There are other aspects related to tensors, such as creating views of tensors, indexing tensors with other tensors, or broadcasting, which simplifies performing element-wise operations between tensors of different size or shape. We will cover them as needed along the way. At this point, we can move on to the next chapter where we will learn how to represent real-world data in PyTorch. We will start with simple tabular data and move on to something more elaborate. In the process, we will get to know more about tensors.

3.13 Exercises

- Create a tensor `a` from `list(range(9))`. Predict then check what the size, offset, and strides are.
 - Create a new tensor using `b = a.view(3, 3)`. What does `view` do? Check that `a` and `b` share the same storage.
 - 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?
 - Apply the function element-wise to `a`. Why does it return an error?
 - What is the operation required to make the function work?
 - Is there a version of your function that operates in-place?

3.14 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; they are the basic data structure in PyTorch .
- PyTorch has a comprehensive standard library for tensor creation, manipulation and 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 (e.g. `Tensor.sqrt_`).

Real-World Data Representation Using Tensors



This chapter covers:

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

In the previous chapter, you learned that 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 (e.g. weights and biases) in a neural network are tensors. Having a good sense of how to perform operations on tensors and index them effectively is central to using tools like PyTorch successfully. Now that you know the basics of tensors, your dexterity with them will grow as you make your way through the book.

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

This is what we'll learn in this chapter. We'll cover different types of data with a focus on the types relevant to this book and showing how to get them represented as tensors. Then you'll learn how to load the data from the most common on-disk formats and also get a feeling for those data types structure so you can see how to prepare them for training a neural network. Often, our raw data won't be perfectly formed for the problem we'd like to solve, so we'll have a chance to practice our tensor manipulation skills on a few more interesting tensor operations.

We'll be using a lot of image and volumetric data through the rest of the book, since those are common data types and they reproduce well in book format. We also cover tabular data, time series, and text, as those will also be of interest to a number of our readers.

Each section in the chapter will describe a data type, and each will come with its own dataset. While we've structured the chapter to have each data type build on the previous, readers should feel free to skip around a bit if so inclined.

Since a picture is worth a thousand words, we start with image data. We then demonstrate working with a three-dimensional array using medical data that represents patient anatomy as a volume. Next, we work with tabular data of data about wines, just like what we'd find in a spreadsheet. After that, we move to *ordered* tabular data, with a time-series dataset from a bike-sharing program. Finally, dip our toes into text data from Jane Austen. Text data retains the ordered aspect, but introduces the problem of representing words as arrays of numbers. At every section, we will stop where a deep learning researcher would start: right before feeding the data to a model. We encourage the reader to keep these datasets around. They will constitute excellent material for when we start learning how to train neural network models in the next chapter.

4.1 Images

The introduction of convolutional neural networks revolutionized computer vision,⁴¹ and image-based systems have since acquired a whole new set of capabilities. Problems that required complex pipelines of highly tuned algorithmic building blocks were now solvable at unprecedented levels of performance by training end-to-end networks using paired input-and-desired-output examples. In order to participate in this revolution, we need to be able to load 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). One might have a single scalar per grid point (the pixel), which would be represented as a grayscale image, or multiple scalars per grid point, which would typically be representing different colors as you saw in the previous chapter, or different *features* like depth from a depth camera.

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

We mentioned colors earlier. There are several ways of encoding numbers into colors⁴². The most common is RGB, where a color is defined by three numbers representing the intensity of

red, green and blue. We can think of a color channel as a grayscale intensity map of only the color in question, similar to what you'd see if you looked at the scene in question using a pair of pure red sunglasses. Figure-4.4 shows a rainbow, where each of the RGB channels captures a certain portion of the spectrum (the figure is simplified, in that it elides things like the orange and yellow bands being represented as a combination of red and green).

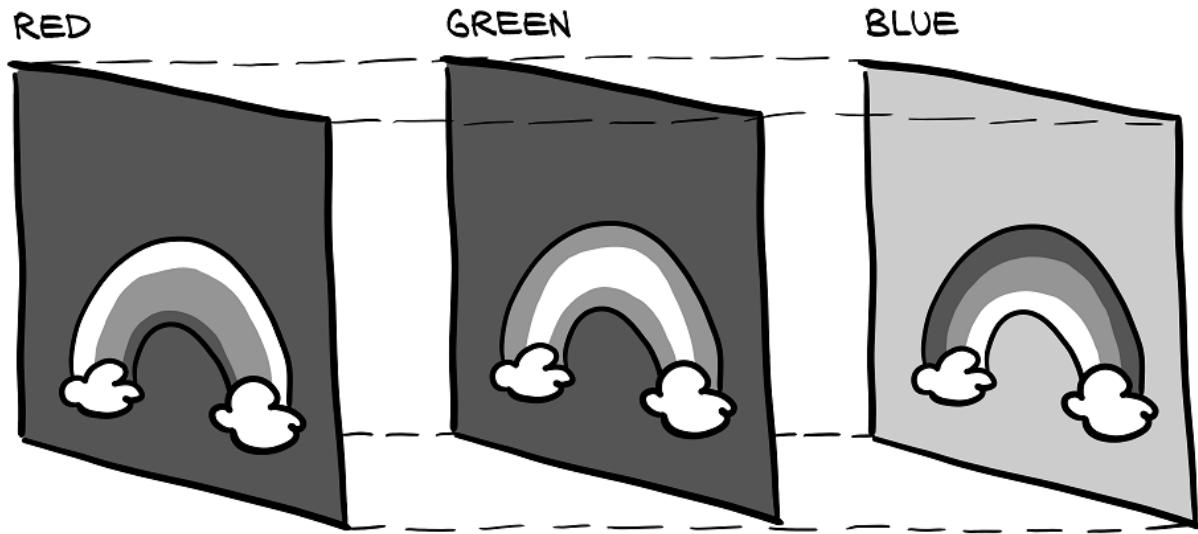


Figure 4.1 A rainbow, broken into red, green, and blue channels.

Images come in several different file formats, but luckily there are plenty of ways to load images in Python. Let's start loading a PNG image using the `imageio` module. We'll use `imageio` throughout the chapter as it handles different data types with a uniform API.⁴³

Let's load an image:

Listing 4.1 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)
```

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

We can use the tensor's `permute` method with the old dimensions for each new dimension to get to an appropriate layout. Given an input tensor `H x W x C` as obtained above, we get to a proper

layout by having channel number 2 first and then 0 and 1:

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

We've seen this previously, but note that the above operation does not make a copy of the tensor data. Instead, `out` uses the same underlying storage as `img` and only plays with the size and stride information at the tensor level. This is convenient because the operation is very cheap, but changing a pixel in `img` will lead to a change in `out` - just as a heads-up.

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

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

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

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

which indicates that our batch will consist of 100 RGB images of 256 pixels in height and 256 pixels in width. Notice the type of the tensor: we're expecting each color to be represented as a 8-bit integer, as in most photographic formats from standard consumer cameras. We can now load all `.png` images from an input directory and store them in the tensor:

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

- ① Here we keep only the first three channels. Sometimes images will have an alpha channel indicating transparency as a fourth channel, but our network only wants RGB input.

We mentioned earlier that neural networks usually work with floating point tensors as their

input. We have already mentioned it and we will also see it in upcoming chapters: neural networks exhibit the best training performance when input data ranges roughly from 0 to 1, or from -1 to 1 (this is an effect of how their building blocks are defined).

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

NOTE Here, we normalize just a single image because we do not know yet how to operate on an entire dataset. It is good practice to compute the mean and standard deviation on the entire training data in advance and then subtract and divide by these fixed pre-computed quantities. We saw the latter in the preprocessing for the image classifier in 2.1.4.

There are several other operations we can perform on inputs, for instance geometric transformations like rotations, scaling, or cropping. These may help with training or may be required to have an arbitrary input conform the input requirements of a network, such as the size of the image. We will stumble on quite a few of these strategies in 12.5. For now, just remember that you have image manipulation options available.

4.2 Volumetric Data

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

CTs only have a single intensity channel, similar to a grayscale image. This means that often in native data formats the channel dimension will be left out, so similar to the last section, the raw data typically has three dimensions. By stacking individual 2D slices into a 3D tensor we can build volumetric data representing the 3D anatomy of a subject. Unlike what we saw in Figure-4.4, the extra dimension in Figure-4.5 represents an offset in physical space, rather than a particular band of the visible spectrum.

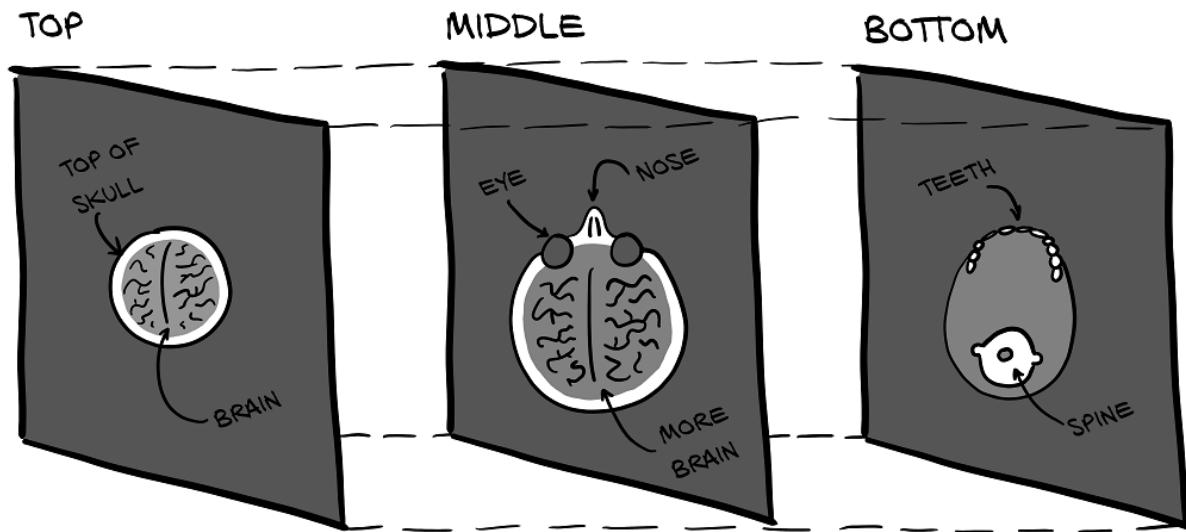


Figure 4.2 Slices of a CT scan, from the top of the head to the jawline.

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

Let's load a sample CT scan using the `volread` function in the `imageio` module, which takes a directory as argument and assembles all 'DICOM' (Digital Imaging Communication and Storage)

files⁴⁴ in a series in a NumPy 3D array.

Listing 4.2 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%)

(99, 512, 512)
```

Also in this case, the layout is different from what PyTorch expects, due to having no channel information. So we'll have to make room for the `channel` dimension using `unsqueeze`:

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

vol.shape

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

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

4.3 Tabular Data

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

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

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

PyTorch tensors, on the other hand, are homogeneous. Other data science packages, like Pandas, have the concept of *dataframe*, an object representing a dataset with named, heterogenous

columns. In contrast, information in PyTorch is typically encoded as a number, typically floating point (though integer types and boolean are supported as well). This numeric encoding is deliberate, since neural networks are mathematical entities that take real numbers as inputs and produce real numbers as output through successive application of matrix multiplications and non-linear functions.

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

There are a large number of tabular datasets freely available on the Internet, see for instance github.com/caesar0301/awesome-public-datasets.

Let's start with something fun: wine. The Wine Quality dataset is a freely available table containing chemical characterizations of samples of *vinho verde*, a wine from north Portugal, together with a sensory quality score. The dataset for white wines can be downloaded here: archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/winequality-white.csv. For convenience, we created a copy of the dataset on the Deep Learning with PyTorch Git repository, under `data/p1ch4/tabular-wine`.

The file contains a comma-separated collection of values organized in 12 columns preceded by a header line containing the column names. The first 11 columns contain values of chemical variables, while the last column contains the sensory quality score from 0 (very bad) to 10 (excellent). These are the column names in the order they appear in the dataset:

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

A possible machine learning task on this dataset is predicting the quality score from chemical characterization alone. Don't worry though, machine learning is not going to kill wine tasting anytime soon. We have to get the training data from somewhere! As we can see in Figure-4.1, we're hoping to find a relationship between one of the chemical columns in our data and the quality column. Here, we're expecting to see quality increase as sulfur decreases.

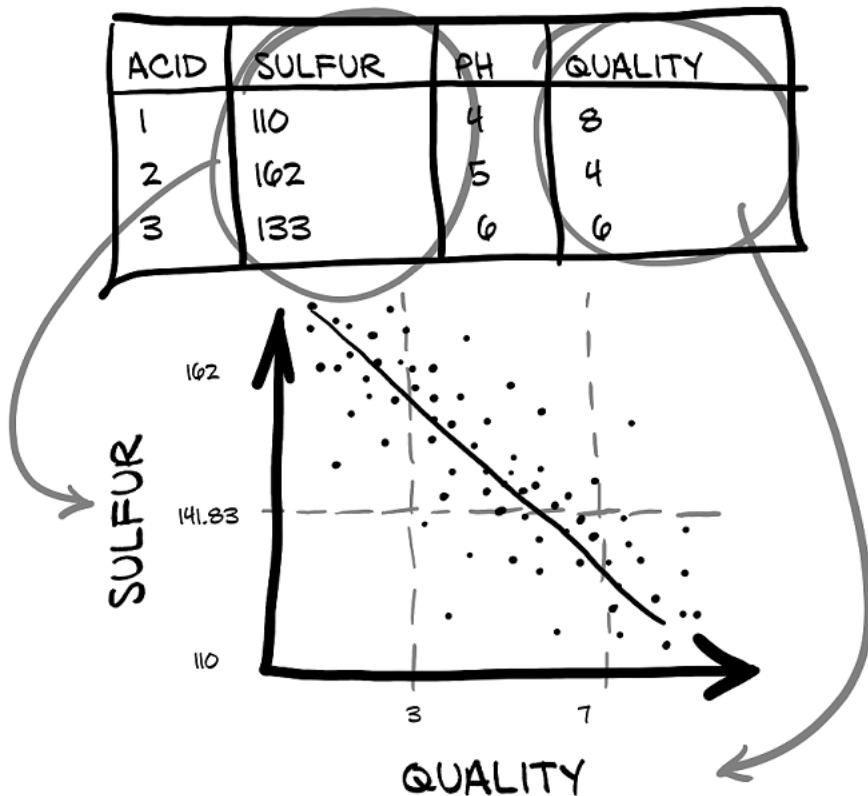


Figure 4.3 The (hopeful) relationship between sulfur and quality in wine.

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

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

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

Listing 4.3 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 we just prescribed what the type of the 2D array should be (32-bit floating point), the delimiter used to separate values in each row and the fact that the first line should not be read since it contains the column names. Let's check that all the data has been read:

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

wineq_numpy.shape, col_list

# Out[3]:
((4898, 12),
 ['fixed acidity',
 'volatile acidity',
 'citric acid',
 'residual sugar',
 'chlorides',
 'free sulfur dioxide',
 'total sulfur dioxide',
 'density',
 'pH',
 'sulphates',
 'alcohol',
 'quality'])
```

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

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

wineq.shape, wineq.dtype

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

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

SIDE BAR **Continuous, ordinal, and categorical values**

There are three different kinds of numerical values that we should be aware of as we attempt to make sense of our data.⁴⁵

The first kind is *continuous* values. These are the most intuitive when represented as numbers. They are strictly ordered, and a difference between various values has a strict meaning. Stating that package A is two kilograms heavier than package B, or that package B came from 100 miles further away than A has a fixed meaning, no matter if package A is three kilograms or ten, nor if B came from two hundred miles away, or two thousand. If you're counting or measuring something with units, it's probably a continuous value. The literature actually divides continuous values further: In the examples above, it makes sense to say something is twice as heavy or three times further away, so those values are said to be on a *ratio scale*. The time of day, on the other hand, does have the notion of difference but it is not reasonable to claim that 6 o'clock is twice as late as 3 o'clock, so it only offers an *interval scale*.

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

Finally, *categorical* values have neither ordering nor numerical meaning to their values. These are often just enumerations of possibilities assigned arbitrary numbers. Assigning water to one, coffee to two, soda to three, and milk to four is a good example. There's no real logic to placing water first and milk last; they simply need distinct values to differentiate them. We could assign coffee to ten and milk to negative three and there would be no significant change (though assigning values in the range $0..N-1$ will have advantages for one-hot encoding and the embeddings we discuss in 4.5). As the numerical values bear no meaning, they are said to be on a *nominal scale*.

We could treat the score as a continuous variable and 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 ways, we will typically remove the score from the tensor of input data and keep it in a separate tensor, so that we can use the score as the ground truth without it being input to our model.

```

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

# Out[5]:
(tensor([[ 7.00,  0.27,  ...,  0.45,  8.80],
       [ 6.30,  0.30,  ...,  0.49,  9.50],
       ...,
       [ 5.50,  0.29,  ...,  0.38, 12.80],
       [ 6.00,  0.21,  ...,  0.32, 11.80]]), torch.Size([4898, 11]))

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

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

```

- ① select all rows, all columns except the last
- ② select all rows, the last column

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

```

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

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

```

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

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

There's a marked difference between the two approaches. Keeping wine quality scores in an integer vector of scores induces an ordering on the scores - which might be totally appropriate in this case, since a score of 1 is lower than a score of 4. It also induces some sort of distance between scores, i.e. the distance between 1 and 3 is the same as the distance between 2 and 4. If this holds for our quantity, then great. If, on the other hand, scores were purely discrete, like grape variety, one-hot encoding will be a much better fit, as there's no implied ordering or distance. One-hot encoding will also be appropriate for quantitative scores when fractional values in-between integer scores, like 2.4, make no sense for the application - for when score is either this or that.

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

```
# In[8]:
target_onehot = torch.zeros(target.shape[0], 10)

target_onehot.scatter_(1, target.unsqueeze(1), 1.0)

# Out[8]:
tensor([[0., 0., ..., 0., 0.],
       [0., 0., ..., 0., 0.],
       ...,
       [0., 0., ..., 0., 0.],
       [0., 0., ..., 0., 0.]])
```

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

1. The dimension along which the following two arguments are specified
2. A column tensor indicating the indices of the elements to scatter
3. A tensor containing the elements to scatter or a single scalar to scatter (1 in this case)

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

The second argument of `scatter_`, the index tensor, is required to have the same number of dimensions as the tensor we scatter into. Since `target_onehot` has two dimensions (4898x10), we needed to add an extra dummy dimension to `target` using `unsqueeze`.

```
# In[9]:
target_unsqueezed = target.unsqueeze(1)
target_unsqueezed

# Out[9]:
tensor([[6],
       [6],
       ...,
       [7],
       [6]])
```

The call to `unsqueeze` added a *singleton* dimension, from a 1D tensor of 4898 elements to a 2D tensor of size (4898x1), without changing its contents - there are no extra elements added, we just decided to use an extra index to access the elements. That is, we access the first element of `target` as `target[0]` and the first element of its unsqueezed counterpart as `target_unsqueezed[0,0]`.

PyTorch allows us to use class indices directly as targets while training neural networks. However, if we wanted to use the score as a categorical input to the network, we would have had

to transform it to a *one-hot* encoded tensor.

Now we have seen ways to deal with both continuous and categorical data. You may ask what is the deal with the ordinal case discussed in the sidebar above. There is no general recipe for it. Most commonly, it is either treated as categorical (losing the ordering part, hoping that maybe our model will pick it up during training if we only have a few categories) or as continuous (introducing an arbitrary notion of distance). We will do the latter for the weather situation in 4.4 below. We summarize our data mapping in a small flow-chart in Figure-4.2.

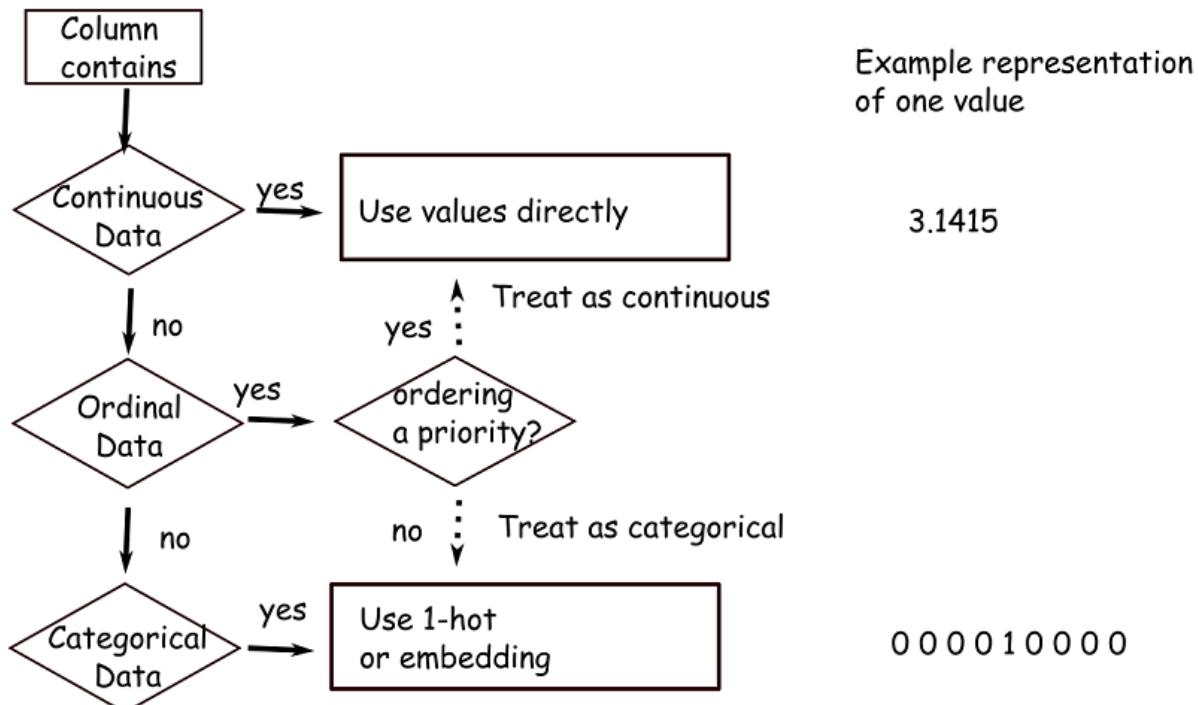


Figure 4.4 How to treat columns with continuous, ordinal and categorical data.

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

```

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

# Out[10]:
tensor([6.85e+00, 2.78e-01, 3.34e-01, 6.39e+00, 4.58e-02, 3.53e+01, 1.38e+02,
        9.94e-01, 3.19e+00, 4.90e-01, 1.05e+01])

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

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

```

In this case, `dim=0` indicates that the reduction is performed along dimension 0. At this point we can normalize the data by subtracting the mean and dividing by the standard deviation, which helps with the learning process (we'll discuss this in more detail in chapter 5, in 5.1.5):

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

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

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

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

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

- ① PyTorch also provides comparisons functions, here `torch.le(target, 3)`, but using operators seems to be a good standard.

Note that only 20 of the `bad_indexes` entries are set to `True`! By leveraging a feature in PyTorch called *advanced indexing*, we can use a tensor with data type `torch.bool` to index the `data` tensor. This will essentially filter `data` to be only items (or rows) corresponding to `True` in the indexing tensor. The `bad_indexes` tensor has the same shape as `target`, with values of `False` or `True` depending to the outcome of the comparison between our threshold and each element in the original `target` tensor.

```
# In[14]:
bad_data = data[bad_indexes]
bad_data.shape

# Out[14]:
torch.Size([20, 11])
```

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

```
# In[15]:
bad_data = data[target <= 3]
mid_data = data[(target > 3) & (target < 7)] ①
good_data = data[target >= 7]

bad_mean = torch.mean(bad_data, dim=0)
mid_mean = torch.mean(mid_data, dim=0)
```

```

good_mean = torch.mean(good_data, dim=0)

for i, args in enumerate(zip(col_list, bad_mean, mid_mean, good_mean)):
    print('{:2} {:20} {:6.2f} {:6.2f} {:6.2f}'.format(i, *args))

# Out[15]:
0 fixed acidity      7.60   6.89   6.73
1 volatile acidity   0.33   0.28   0.27
2 citric acid        0.34   0.34   0.33
3 residual sugar     6.39   6.71   5.26
4 chlorides          0.05   0.05   0.04
5 free sulfur dioxide 53.33  35.42  34.55
6 total sulfur dioxide 170.60 141.83 125.25
7 density            0.99   0.99   0.99
8 pH                 3.19   3.18   3.22
9 sulphates          0.47   0.49   0.50
10 alcohol           10.34  10.26  11.42

```

- ① For boolean numpy arrays and PyTorch tensors, the & operator does a logical "and" operation.

It looks like we're onto something here: at a first glance, the bad wines seem to have higher total sulfur dioxide, among other differences. We could use a threshold on total sulfur dioxide as a crude criterion for discriminating good wines from bad ones. Let's get the indexes where the total sulfur dioxide column is below the midpoint we calculated earlier, like so:

```

# In[16]:
total_sulfur_threshold = 141.83
total_sulfur_data = data[:,6]
predicted_indexes = torch.lt(total_sulfur_data, total_sulfur_threshold)

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

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

```

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

```

# In[17]:
actual_indexes = target > 5

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

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

```

Since there are about 500 more actually good wines than our threshold predicted, we already have hard evidence that it's not perfect. Now we need to see how well our predictions line up with the actual rankings. We will perform a logical "and" between our prediction indexes and the actual good indexes (remember that each is just an array of 0s and 1s), and use that intersection of wines-in-agreement to determine how well we did.

```

# In[18]:
n_matches = torch.sum(actual_indexes & predicted_indexes).item()
n_predicted = torch.sum(predicted_indexes).item()
n_actual = torch.sum(actual_indexes).item()

```

```
n_matches, n_matches / n_predicted, n_matches / n_actual  
# Out[18]:  
(2018, 0.74000733406674, 0.6193984039287906)
```

We got around 2000 wines right! Since we had 2700 wines predicted, that gives us a 74% chance that if we predict a wine to be high quality, it actually is. Unfortunately, there are 3200 good wines, and we only identified 61% of them. Well, I guess we got what we signed up for; that's only just barely better than random! Of course this is all very naive: we know for sure that there are multiple variables that 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 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. We'll have the tools to tackle this problem after the next two chapters, once we have learned how to build our first neural network from scratch.

We will also revisit how to better grade our results in chapter 12, Recall, and Pretty Pictures.

Let's move on to other data types for now.

4.4 Time Series

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

Going back to the wine dataset, we could have had a "year" column that allowed us 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 manually collecting the data samples, bottle by bottle. Stuff for our 2nd Edition.

In the meantime we'll switch to another interesting dataset: data from a Washington, D.C. bike sharing system reporting *hourly count of rental bikes between years 2011 and 2012 in the Capital bikeshare system with the corresponding weather and seasonal information*⁴⁶.

Our goal here will be to take a flat two-dimensional data set, and transform it into a three-dimensional one, as shown in Figure-4.3.

In the source data, each row is a separate hour of data (Figure-4.3 shows a transposed version of this to better fit on the printed page). We want to change the row-per-hour organization, so that

we have one axis that increases at a rate of one day per index increment, and another axis that represents hour of the day (independent of the date). The third axis will be our different columns of data (weather, temperature, etc.).

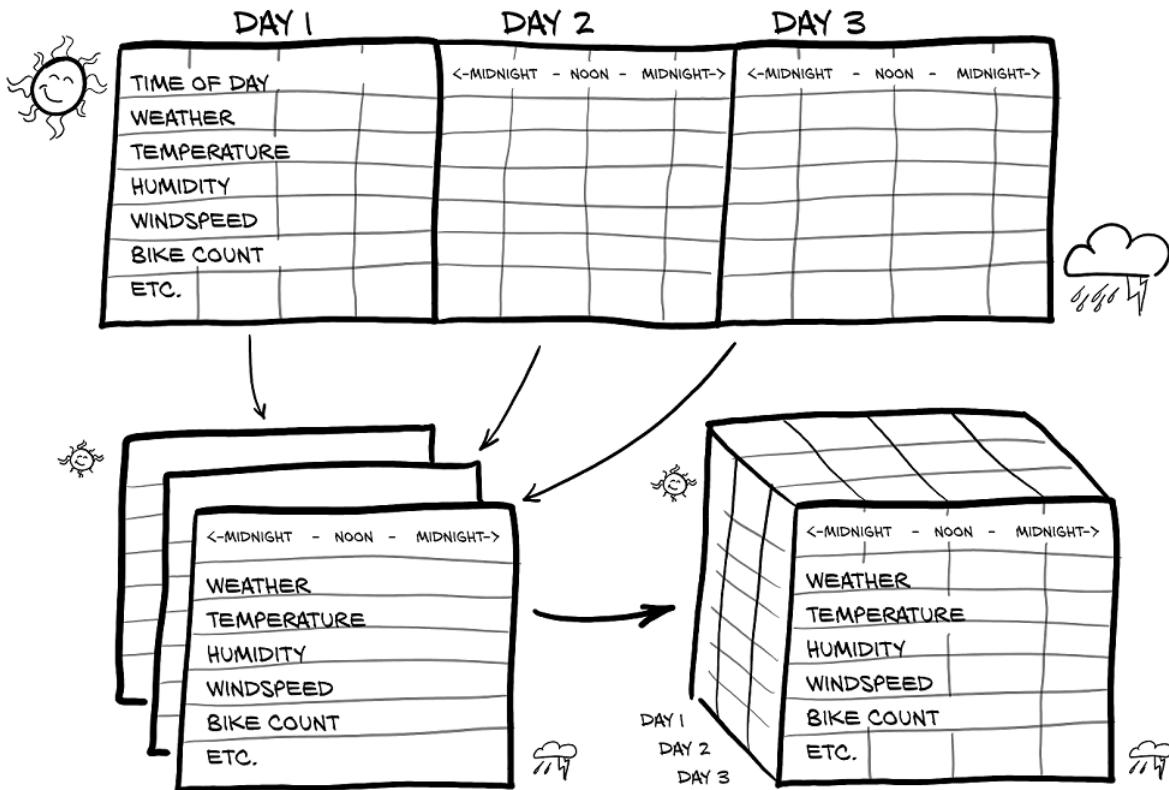


Figure 4.5 Transforming a 1D, multi-channel dataset into a 2D, multi-channel dataset by separating the date and hour of each sample into separate axes.

Let's load the data.

Listing 4.4 code/p1ch4/2_time_series_bikes.ipynb

```
# In[2]:
bikes_numpy = np.loadtxt("../data/plch4/bike-sharing-dataset/hour-fixed.csv",
                      dtype=np.float32,
                      delimiter=",",
                      skiprows=1,
                      converters={1: lambda x: float(x[8:10])}) ①
bikes = torch.from_numpy(bikes_numpy)
bikes

# Out[2]:
tensor([[1.0000e+00, 1.0000e+00, ..., 1.3000e+01, 1.6000e+01],
       [2.0000e+00, 1.0000e+00, ..., 3.2000e+01, 4.0000e+01],
       ...,
       [1.7378e+04, 3.1000e+01, ..., 4.8000e+01, 6.1000e+01],
       [1.7379e+04, 3.1000e+01, ..., 3.7000e+01, 4.9000e+01]])
```

- ① We convert date strings to numbers corresponding to the day of the month in column 1.

For every hour, the dataset reports the following variables

instant	①
day	②
season	③
yr	④
mnth	⑤
hr	⑥
holiday	⑦
weekday	⑧
workingday	⑨
weathersit	⑩
temp	⑪
atemp	⑫
hum	⑬
windspeed	⑭
casual	⑮
registered	⑯
cnt	⑰

- ① index of record
- ② day of month
- ③ season (1: spring, 2: summer, 3: fall, 4: winter)
- ④ year (0: 2011, 1: 2012)
- ⑤ month (1 to 12)
- ⑥ hour (0 to 23)
- ⑦ holiday status
- ⑧ day of the week
- ⑨ working day status
- ⑩ weather situation (1: clear, 2:mist, 3: light rain/snow, 4: heavy rain/snow)
- ⑪ temperature in C
- ⑫ perceived temperature in C
- ⑬ humidity
- ⑭ windspeed
- ⑮ number of causal users
- ⑯ number of registered users
- ⑰ count of rental bikes

In a time series dataset such as this one, rows represent successive time-points: there is a dimension along which they are ordered. Sure, one could treat each row as independent and try to predict the number of circulating bikes based at, say, a particular time of day irrespective of what happened at an earlier time. However, the existence of an ordering gives us the opportunity to exploit causal relationships across time. For instance, it allows us to predict bike rides at one time based on the fact that it was raining at an earlier time. For the time being, we're going to

focus on learning how to turn our bike sharing dataset into something that our neural network will be able to ingest in fixed-size chunks.

Said neural network model will need to see a number of sequences of values for each different quantity, such as ride count, time of day, temperature, weather conditions. So n parallel sequences of size c . c stands for *channel*, in neural network parlance, and is the same as *column* for one-dimensional data like we have here. The n dimension represents the time axis, here one entry per hour.

We might want to break up the 2-year dataset in wider observation periods, like days. This way we'll have n (for *number of samples*) collections of c sequences of length L . In other words, our time-series dataset will be a tensor of dimension 3 and shape $n \times c \times L$.¹ The c would remain our 17 channels, while 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.

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

NOTE The version of the file we use, `hour-fixed.csv`, has had some processing done to include rows missing from the original data set. We presumed that the missing hours had zero bikes active (they were typically in the early morning hours).

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

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

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

```
# In[4]:  
daily_bikes = bikes.view(-1, 24, bikes.shape[1])  
daily_bikes.shape, daily_bikes.stride()  
  
# Out[4]:  
(torch.Size([730, 24, 17]), (408, 17, 1))
```

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

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

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

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

Here we see that the right-most dimension is the number of columns in the original dataset. Then in the middle dimension we have time, split in chunks of 24 sequential hours. In other words, we now have `N` sequences of `L` hours in a day, for `C` channels. To get to our desired `NxCxL` ordering, we need to transpose the tensor:

```
# In[5]:
daily_bikes = daily_bikes.transpose(1, 2)
daily_bikes.shape, daily_bikes.stride()

# Out[5]:
(torch.Size([730, 17, 24]), (408, 1, 17))
```

The "weather situation" variable is ordinal. In fact it has 4 levels, 1 for good weather, 4 for, er, really bad. We could treat this variable as categorical, with levels interpreted as labels, or as a continuous variable. If we decided for categorical, we would turn the variable into a one-hot encoded vector and concatenate the columns with the dataset.⁴⁷

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

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

```
# Out[6]:  
tensor([1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 2, 2, 2])
```

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

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

- ❶ We are also decreasing the values by one because weather situation ranges from 1 to 4, while indices are 0-based.

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

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

```
# In[8]:  
torch.cat((bikes[:24], weather_onehot), 1)[:1]  
  
# Out[8]:  
tensor([[ 1.0000,  1.0000,  1.0000,  0.0000,  1.0000,  0.0000,  0.0000,  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 we prescribed our original `bikes` dataset and our one-hot encoded "weather situation" matrix to be concatenated along the *column* dimension (i.e. 1). In other words, the columns of the two datasets are stacked together, or, equivalently, the new one-hot encoded columns are appended to the original dataset. For `cat` to succeed, it is required that the tensors have the same size along the other dimensions - the *row* dimension, in this case.

Note that our new last four columns are 1, 0, 0, 0; exactly as we would expect with a weather value of 1.

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

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

```
# Out[9]:  
torch.Size([730, 4, 24])
```

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

```
# In[10]:  
daily_weather_onehot.scatter_(1, daily_bikes[:, 9, :].long().unsqueeze(1) - 1, 1.0)  
daily_weather_onehot.shape  
  
# Out[10]:  
torch.Size([730, 4, 24])
```

and we concatenate along the c dimension:

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

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

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

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

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

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

or we can 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, our variable will have zero mean and unitary standard deviation. If our variable were drawn from a Gaussian distribution, 68% of the samples would sit in the [-1.0, 1.0] interval.

Great, we've built another nice dataset we'll get to use later on. For now, it's important only that we got an idea of how a time series is laid out and how we can wrangle the data in a form that a

network will digest.

There are other kinds of data that look like a time series, in that there is a strict ordering. Top two of the list? Text and audio. We'll take a look at text next, and 1.5 has links to additional examples for audio.

4.5 Text

Deep learning has taken the field of Natural Language Processing (NLP) by storm, particularly using models that repeatedly consume a combination of new input and previous model output. These models are called *recurrent neural networks*, and they have been applied with great success to text categorization, text generation and automated translation systems. More recently, a class of networks called *transformers* with a more flexible way to incorporate past information has made a big splash. Previous NLP workloads were characterized by sophisticated multi-stage pipelines that included rules encoding the grammar of a language^{48 49}. Now, state of the art work trains networks end-to-end on large corpuses starting from scratch, letting those rules emerge from data. For the last several years, the most used automated translation systems available as services on the Internet have been based on deep learning.

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

There are two particularly intuitive levels at which networks operate on text: at the character level, by processing one character at a time, and at the word level, where individual words are the finest grained entity to be seen by the network. The technique with which we encode text information into tensor form is the same whether we operate at the character or at the word level. And it's nothing magic either, we've stumbled upon it earlier: it's one-hot encoding.

Let's start with a character-level example. First off, let's get some text to process. An amazing resource here is Project Gutenberg⁵⁰, a volunteer effort to digitize and archive cultural work and make it available for free in open formats, including plain text files. If we're aiming at larger-scale corpora, the Wikipedia corpus stands out: it's the complete collection of Wikipedia articles containing 1.9 billion words and more than 4.4 million articles. Several other corpora can be found at the English Corpora website⁵¹.

Let's load Jane Austen's Pride and Prejudice from the Project Gutenberg website⁵². Let's just save the file and read it in:

Listing 4.5 code/p1ch4/3_text_jane_austen.ipynb

```
# In[2]:  
with open('../data/plch4/jane-austen/1342-0.txt', encoding='utf8') as f:  
    text = f.read()
```

There's one more detail we need to take care about before we proceed: encoding. This is a pretty vast subject and we will just touch upon it. Every written character is represented by a code, a sequence of bits of appropriate length so that each character can be uniquely identified. The simplest of such encoding is ASCII, which stands for American Standard Code for Information Interchange, dating back to the '60s. ASCII encodes 128 characters using 128 integers. For instance, letter *a* corresponds to binary 1100001, or decimal 97, letter *b* to binary 1100010, or decimal 98, and so on. The encoding would fit 8 bits, which was a big bonus in 1965.

NOTE

128 characters are clearly not enough to account for all the glyphs, accents, ligatures, etc. that are needed to properly represent written text in languages beyond English. To this end, a number of encodings have been developed, whereby a larger number of bits is used 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, while the representation in bits of those numbers is provided by a specific encoding. Popular encodings are UTF-8, UTF-16 and UTF-32, whereby the numbers are a sequence of 8, 16 or 32 bit integers. Strings in Python 3.x are Unicode strings.

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

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

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

```
# In[3]:  
lines = text.split('\n')  
line = lines[200]  
line
```

```
# Out[3]:  
'"Impossible, Mr. Bennet, impossible, when I am not acquainted with him'
```

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

```
# In[4]:  
letter_t = torch.zeros(len(line), 128) ❶  
letter_t.shape  
  
# Out[4]:  
torch.Size([70, 128])
```

- ❶ 128 hardcoded due to the limits of ASCII.

Note that `letter_t` will hold a one-hot encoded character per row. Now we just have to 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 ❶  
    letter_t[i][letter_index] = 1
```

- ❶ The text uses directional double-quotes, which are not valid ASCII, so we screen them out here.

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

We'll define `clean_words`, which takes text and returns it lower-case and stripped of punctuation. When we call it on our "Impossible, Mr. Bennet" line, we get the following:

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

```
'i',
'am',
'not',
'acquainted',
'with',
'him'])
```

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

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

len(word2index_dict), word2index_dict['impossible']

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

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

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

print(word_t.shape)

# Out[8]:
0 3394 impossible
1 4305 mr
2 813 bennet
3 3394 impossible
4 7078 when
5 3315 i
6 415 am
7 4436 not
8 239 acquainted
9 7148 with
10 3215 him
torch.Size([11, 7261])
```

At this point, `tensor` represents one sentence of length 11 in an encoding space of size 7261, the number of words in our dictionary.

We compare the gist of our two options for splitting text (and using the embeddings we look at in the next section) in Figure-4.4.

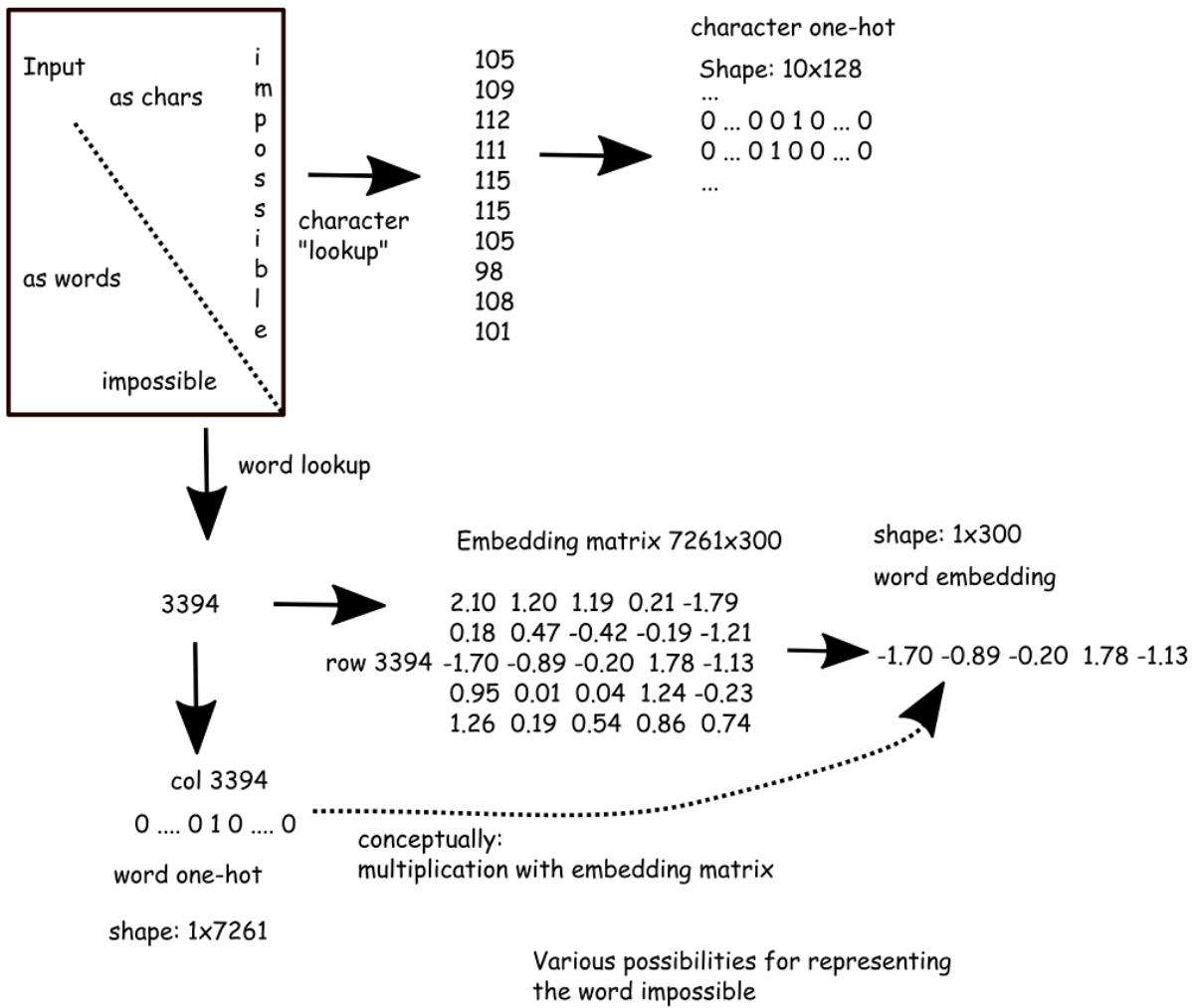


Figure 4.6 Three ways to encode a word.

The choice between character-level and word-level encoding leaves us to make a tradeoff. In many languages, there will be significantly fewer characters than words, so representing characters has us representing just a few classes while representing words requires us to represent a very large number of classes and in, in any practical application, deal with words not in the dictionary. On the other hand, words do convey much more meaning than individual characters, so a representation of words is considerably more informative by itself. Given the stark contrast between these two options, it is perhaps unsurprising that an intermediate way has been sought and indeed been found and applied with great success: For example, the *byte pair encoding* method⁵³ starts with a dictionary individual letters but then iteratively adds the most frequently observed pairs to the dictionary until reaching a prescribed dictionary size. Our example sentence might then be split into tokens shown below.⁵⁴

```
Im|pos|s|ible|, |Mr| . |B|en|net|, |impossible|, |when| I |am|not|acquainted|with|him
```

For most things, our mapping is just splitting by words, but the more rare parts — the capitalized "impossible" and the name — are composed of subunits.

4.5.1 Text embeddings

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

Now, we certainly could do some work to deduplicate words, condense alternate spellings, collapse past and future tenses into a single token, that kind of thing. Still, a general-purpose english-language encoding is going to be *huge*. Even worse, every time we encounter a new word, we have to add a new column to the vector, which means 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 we compress our encoding down to a more manageable size and put a cap on the size growth? Well, instead of vectors of many zeros and a single 1, we 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 now to find an effective way to map individual words into this 100-dimensional space in a way that facilitates downstream learning. This is called an *embedding*.

In principle, we could simply iterate over our vocabulary and generate a set of 100 random floating point numbers for each word. This would work, in that we could cram a very large vocabulary in just 100 numbers, but it would forego any concept of distance between words based on meaning or context. A model using this word embedding would have to deal with very little structure in its input vectors. An ideal solution would be to generate the embedding in such a way that words used in similar contexts map to nearby regions of the embedding.

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

As we start embedding words we can map `apple` to a number in the `fruit` and `red` quadrant. Likewise, we can easily map `tangerine`, `lemon`, `lychee`, and `kiwi` (to round out our list of colorful fruits). Then we can start on flowers, and assign `rose`, `poppy`, `daffodil`, `lily`, and... Hrm. Not many brown flowers out there. Well, `sunflower` can get `flower`, `yellow`, and `brown`, and then `daisy` can get `flower`, `white`, and `yellow`. Perhaps we should update `kiwi` to map close to `fruit`, `brown`, and `green`⁵⁵. For dogs and color, we can embed `redbone` near `red`, `uh`, `fox` perhaps for `orange`, `golden retriever`, `poodles` for `white`, and... most kinds of dogs are brown.

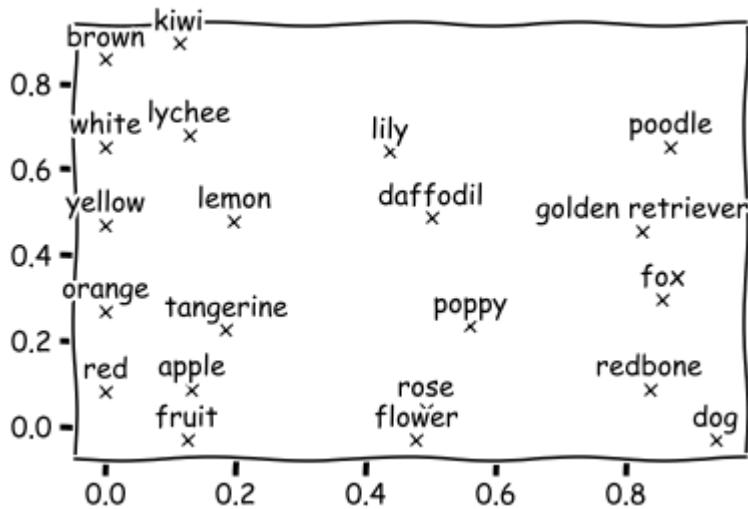


Figure 4.7 Our manual word embeddings

While doing this manually isn't really feasible for a large corpus, what we should note is that while we had an embedding size of 2, we described 15 different words *besides the base eight*, and could probably cram quite a few more in should we take the time to be creative about it.

As we've probably already guessed, this kind of work can be automated. By processing a large corpus of organic text, embeddings similar to the one we discussed can be generated. The main differences are that there are 100 to 1000 elements in the embedding vector, and that axes do not map directly to concepts, but rather conceptually similar words map in neighboring regions of an embedding space, whose axes are arbitrary floating point dimensions.

While the exact algorithms⁵⁶ used are a bit out of scope for what we're wanting to focus on here, we'd just like to mention that embeddings are often generated using neural networks, trying to predict a word from nearby words (the context) in a sentence. In this case, we could start from one-hot encoded words and use a (usually rather shallow) neural network to generate the embedding. Once the embedding is available, we could use it for downstream tasks.

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

More contemporary embedding models — with BERT and GPT-2 making headlines even in mainstream media — are much more elaborate and are context-sensitive, i.e. the mapping of a word in the vocabulary to a vector will not be fixed but depend on the surrounding sentence. Yet, they are often used just like the more simple *classic* embeddings we touched upon here.

4.5.2 Text embeddings a a blueprint

Embeddings are an essential tool for when a large number of entries in the vocabulary have to be represented by numeric vectors. But we won't be using text and text embeddings in this book, so you might wonder why we introduce them here. We believe that how text is represented and processed can also be seen as an example for dealing with categorical data in general. Embeddings are useful wherever one-hot encoding becomes cumbersome. Indeed, in the form described above they are an efficient way of representing one-hot encoding immediately followed by multiplication with the matrix containing the embedding vectors as rows.

In non-text applications we likely do not have the ability to construct the embeddings beforehand, but we will start with the random numbers we eschewed above and consider improving them part of our learning problem. This is a standard technique so much that embeddings are a prominent alternative to one-hot embeddings for any categorical data. On the flip side, even when we deal with text, improving the pre-learned embeddings while solving our problem at hand has become a common practice⁵⁷.

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

4.6 Conclusion

We covered a lot of ground this chapter. We have learned to load the most common types of data and shape them up for consumption by a neural network. Of course, there are more data formats in the wild than we could hope to describe in a single volume. Some, like medical histories, are too complex to cover in this volume. Others, like audio and video, were deemed less crucial for the path of this book. For the interested reader, however, we do provide short examples of audio and video tensor creation in bonus Jupyter notebooks provided in our code repository.⁵⁸ Now that we're familiar with tensors and how to store data in them, we can now move on to the next step for the goal of the book: teaching you to train deep neural networks! The next chapter covers 5 for simple linear models.

4.7 Exercises

- Take several pictures of red, blue, and green items with your phone or other digital camera.⁵⁹
 - Load each image and convert them to tensors.
 - For each image tensor, use the `.mean()` method to get a sense of how bright the image is.
 - Now take the mean of each channel of your images. Can you identify the red, green, blue items from only the channel averages?
- Select a relatively large file containing python source code.
 - Build an index of all of the words in the source file (feel free to make your tokenization as simple or as complex as you like; we suggest starting with replacing `r" [^a-zA-Z0-9_]+"` with spaces).
 - Compare your index with the one we 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 to what's lost in the *Pride and Prejudice* encoding?

4.8 Summary

- Neural networks require data to be represented as multi-dimensional 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, e.g. convolutional vs recurrent; reshaping of data can be achieved effectively with the PyTorch tensor API.
- Spreadsheets can be very 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. Very often, embeddings give good and efficient representations.
- Images can have one or many channels. The most common is the red-green-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 all be stored in a 32-bit floating point number without loss of precision.

The Mechanics of Learning

5

This chapter covers:

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

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

As a matter of fact, building models that allow us to explain input/output relationships dates back centuries at least. When Johannes Kepler, a German mathematical astronomer who lived between 1571 and 1630, figured out his three laws of planetary motion in the early 1600's, 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 (actually, Netwon's used Kepler's work to figure his things out), he extrapolated the simplest possible geometric model that could fit the data. And by the way, it took him six years of staring at data that didn't make sense to him, and incremental realizations, to finally formulate these laws.⁶⁰ We can see this process in 5.1.

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, like 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. Once he figured out the second law — "A line joining a planet and the Sun sweeps out equal areas during equal intervals of time" — he could also tell *when* a planet would be at a particular point in space given observations in time.⁶¹

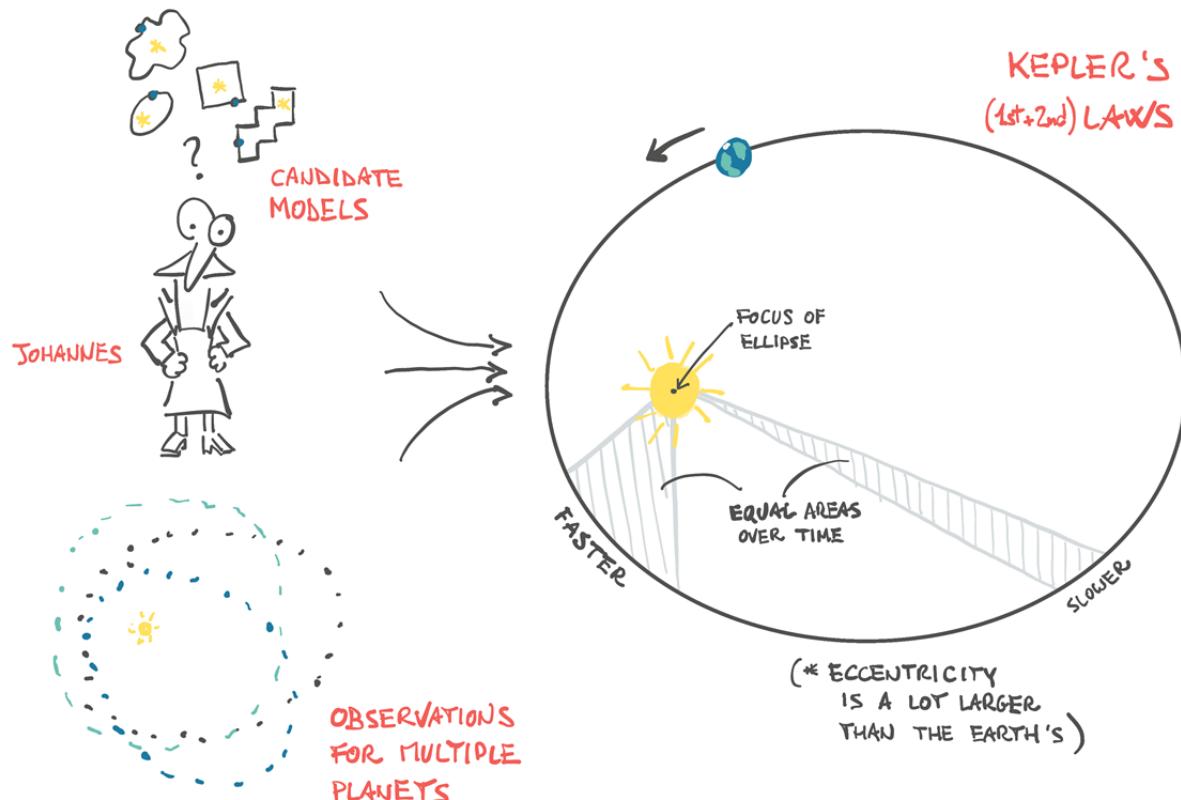


Figure 5.1 Johannes Kepler considers multiple candidate models that might fit the data at hand, settling on an ellipse.

So, 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? We can learn how from Kepler's own recollection in his book *New Astronomy*, or, from how J.V. Field put it in his "The origins of proof" series⁶²:

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

— J.V. Field *The origins of proof*

So let's sum things up. Over those six years, Johannes:

1. got lots of good data from his friend Tycho (not without some struggle)
2. tried to visualize the heck out of it, because he felt there was something fishy going on
3. chose the simplest possible model that had a chance to fit the data (an ellipse)
4. split the data so that he could work on a 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 literally constructed on the seven steps laid out above. And we have learned over the centuries that deviating from those is a recipe for disaster⁶³.

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

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

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

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

5.1 Learning is just parameter estimation

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

Figure-5.2 shows the high-level overview of what we'll 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 to the ground truth. In order to optimize the parameter of the model, its *weights*, the change in the error following a unit change in weights (i.e. the *gradient* of the error with respect to the parameters) is computed using the chain rule for the derivative of a composite function (backward pass). The value of the weights is then updated in the direction that leads to a decrease in the error. The procedure is repeated until the error, evaluated on unseen data, falls below an acceptable level. If what we just said sounds obscure, we've got a whole chapter to clear things out. By the time we're done, all the pieces will fall into place and the above paragraph will make perfect sense.

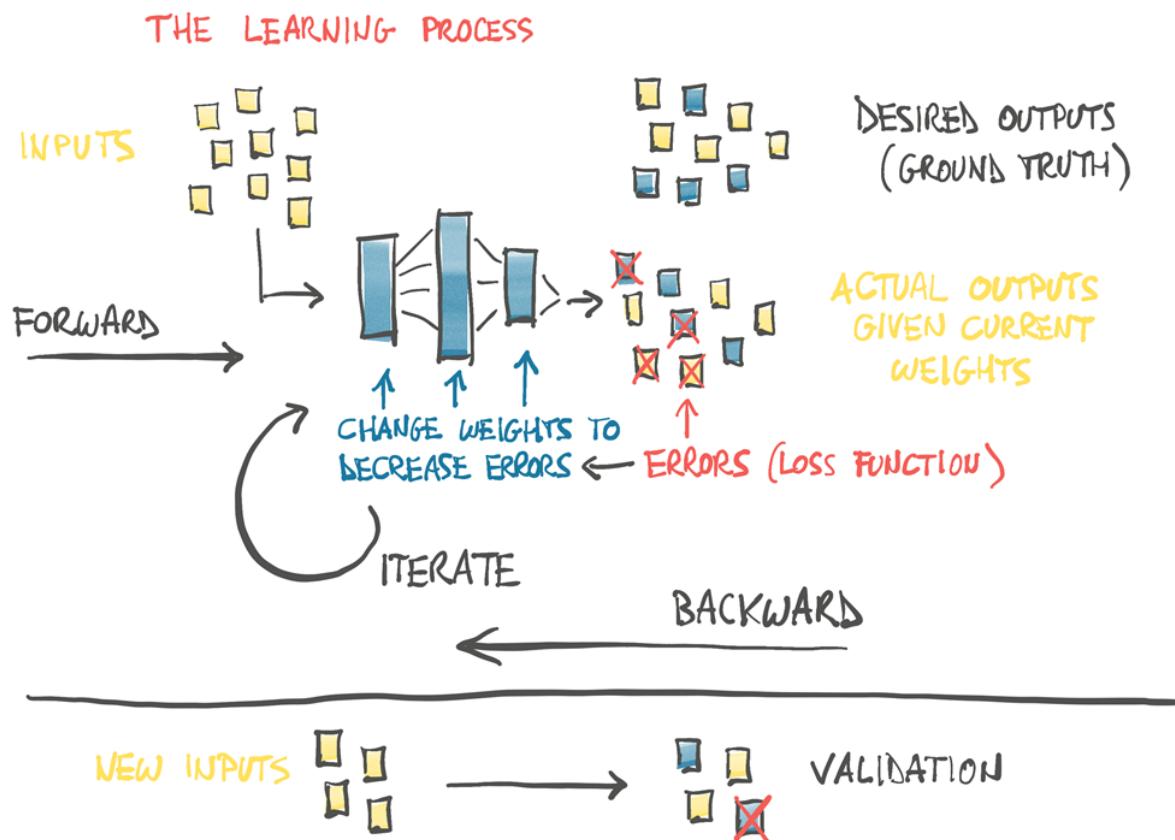


Figure 5.2 Our mental model of the learning process.

We're now going to take a problem with a noisy data set, build a model and implement a

learning algorithm for it. As we start we'll be doing everything by hand, but by the end of the chapter we'll be letting PyTorch do all the heavy lifting for us. By the end we will have covered many of the essential concepts that underlie training deep neural networks, even if our motivating example is very simple and our model isn't actually a neural network (yet!).

5.1.1 A Hot Problem

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

We'll start by making a note of temperature data in good old Celsius⁶⁴ and measurements from our new thermometer and figure things out.

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

Listing 5.1 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)
```

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

5.1.2 Choosing a linear model as a first try

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

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

Is this a reasonable assumption? Probably it is; we'll see how well the final model performs. We chose to name w and b after *weight* and *bias*, two very common terms for linear scaling and the additive constant - we'll bump into those all the time.

NOTE

Spoiler alert: we know that a linear model is correct because the problem and data have been fabricated, but please bear with us; it's a useful motivating example to build our understanding of what PyTorch is doing under the hood.

Ok, we'll now need to estimate w and b , the parameters in our model, based on the data we have. We must do it so that temperatures we obtain from running the unknown temperatures t_u through the model are close to temperatures we actually measured in Celsius. If that sounds like fitting a line through a set of measurements, well, yes, because that's exactly what we're doing. We'll go through this simple example using PyTorch and realize that training a neural network will essentially involve changing the model for a slightly more elaborate one, with a few (or a metric ton) more parameters.

Let's flesh it out again: we have a model with some unknown parameters, we need to estimate such parameters so that the error between predicted outputs and measured values are as low as possible. We notice that we still need to exactly 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. Our optimization process should therefore aim at finding w and b so that the loss function is at a minimum.

5.1.3 Less loss is what we want

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

We need to make sure the loss function makes the loss positive both when t_p is above and when below the true t_c , since the goal is to minimize this value (being able to push the loss infinitely negative isn't useful). We have a few choices, the most straightforward being $|t_p - t_c|$ and $(t_p - t_c)^2$. Based on the mathematical expression we choose, we can emphasise or discount certain errors. Conceptually, a loss function is a way of prioritizing which errors to fix from our training samples, so that our parameter updates result in adjustments to the outputs for the highly-weighted samples instead of changes to some other samples' output that had a smaller loss.

Both of the example loss functions above have a clear minimum in zero and grow monotonically as the predicted value moves further apart from the true value in either direction. As the steepness of the growth is also monotonically increasing away from the minimum, both of them are said to be *convex*. Since our model is linear, the loss as a function of w and b is also convex.

Cases where the loss is a convex function of the model parameters are usually great to deal with because one can find a minimum in a very efficient way through specialized algorithms. For deep neural networks, the loss is not a convex function of the inputs, however, so those methods aren't generally useful to us.

For our two loss functions $|t_p - t_c|$ and $(t_p - t_c)^2$ as shown in Figure-5.5, we 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 we'd like to converge. This is less of an issue than it looks in practice, but we'll stick to the square of differences for the time being.

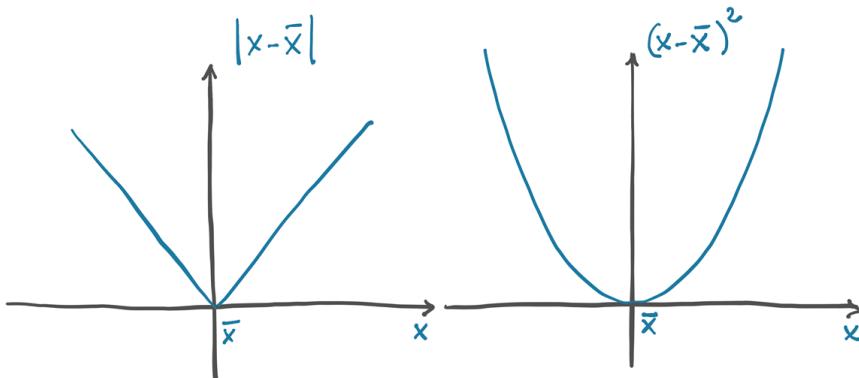


Figure 5.3 Absolute difference versus difference squared.

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 as desired.

5.1.4 From Problem to PyTorch

We've figured out the model and the loss function - we've already got a good part of the high-level picture in Figure-5.2 figured out. Now we need to set the learning process in motion and feed it actual data. Also, enough with math notation already, let's switch to PyTorch - after all we came here for the *fun*.

We've already created our data tensors, so now let's write out the model as a Python function:

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

where we're expecting t_u , w and b to be the input tensor, weight parameter, and bias parameter, respectively. In our model the parameters will be PyTorch scalars (aka zero-dimensional tensors), and the product operation will use broadcasting to yield the returned tensors. Anyway, time to define our loss:

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

Note that we are building a tensor of differences, taking their square element-wise, and finally producing a scalar loss function by averaging all elements in the resulting tensor. It is a *mean square loss*.

We can now initialize the parameters, invoke the model:

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

t_p = model(t_u, w, b)
t_p

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

We've got the model and the loss implemented in this section. We finally got to the meat of the section: how do we estimate the `w` and `b` such that the loss reaches a minimum? We'll first work things out by hand, then learn how to leverage PyTorch superpowers to solve the same problem in a more general, off-the-shelf way.

BROADCASTING

We mentioned broadcasting in 3, but we promised to look at it more carefully when we need it. In the above, we have two scalars (0-dimensional tensors) `w` and `b` and we multiply them with and add them to vectors (1-dimensional tensors) of length `b`.

Usually — and in early versions of PyTorch, too — we can only use elementwise binary operations such as addition, subtraction, multiplication, and division for arguments of the same shape. The entries in matching positions of each of the tensors will be used to calculate the corresponding entry in the result tensor.

Broadcasting, popular in NumPy and adapted by PyTorch, relaxes this assumption for most binary operations. It uses the following rules to match tensor elements:

- For each index dimension counted from the back, if one of the operands is of size 1 in that dimension, PyTorch will use the single "entry" along this dimension with each of the

entries in the other tensor along this dimension.

- If both sizes are greater than one, they must be the same and the natural matching is used.
- If one of the tensors is of higher dimension, the entirety of the other tensor will be used for each entry along these dimensions.

This sounds complicated (and it can be error prone when not paying close attention, which is why we have named tensor dimensions as seen in 3.3), but usually, we can either write down the tensor dimensions to see what happens or picture what happens by using space dimensions to show the broadcasting as in the following image:

PLACEHOLDER

Of course, this would all be theory if we didn't have some code examples.

```
# In[7]:
x = torch.ones(())
y = torch.ones(3,1)
z = torch.ones(1,3)
a = torch.ones(2, 1, 1)
print(f"shapes: x: {x.shape}, y: {y.shape}, z: {z.shape}, a: {a.shape}")
print("x * y:", (x * y).shape)
print("y * z:", (y * z).shape)
print("y * z * a:", (y * z * a).shape)

# Out[7]:
shapes: x: torch.Size([]), y: torch.Size([3, 1]), z: torch.Size([1, 3]), a: torch.Size([2, 1, 1])
x * y: torch.Size([3, 1])
y * z: torch.Size([3, 3])
y * z * a: torch.Size([2, 3, 3])
```

But now enough about broadcasting, let's get back to using our model.

5.1.5 Down Along the Gradient

We'll optimize the loss function with respect to the parameters using the so-called *gradient descent* algorithm. In this section we'll build our intuition for how gradient descent works from first principles, which will help us a lot in the future. As we mentioned there are ways to solve our particular example problem more efficiently, but those approaches aren't applicable to most deep learning tasks. Gradient descent is actually a very simple idea, and it scales up surprisingly well to large neural network models with millions of parameters.

Let's start with a mental image, which we conveniently sketched out in Figure-5.6. Suppose we are in front of a machine sporting two knobs, labeled w and b . We are allowed to see the value of the loss on a screen, and are told to minimize that value. Not knowing the effect of the knobs on the loss, we would probably start fiddling with those and decide for each knob what is the direction that makes the loss decrease. We would probably decide to rotate both knobs in their direction of decreasing loss. Supposing we're far from the optimal value: we'd likely see the loss decreasing quickly, then slow down as it gets closer to the minimum. We would notice that at

some point the loss climbs back up again, so we would invert the direction of rotation for one or both. We would also learn that when the loss changes slowly it's a good idea to adjust the knobs more finely, to avoid reaching the point where the loss goes back up. After a while, eventually, we'd converge to a minimum.

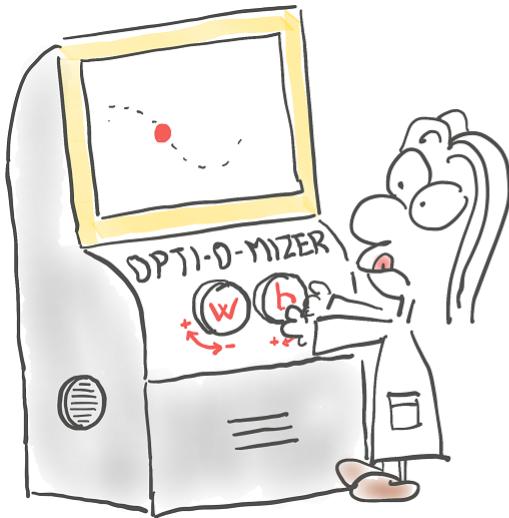


Figure 5.4 A cartoon depiction of the optimization process, where a person with knobs for w and b searches for the direction that makes the loss decrease.

Gradient descent is not that 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. Just like when we were fiddling with the knobs, we could estimate such rate of change by applying a small change to w and b and seeing how much the loss is changing in that neighborhood:

```
# In[8]:
delta = 0.1

loss_rate_of_change_w = \
    (loss_fn(model(t_u, w + delta, b), t_c) - \
     loss_fn(model(t_u, w - delta, b), t_c)) / (2.0 * delta)
```

The above is saying: 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, then we'll need to increase w to minimize the loss, whereas if the change is positive, we'll need to decrease w . By how much? Applying a change to w that is proportional to the rate of change of the loss is a good idea, especially when the loss has several parameters: we would apply a change to those that exert a significant change on the loss. It is also wise to change the parameters slowly in general, because the rate of change could be dramatically different at a distance from the neighborhood of the current w value. Therefore we should scale the rate of change by a typically small factor. This scaling factor has many names; the one we use in machine learning is `learning_rate`.

```
# In[9]:
learning_rate = 1e-2
```

```
w = w - learning_rate * loss_rate_of_change_w
```

We can do the same with b :

```
# In[10]:  
loss_rate_of_change_b = \  
    (loss_fn(model(t_u, w, b + delta), t_c) -  
     loss_fn(model(t_u, w, b - delta), t_c)) / (2.0 * delta)  
  
b = b - learning_rate * loss_rate_of_change_b
```

This represents the basic parameter update step for gradient descent. By re-iterating the above evaluations (and provided we choose a small enough learning rate) we would converge to an optimal value of the parameters for which the loss computed on the given data is minimal. We'll show the complete iterative process soon, but the way we just computed our rates of change is rather crude and need an upgrade before we move on. Let's see why and how.

5.1.6 Getting Analytical

Computing the rate of change using repeated evaluations of model and loss in order to probe the behavior of the loss function in the neighborhood of w and b doesn't scale well to models with many parameters. Also, it is not always clear how large that neighborhood should be. We chose δ equal to 0.1 above, but it all depends on the shape of the loss as a function of w and b . If the loss changes too quickly compared to δ , we won't have a very good idea on where downhill is.

What if we could make the neighborhood infinitesimally small, as in Figure-5.7? That's exactly what happens when we take the derivative of the loss with respect to a parameter analytically. In a model with two or more parameters as the one we're dealing with, we compute the individual derivatives of the loss with respect to each parameter and put them in a vector of derivatives, the *gradient*.

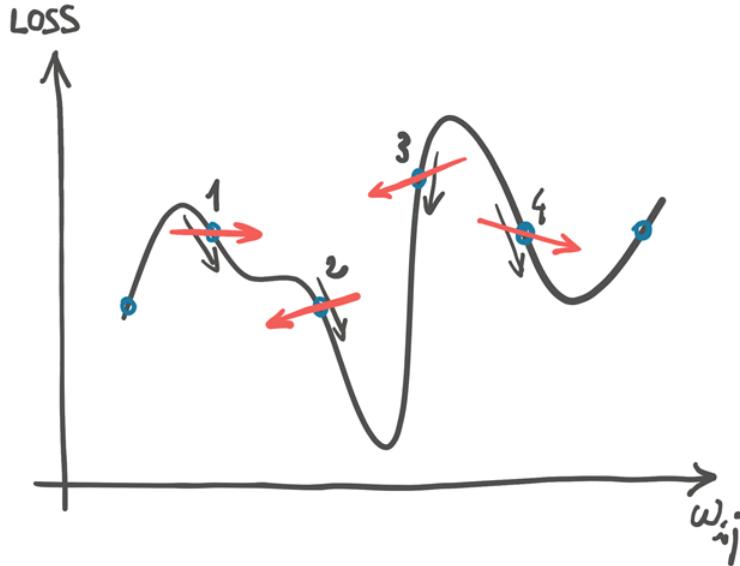


Figure 5.5 Differences in the estimated downhill directions when evaluating them at discrete locations vs analytically.

In order to compute the derivative of the loss with respect to a parameter, we can apply the chain rule and compute the derivative of the loss with respect to its input (which is the output of the model), times the derivative of the model with respect to the parameter:

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

Recall that our model is a linear function and our loss is a sum of squares. Let's figure out the expressions for the derivatives. Recalling the expression for the loss:

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

and remembering that $d x^2 / d x = 2 x$, we get

```
# In[11]:
def dloss_fn(t_p, t_c):
    dsq_diffs = 2 * (t_p - t_c) / t_p.size(0) ①
    return dsq_diffs
```

① The division is from the the derivative of mean.

As for the model, recalling that our model is:

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

we get derivatives of:

```
# In[12]:
```

```
def dmodel_dw(t_u, w, b):  
    return t_u
```

```
# In[13]:  
def dmodel_db(t_u, w, b):  
    return 1.0
```

Putting all this together, the function returning the gradient of the loss with respect to w and b is:

```
# In[14]:  
def grad_fn(t_u, t_c, t_p, w, b):  
    dloss_dtp = dloss_fn(t_p, t_c)  
    dloss_dw = dloss_dtp * dmodel_dw(t_u, w, b)  
    dloss_db = dloss_dtp * dmodel_db(t_u, w, b)  
    return torch.stack([dloss_dw.sum(), dloss_db.sum()]) ①
```

- ① The summation is the backward of the broadcasting we implicitly do when applying the parameters to an entire vector of inputs in `model`.

The same idea expressed in mathematical notation is shown in 5.6.

$$\nabla_{w,b} L = \left(\frac{\partial L}{\partial w}, \frac{\partial L}{\partial b} \right) = \left(\frac{\partial L}{\partial m} \cdot \frac{\partial m}{\partial w}, \frac{\partial L}{\partial m} \cdot \frac{\partial m}{\partial b} \right)$$

Diagram annotations:

- A blue arrow labeled "gradient" points to the left side of the equation.
- A blue arrow labeled "partial derivatives" points to the two terms inside the parentheses.
- A blue arrow labeled "model" points to the term $\frac{\partial L}{\partial m}$.
- A blue arrow labeled "parameters" points to the term $\frac{\partial m}{\partial w}$ and $\frac{\partial m}{\partial b}$.
- A blue arrow labeled "loss $L(m_{w,b}(x))$ " points to the top of the equation.

Figure 5.6 The derivative of the loss function with respect to the weights.

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

5.1.7 The Training Loop

We now have everything in place to optimize our parameters. Starting from a tentative value for a parameter, we can iteratively apply updates to it for a fixed number of iterations, or until w and b stop changing. There are several stopping criteria, we'll stick to a fixed number of iterations for now.

Since we're at it, let's introduce another piece of terminology. We call a training iteration during which we update the parameters for all of our training samples an *epoch*.

The complete training loop looks like this:

Listing 5.2 code/p1ch5/1_parameter_estimation.ipynb

```
# In[15]:
def training_loop(n_epochs, learning_rate, params, t_u, t_c):
    for epoch in range(1, n_epochs + 1):
        w, b = params

        t_p = model(t_u, w, b)      ❶
        loss = loss_fn(t_p, t_c)
        grad = grad_fn(t_u, t_c, t_p, w, b)  ❷

        params = params - learning_rate * grad

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

    return params
```

- ❶ This is the forward pass
- ❷ And this is the backward pass
- ❸ This logging line can be very verbose.

The actual logging logic used for the output in this text is more complicated (see cell 15 in the same notebook⁶⁵), but the differences are unimportant for understanding the core concepts in this chapter.

Now, let's invoke our training loop:

```
# In[17]:
training_loop(
    n_epochs = 100,
    learning_rate = 1e-2,
    params = torch.tensor([1.0, 0.0]),
    t_u = t_u,
    t_c = t_c)

# Out[17]:
Epoch 1, Loss 1763.884644
    Params: tensor([-44.1730, -0.8260])
    Grad:   tensor([4517.2969,  82.6000])
Epoch 2, Loss 5802485.500000
    Params: tensor([2568.4014,  45.1637])
    Grad:   tensor([-261257.4219, -4598.9712])
Epoch 3, Loss 19408035840.000000
    Params: tensor([-148527.7344, -2616.3933])
    Grad:   tensor([15109614.0000, 266155.7188])
...
Epoch 10, Loss 90901154706620645225508955521810432.000000
    Params: tensor([3.2144e+17,  5.6621e+15])
    Grad:   tensor([-3.2700e+19, -5.7600e+17])
Epoch 11, Loss inf
    Params: tensor([-1.8590e+19, -3.2746e+17])
    Grad:   tensor([1.8912e+21,  3.3313e+19])
    tensor([-1.8590e+19, -3.2746e+17])
```

Wait, what happened? Our training process literally blew up, leading to losses becoming `inf`. This is a clear sign that `params` is receiving updates that are too large and their values start

oscillating back and forth, as each update overshoots, and the next over-corrects even more. The optimization process is unstable, it *diverges* instead of converging to a minimum. We want to see smaller and smaller updates to `params`, not larger, as shown in Figure-5.9.

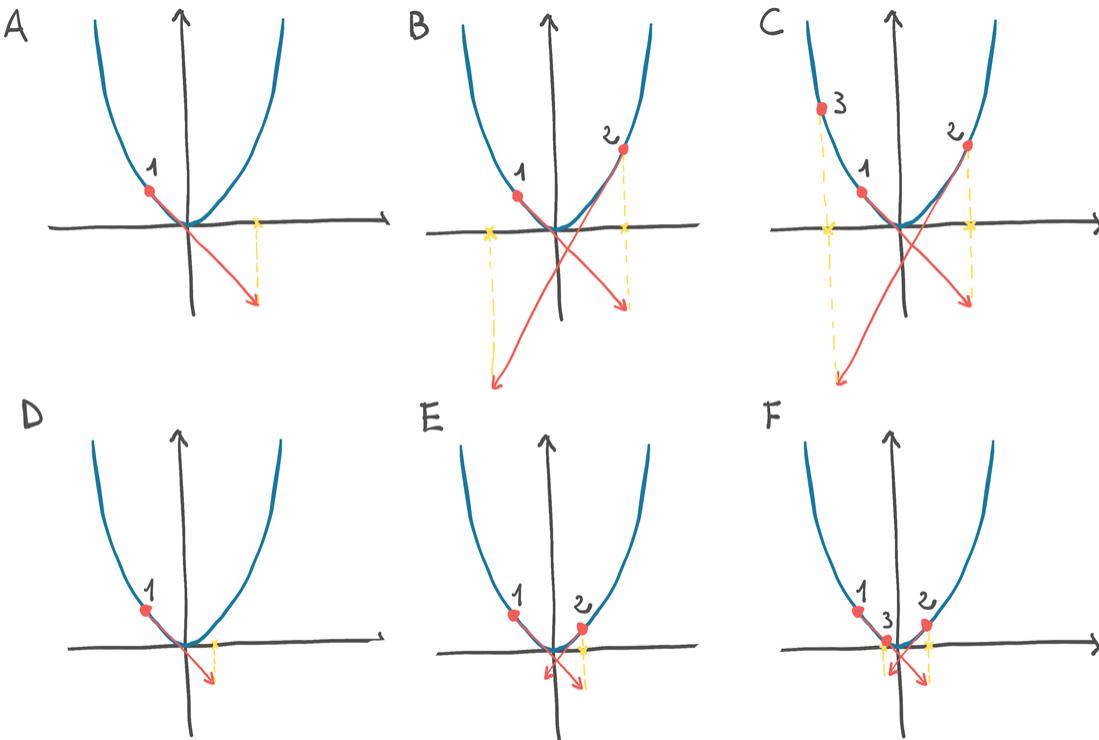


Figure 5.7 TOP: diverging optimization on convex function (parabola-like) due to large steps; **BOTTOM:** converging optimization with small steps.

How can we limit the magnitude of the `learning_rate * grad`? Well, that looks easy. We could simply choose a smaller `learning_rate`. We usually change learning rates by order of magnitude, so we might try with `1e-3` or `1e-4`, which would decrease the magnitude of updates by orders of magnitude. Let's go with `1e-4` and see how that works out.

```
# In[18]:
training_loop(
    n_epochs = 100,
    learning_rate = 1e-4,
    params = torch.tensor([1.0, 0.0]),
    t_u = t_u,
    t_c = t_c)

# Out[18]:
Epoch 1, Loss 1763.884644
Params: tensor([ 0.5483, -0.0083])
Grad:   tensor([4517.2969,   82.6000])
Epoch 2, Loss 323.090546
Params: tensor([ 0.3623, -0.0118])
Grad:   tensor([1859.5493,   35.7843])
Epoch 3, Loss 78.929634
Params: tensor([ 0.2858, -0.0135])
Grad:   tensor([765.4667,   16.5122])
...
Epoch 10, Loss 29.105242
Params: tensor([ 0.2324, -0.0166])
```

```

Grad: tensor([1.4803, 3.0544])
Epoch 11, Loss 29.104168
Params: tensor([ 0.2323, -0.0169])
Grad: tensor([0.5781, 3.0384])
...
Epoch 99, Loss 29.023582
Params: tensor([ 0.2327, -0.0435])
Grad: tensor([-0.0533, 3.0226])
Epoch 100, Loss 29.022669
Params: tensor([ 0.2327, -0.0438])
Grad: tensor([-0.0532, 3.0226])

tensor([ 0.2327, -0.0438])

```

Nice, the behavior is now stable, but there's another problem. Updates to parameters are very small, so the loss decreases very slowly and eventually stalls. We could obviate to this issue by making the `learning_rate` adaptive, that is, change according to the magnitude of updates. There are optimization schemes that do that; we'll see one towards the end of this Chapter, in section 5.2.1.

However, there's another potential trouble maker in the update term: the gradient itself. Let's go back and look at `grad` at epoch 1 during optimization.

We can see that the first-epoch gradient for the weight is about 50 times larger than the gradient for the bias. This means the weight and bias live in differently scaled spaces. If this is the case, a learning rate that's large enough to meaningfully update one will be so large as to be unstable for the other, or appropriate for the other won't be large enough to meaningfully change the first. That means we're not going to be able update our parameters unless we change something about our formulation of the problem. We could have individual learning rates for each parameter, but for models with many parameters this would be too much to bother with; it's babysitting of the kind we don't like.

There's a simpler way of keeping things in check, and that's to change the inputs so that the gradients aren't quite so different. We can make sure the range of the input doesn't get too far from the range of -1.0 to 1.0, roughly speaking. In our case we can achieve something close enough to that by simply multiplying `t_u` by 0.1:

```
# In[19]:
t_un = 0.1 * t_u
```

Here, we'll denote the normalized version of `t_u` by appending an `n` to the variable name. At this point we can run the training loop on our normalized input:

```

# In[20]:
training_loop(
    n_epochs = 100,
    learning_rate = 1e-2,
    params = torch.tensor([1.0, 0.0]),
    t_u = t_un, ❶
    t_c = t_c)

# Out[20]:
```

```

Epoch 1, Loss 80.364342
    Params: tensor([1.7761, 0.1064])
    Grad:   tensor([-77.6140, -10.6400])
Epoch 2, Loss 37.574917
    Params: tensor([2.0848, 0.1303])
    Grad:   tensor([-30.8623, -2.3864])
Epoch 3, Loss 30.871077
    Params: tensor([2.2094, 0.1217])
    Grad:   tensor([-12.4631, 0.8587])
...
Epoch 10, Loss 29.030487
    Params: tensor([ 2.3232, -0.0710])
    Grad:   tensor([-0.5355, 2.9295])
Epoch 11, Loss 28.941875
    Params: tensor([ 2.3284, -0.1003])
    Grad:   tensor([-0.5240, 2.9264])
...
Epoch 99, Loss 22.214186
    Params: tensor([ 2.7508, -2.4910])
    Grad:   tensor([-0.4453, 2.5208])
Epoch 100, Loss 22.148710
    Params: tensor([ 2.7553, -2.5162])
    Grad:   tensor([-0.4446, 2.5165])
tensor([ 2.7553, -2.5162])

```

- ① We've updated t_u to our new, re-scaled t_{un} .

Even though we set our learning rate back to $1e-2$, parameters didn't blow up during iterative updates. Let's take a look at the gradients: they were of similar magnitude, so using a single `learning_rate` for both parameters worked just fine. We could probably do a better job of normalization than a simple rescaling by a factor of ten, but since doing so is good enough for our needs, we're going to stick with just that for now.

NOTE

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

Let's run the loop for enough iterations to see changes in `params` get small. Let's change `n_epochs` to 5000.

```

# In[21]:
params = training_loop(
    n_epochs = 5000,
    learning_rate = 1e-2,
    params = torch.tensor([1.0, 0.0]),
    t_u = t_un,
    t_c = t_c,
    print_params = False)

params
# Out[21]:

```

```

Epoch 1, Loss 80.364342
Epoch 2, Loss 37.574917
Epoch 3, Loss 30.871077
...
Epoch 10, Loss 29.030487
Epoch 11, Loss 28.941875
...
Epoch 99, Loss 22.214186
Epoch 100, Loss 22.148710
...
Epoch 4000, Loss 2.927680
Epoch 5000, Loss 2.927648

tensor([ 5.3671, -17.3012])

```

Good: we saw our loss decreasing while we were changing parameters along the direction of gradient descent. It didn't go exactly to zero. This could mean that iterations were not enough to converge to zero, or that the data points are not exactly sitting on a line. As we anticipated, our measurements were not perfectly accurate, or there was noise involved in the reading.

But look: the value for `w` and `b` looks an awful lot like the numbers we need to use to convert Celsius to Fahrenheit (after accounting for our earlier normalization when we multiplied our inputs by 0.1). The exact values would be `w=5.5556` and `b=-17.7778`. Our fancy thermometer was showing temperatures in Fahrenheit the whole time. No big discovery, except that our gradient descent optimization process works!

Let's do something we should have done right from the start: plotting our data. We didn't do it here in the chapter until just for the sake of drama, for the surprise effect. But seriously, this is the one first thing anyone doing data science should do. Plot the heck out of the data, always.

```

# In[22]:
%matplotlib inline
from matplotlib import pyplot as plt

t_p = model(t_un, *params)      ❶

fig = plt.figure(dpi=600)
plt.xlabel("Fahrenheit")
plt.ylabel("Celsius")
plt.plot(t_u.numpy(), t_p.detach().numpy()) ❷
plt.plot(t_u.numpy(), t_c.numpy(), 'o')

```

- ❶ Remember that we're training on the normalized unknown units.
- ❷ But we're plotting the raw unknown values.

This code produces 5.8:

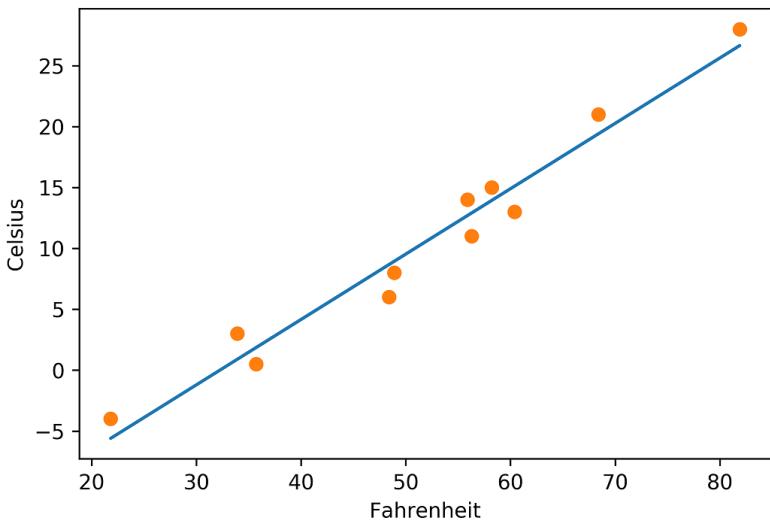


Figure 5.8 The plot of our linear-fit model (solid line) vs. our input data (circles).

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

5.2 PyTorch's Autograd: Back-propagate all things

In our little adventure so far, we just saw a simple example of back-propagation: we computed the gradient of a composition of functions - the model and the loss - with respect to their inner-most parameters - `w` and `b` - by propagating derivatives backwards using the *chain rule*. The basic requirement here is that all functions we're dealing with are differentiable analytically. If this is the case, we can compute the gradient, what we called earlier "the rate of change of the loss", with respect to the parameters, in one sweep.

Should we have a complicated model with millions of parameters, as long as our model is differentiable, computing the gradient of the loss with respect to parameters amounts to writing the analytical expression for the derivatives and evaluate them *once*. Granted, writing the analytical expression for the derivatives of a very deep composition of linear and non-linear functions is not a lot of fun.⁶⁶ It isn't particularly quick, either.

This is when PyTorch tensors come to the rescue, with a PyTorch component called autograd. In chapter 3 we had a comprehensive overview of what tensors are and what functions we can call on them. We left out one very interesting aspect, however: PyTorch tensors can remember where they come from, in terms of the operations and parent tensors that originated them, and they can provide the chain of derivatives of such operations with respect to their inputs automatically. This means that we won't need to derive our model by hand⁶⁷; given a forward expression, no matter how nested, PyTorch will provide the gradient of that expression with respect to its input parameters automatically.

At this point, the best way to proceed is to rewrite our thermometer calibration code, this time leveraging autograd, and see what happens. First we recall our model and loss function:

Listing 5.3 code/p1ch5/2_autograd.ipynb

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

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

Let's again initialize a parameters tensor:

```
# 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 will have `params` as an ancestor will have access to the chain of functions that were called to get from `params` to that tensor. In case these functions are differentiable (and most PyTorch tensor operations will be), the value of the derivative will be automatically populated as a `grad` attribute of the `params` tensor.

In general, all PyTorch tensors have an attribute named `grad`. Normally, it's `None`:

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

# Out[6]:
True
```

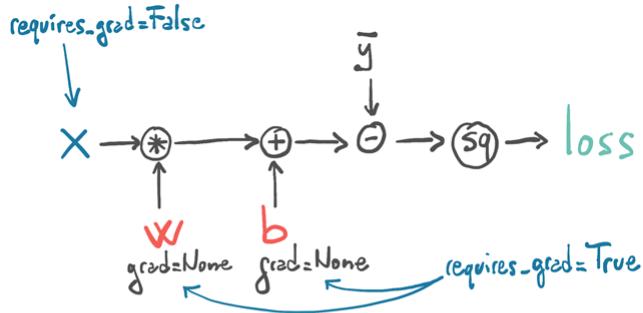
All we have to do to populate it is to start with a tensor with `requires_grad` set to `True`, then call the model and compute the loss, and then call `backward` on the `loss` tensor:

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

params.grad

# Out[7]:
tensor([4517.2969, 82.6000])
```

At this point, the `grad` attribute of `params` contains the derivatives of the loss with respect to each element of `params`.



`loss.backward()`

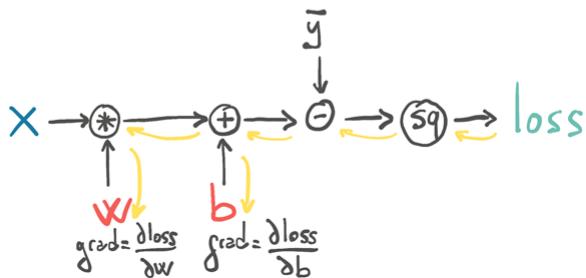


Figure 5.9 The forward graph and backward graph of model as computed with autograd.

We 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 where PyTorch newcomers - and a lot of more experienced folks too - trip up on regularly. We just wrote *accumulate*, not *store*.

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

Let's repeat together: calling `backward` will lead derivatives to *accumulate* at leaf nodes. So if `backward` has been called earlier, the loss is evaluated again and `backward` is called again (as in any training loop), the gradient at each leaf will be accumulated (i.e. summed) on top of the one computed at the previous iteration, which will lead to an incorrect value for the gradient.

In order to prevent this from occurring, we need to **zero the gradient explicitly** at each iteration. We can do it easily using the in-place `zero_` method:

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

NOTE

You might be curious why zeroing the gradient is a required step instead of doing it automatically whenever we call `backward`. It's to provide more flexibility and control when working with gradients in complicated models.

Having this reminder drilled into our heads, let's see how our 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:      ❶  
            params.grad.zero_()  
  
        t_p = model(t_u, *params)  
        loss = loss_fn(t_p, t_c)  
        loss.backward()  
  
        with torch.no_grad():          ❷  
            params -= learning_rate * params.grad  
  
        if epoch % 500 == 0:  
            print('Epoch %d, Loss %f' % (epoch, float(loss)))  
  
    return params
```

- ❶ This could be done at any point in the loop prior to calling `loss.backward()`
- ❷ This is a somewhat cumbersome bit of code, but as we'll see in 5.2.1, it's not an issue in practice.

You'll note that when our code updating `params` is not quite as straightforward as we might have expected. There are two particularities. First, we are encapsulating the update in a `no_grad` context using the Python `with` statement. This means that within, the PyTorch autograd mechanism should *look away*⁶⁸, i.e. not add edges to the forward graph in Figure-5.11. In fact, when we are executing this bit of code, the forward graph that PyTorch recorded has been consumed when we called `backward`, leaving us with the `params` leaf node. But now we want to change this leaf node before we start out building a fresh forward graph on top of it. While this use case is usually wrapped inside the optimizers we see in 5.2.1, we will take a closer look when we see another common use of `no_grad` in 5.2.3.

Secondly, we update `params` in-place. This means that we keep the same tensor `params` around, but subtract our update from it. When using autograd, we usually avoid in-place updates because PyTorch's autograd engine might need the values we would be modifying for the backward. Here, however, we are operating without autograd, and in fact it is beneficial to keep the `params` tensor around. Not replacing the parameters by assigning new tensors to their variable name will become crucial when we register our parameters with the optimizer in 5.2.1.

Let's see if it works:

```

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

```

- ❶ Adding this `requires_grad=True` is key.
- ❷ Note that again, we're using the normalized `t_un` instead of `t_u`.

Same result as we got previously. Good for us! It means that while we are *capable* of computing derivatives by hand, we no longer need to.

5.2.1 Optimizers *a-la Carte*

In this code, we used *vanilla* gradient descent for optimization, which worked fine for our simple case. Needless to say, there are several optimization strategies and tricks that can help convergence, especially when models get complicated.

We'll dive deeper into this topic in later Chapters, but now it's the right time to introduce the way PyTorch abstracts the optimization strategy away from user code, i.e. the training loop above. This saves us from the boilerplate busywork of having to update each and every parameter to our model ourselves. The `torch` module has an `optim` submodule where we can find classes implementing different optimization algorithms. Here's an abridged listing:

Listing 5.4 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-5.12.

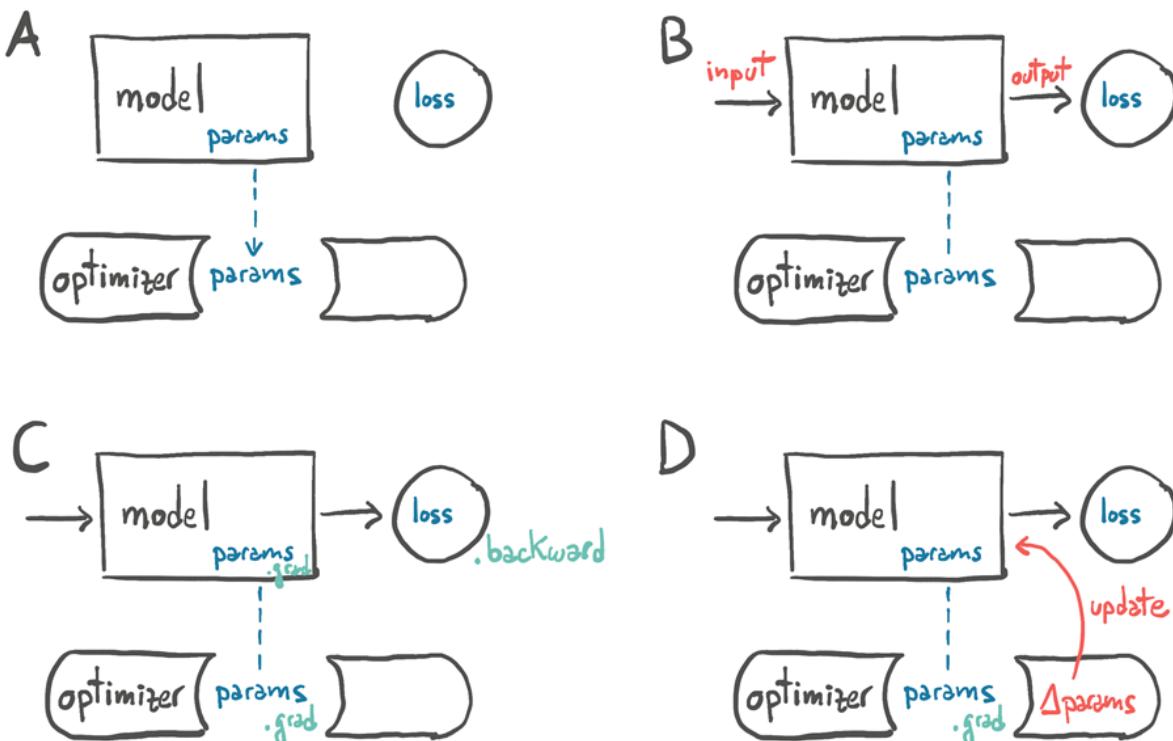


Figure 5.10 Conceptual representation of how an optimizer holds a reference to parameters (A), and after a loss is computed from inputs (B), a call to `.backward` leads to `.grad` being populated on parameters ©. At that point, the optimizer can access `.grad` to compute the parameter updates (D).

Each optimizer exposes two methods: `zero_grad` and `step`. The former zeroes 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.

Let's create `params` and instantiate a gradient descent optimizer:

```
# In[6]:  
params = torch.tensor([1.0, 0.0], requires_grad=True)  
learning_rate = 1e-5  
optimizer = optim.SGD([params], lr=learning_rate)
```

Here SGD stands for Stochastic Gradient Descent. Actually, the optimizer itself is exactly a vanilla Gradient Descent (as long as the `momentum` argument is set to `0.0`, which is the default). The terms *stochastic* comes from the fact that the gradient is typically obtained by averaging over a random subset of all input samples, called a *minibatch*. However, the optimizer itself does not know if the loss was evaluated on all the samples (vanilla) or a random subset thereof (stochastic), so the algorithm is literally the same in the two cases.

Anyway, let's take our fancy new optimizer for a spin:

```
# In[7]:  
t_p = model(t_u, *params)  
loss = loss_fn(t_p, t_c)  
loss.backward()  
  
optimizer.step()  
  
params  
  
# Out[7]:  
tensor([ 9.5483e-01, -8.2600e-04], requires_grad=True)
```

The value of `params` was updated upon calling `step` without us having to touch it ourselves! What happened is that the optimizer looked into `params.grad` and updated `params` subtracting `learning_rate` times `grad` from it, exactly as in our former hand-rolled code.

Ready to stick this code in a training loop? Nope! Big gotcha almost got us — we forgot to zero out the gradients. Had we called the above code in a loop, gradients would have accumulated in the leaves at every call to `backward` and our gradient descent would have been all over the place! Here's the loop-ready code, with the extra `zero_grad` at the right spot (right before the call to `backward`):

```
# In[8]:  
params = torch.tensor([1.0, 0.0], requires_grad=True)  
learning_rate = 1e-2  
optimizer = optim.SGD([params], lr=learning_rate)  
  
t_p = model(t_un, *params)  
loss = loss_fn(t_p, t_c)  
  
optimizer.zero_grad() ❶
```

```

loss.backward()
optimizer.step()

params

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

```

- ① As before, the exact placement of this call is somewhat arbitrary. It could be earlier in the loop as well.

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

Let's update our 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) ①

training_loop(
    n_epochs = 5000,
    optimizer = optimizer,
    params = params, ①
    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)

```

- ① It's important that both `params` here are the same object; otherwise the optimizer won't know what parameters were used by the model.

Again, same result as before. Great, it means further confirmation we know how to descend a gradient by hand! In order to test more optimizers, all we have to do is instantiate a different optimizer, say Adam, instead of SGD. The rest of the code stays as it is. Pretty handy stuff.

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

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

# 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)
```

- ❶ New optimizer class here.
- ❷ Note that we're back to the original t_u as our input.

The optimizer is not the only flexible part of our training loop. Let's turn our attention on the model. In order to train a neural network on the same data and the same loss, all we would need to change is the `model` function. It wouldn't make particular sense in this case, since we know that converting Celsius to Fahrenheit amounts to a linear transformation, but we'll do it anyway in chapter 6, 6.2.1. We'll see quite soon that neural networks allow us to remove our arbitrary assumptions about the shape of the function we should be approximating. Even so, we'll see how neural networks manage to be trained even when the underlying processes are highly non-linear (such in the case of describing an image with a sentence, as we've seen in Chapter 2).

By now, we have touched upon a lot of the essential concepts that will enable us to train complicated deep learning models, while knowing what's going on under the hood: back propagation to estimate gradients, autograd, optimizing weights of models using gradient descent or other optimizers. Really, there isn't a whole lot more. The rest is mostly filling in the blanks, however extensive they are.

Next up we're going to have an aside about how to split up our samples because that sets up a perfect use case for learning how to better control autograd.

5.2.2 Training, Validation, and Overfitting

There's one last thing Johannes Kepler taught us that we didn't follow so far, remember? He kept a part of the data on the side so that he could validate his models on independent observations. This is a vital thing to do, especially when the model we adopt could potentially approximate functions of any shape, as in the case of neural networks. In other words, a highly adaptable model will tend to use its many parameters to make sure the loss is minimal *at* the data points, but we'll have no guarantee that the model behaves well *away* or *in between* the data points. After all, that's all we're asking the optimizer to do: minimize the loss *at* the data points. Sure enough, if we had independent data points that we didn't use to evaluate our loss or descend along its negative gradient, we would soon find out that evaluating the loss at those independent data points would yield higher than expected loss. We have already mentioned this phenomenon, called *overfitting*.

The first action we can take to combat overfitting is recognizing that it might happen. In order to do so, as Kepler figured out in 1600, we must take out a few data points from our dataset (the *validation set*) and only fit our model on the remaining data points (the *training set*), as in Figure-5.13. Then, while we're fitting the model, we can evaluate the loss once on the training set and once on the validation set. When we're trying to decide if we've done a good job of fitting our model to the data, we must look at each!

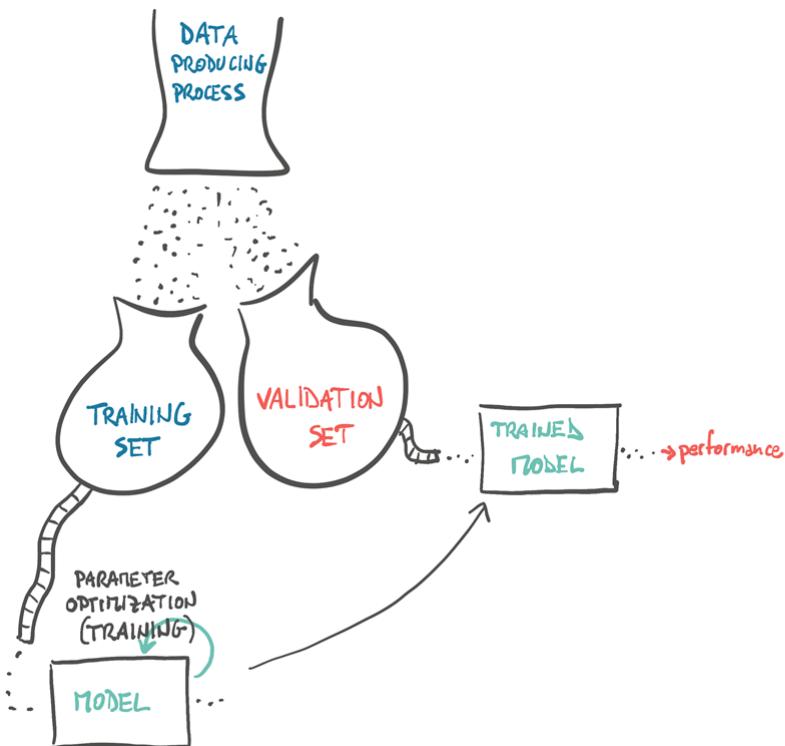


Figure 5.11 Conceptual representation of a data-producing process and the collection and use of training data and independent validation data.

The first figure, the training loss, will tell us if our model can fit the training set at all - in other

words, if our model has enough *capacity* to process the relevant information in the data. If our mysterious thermometer somehow managed to measure temperatures using a logarithmic scale, our poor linear model would not have had a chance to fit those measurements and provide us with a sensible conversion to Celsius. In that case, our training loss (the loss we were printing in the training loop) would stop decreasing well before approaching zero.

A deep neural network can potentially approximate complicated functions, provided that the number of neurons, and therefore parameters, is high enough. The fewer the number of parameters, the simpler the shape of the function our network will be able to approximate. So, rule one: if the training loss is not decreasing, chances are that the model is too simple for the data. The other possibility is that our data just doesn't contain meaningful information for it to explain the output: if the nice folks at the shop sold us a barometer instead of a thermometer, we would have little chance to predict temperature in Celsius from just pressure, even if we used the latest neural network architecture from Quebec.⁶⁹

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

Let's delve into this phenomenon a little, going back to our thermometer example. We could have decided to fit the data with a more complicated function, like a piecewise polynomial, or a really large neural network. It could generate a model meandering its way through the data points, like in Figure-5.14, just because it pushes the loss very close to zero. Since the behavior of the function away from the data points does not increase the loss, there's nothing to keep the model away from the training data points in check.

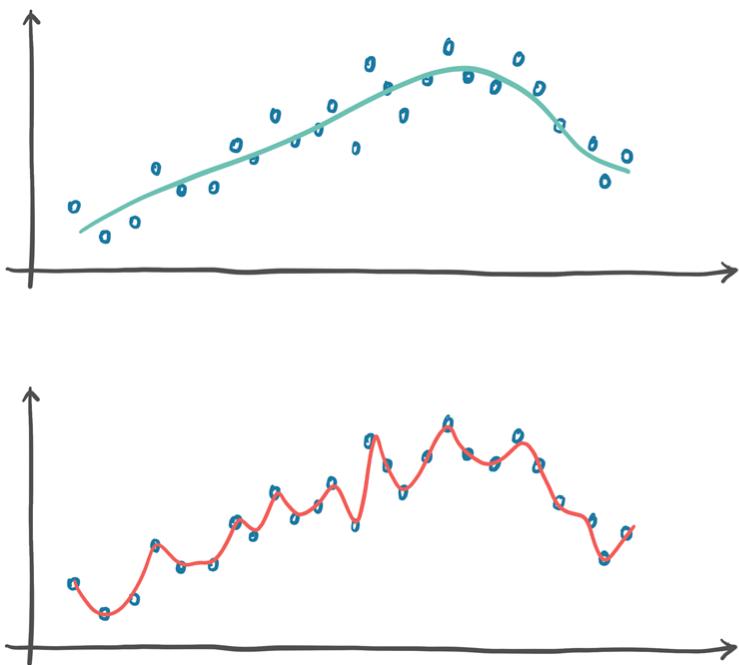


Figure 5.12 Rather extreme example of overfitting.

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

Assuming we have enough data points, we should make sure the model that is capable of fitting the training data is as regular as possible in-between them. There are several ways to achieve this. One is adding so-called penalization terms to the loss function, to make it cheaper for the model to behave more smoothly, to change more slowly (up to a point). Another is to add noise to the input samples, to artificially create new data points in-between training data samples and force the model to try to fit those too. There are several other ways, all of them somewhat related to the above. But the best favor we can do to ourselves, at least as a first move, is to make our model simpler. From an intuitive standpoint, a simpler model may not fit the training data as perfectly as a more complicated model would do, but it will likely behave more regularly in-between data points.

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

We'll see more about this in chapter 12, 12.4.2 - we'll discover that our life will be a balancing act between fitting and overfitting. For now, let's get back to our example and see how we can split the data into a training and a validation set. We'll do it by shuffling t_u and t_c in the same

way, and then splitting the resulting shuffled tensors in two parts.

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

```
# In[12]:
n_samples = t_u.shape[0]
n_val = int(0.2 * n_samples)

shuffled_indices = torch.randperm(n_samples)

train_indices = shuffled_indices[:-n_val]
val_indices = shuffled_indices[-n_val:]

train_indices, val_indices ①

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

- ① Since these are random, don't be surprised if your values end up different from here on out.

We just got index tensors that we can use to build training and validation sets starting from the data tensors

```
# In[13]:
train_t_u = t_u[train_indices]
train_t_c = t_c[train_indices]

val_t_u = t_u[val_indices]
val_t_c = t_c[val_indices]

train_t_un = 0.1 * train_t_u
val_t_un = 0.1 * val_t_u
```

Our training loop doesn't really change. We just want to additionally evaluate the validation loss at every epoch, to have a chance to recognize whether we're over-fitting:

```
# In[14]:
def training_loop(n_epochs, optimizer, params, train_t_u, val_t_u, train_t_c, val_t_c):
    for epoch in range(1, n_epochs + 1):
        train_t_p = model(train_t_u, *params) ①
        train_loss = loss_fn(train_t_p, train_t_c)

        val_t_p = model(val_t_u, *params) ①
        val_loss = loss_fn(val_t_p, val_t_c)

        optimizer.zero_grad() ②
        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
```

- ① these two pairs of lines are the same except for the `train_*` vs. `val_*` inputs.

- ② Note that there is no `val_loss.backward()` here, since we don't want to train the model on the validation data.

```
# 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, ①
    val_t_u = val_t_un, ①
    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)
```

- ① Since we're using SGD again, we're back to using normalized inputs.

Here we are not being entirely fair to our model. The validation set is really small, so the validation loss will only be meaningful up to a point. In any case we note that the validation loss is higher than our training loss, although not by an order of magnitude. That a model performs better on the training set is expected, since the model parameters are being shaped by the training set. Our main goal is to also see both the training loss *and* the validation loss decreasing. While ideally both losses would be roughly the same value, as long as validation loss stays reasonably close to the training loss, we know that our model is continuing to learn generalized things about our data. Looking at Figure-5.15, case C is ideal, while D is acceptable. In case A, the model isn't learning at all, and in case B, we see over-fitting. We'll see more meaningful examples of overfitting in chapter 12, 12.4.2.

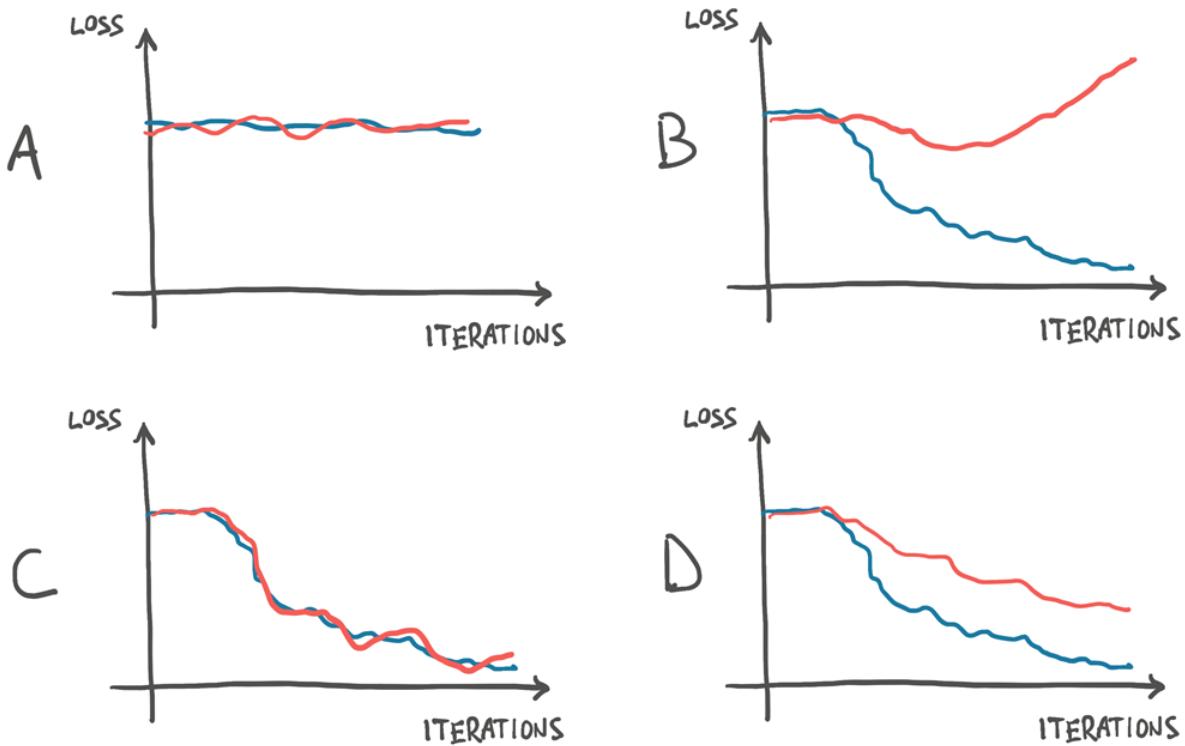


Figure 5.13 Overfitting scenarios when looking at the training (blue) and validation (red) losses. A: training and validation losses do not decrease, the model is not learning due to no information in the data or insufficient capacity of the model. B: training loss decreases while validation increases, overfitting. C: training and validation losses decrease exactly in tandem, performance may be improved further as model is not at the limit of overfitting. D: training and validation losses have different absolute values but similar trends, overfitting is under control.

5.2.3 Autograd Nits and Switching it Off

From the training loop above we can appreciate that we only ever call `backward` on the `train_loss`. Therefore, errors will only ever back-propagate 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 the other on `val_t_u`, then `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 for us, this isn't the case. The first line in the training loop evaluates `model` on `train_t_u` to produce `train_t_p`. Then `train_loss` is evaluated from `train_t_p`. This creates a computation graph that links `train_t_u` to `train_t_p` to `train_loss`. When `model` is evaluated again on `val_t_u`, it produces `val_t_p` and `val_loss`. In this case, a separate

computation graph will be created that links `val_t_u` to `val_t_p` to `val_loss`. Separate tensors have been run through the same functions, `model` and `loss_fn`, generating separate computation graphs, as shown in Figure-5.16.

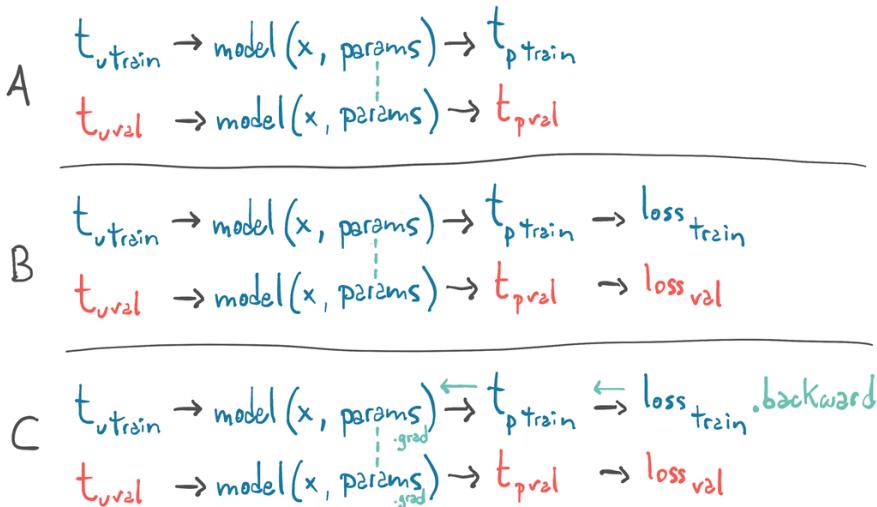


Figure 5.14 Diagram showing how gradients propagate through a graph with two losses when `.backward` is called on one of them.

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

If we (incorrectly) called `backward` on `val_loss` as well, we would 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 we called `backward`, unless we zeroed out gradients explicitly? Well, here something very similar would happen: calling `backward` on `val_loss` would lead to gradients accumulating in the `params` tensor, on top of those generated during the `train_loss.backward()` call. In this case, we would effectively train our model on the whole dataset (both training and validation), since the gradient would depend on both. Pretty interesting.

There's another element for discussion here. Since we're not ever calling `backward` on `val_loss`, why are we building the graph in the first place? We could in fact just call `model` and `loss_fn` as plain functions, without tracking history. However optimized, tracking history comes with additional costs that we could totally forego during the validation pass, especially when the model has millions of parameters.

In order to address this, PyTorch allows us to switch off autograd when we don't need it using the `torch.no_grad` context manager. We won't see any meaningful advantage in terms of speed or memory consumption on our small problem. However, for larger models the differences can

add up. We can make sure this works by checking the value of the `requires_grad` attribute on the `val_loss` tensor:

```
# In[16]:
def training_loop(n_epochs, optimizer, params, train_t_u, val_t_u, train_t_c, val_t_c):
    for epoch in range(1, n_epochs + 1):
        train_t_p = model(train_t_u, *params)
        train_loss = loss_fn(train_t_p, train_t_c)

        with torch.no_grad(): ①
            val_t_p = model(val_t_u, *params)
            val_loss = loss_fn(val_t_p, val_t_c)
            assert val_loss.requires_grad == False ②

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

- ① Context manager here.
- ② We can check that here that our outputs `requires_grad` args are forced to `False` inside this block.⁷⁰

Using the related `set_grad_enabled` context, we can also condition code to run with `autograd` enabled or disabled, according to a boolean expression - typically indicating whether we are running in training or inference. We could for instance 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
```

5.3 Conclusion

We started this chapter with a big question: how is it that a machine can learn from examples? We spent the rest of the chapter describing the mechanism with which a model can be optimized to fit data. We chose to stick with a simple model in order to see all the moving parts without unneeded complications.

So we've had our fill of appetizers, in chapter 6 we'll finally get to the main course: Using A Neural Network To Fit Our Data. We'll be working on solving the same thermometer problem, but with the more powerful tools provided by the `torch.nn` module. We'll adopt the same spirit of using this small problem to illustrate the larger uses of PyTorch. The problem doesn't need a neural network to reach a solution, but it will allow us to develop a simpler understanding of what's needed to train a neural network.

5.4 Exercises

- Redefine the model to be $w2 * t_u ** 2 + w1 * t_u + b$
 - What parts of the training loop, etc. needed to change to accommodate this redefinition?
 - What parts were agnostic to swapping out the model?
 - Is the resulting loss higher or lower after training?
 - Is the actual result better or worse?

5.5 Summary

- Linear models are the simplest reasonable model to use to fit data.
- Convex optimization techniques can be used for linear models, but they do not generalize to neural networks, so we focus on stochastic gradient descent for parameter estimation.
- Deep learning can be used for generic models that are not engineered for solving a specific task, but instead can be automatically adapted to specialize themselves on the problem at hand.
- Learning algorithms amount to optimizing parameters of models based on observations. 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 allows users to rely on the dynamic computation graph during complex forward passes.
- Context managers like `with torch.no_grad():` can be used to control autograd behavior.
- Data is often split into separate sets of training samples and validation samples. This allows a model to be evaluated on data it was not trained on.
- Overfitting a model happens when the model's performance continues to improve on the training set, but degrade on the validation set. This is usually due to the model not generalizing, and instead memorizing the desired outputs for the training set.

6

Using A Neural Network To Fit the Data

This chapter covers:

- The use of non-linear activation functions as the key difference from linear models
- The many different 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

So far we took a close look on how a linear model can learn and how to make it happen in PyTorch. We have focused on a very simple regression problem, that only required us to use a linear model with one input and one output. Such a simple example allowed us to dissect the mechanics of a model that learns without getting overly distracted with the implementation of the model itself. As we saw in the overview diagram in Chapter 5 (repeated here as Figure 6.1) Figure-5.2, the exact details of the model are not needed to understand the high-level process that trains a model. Back-propagating error to the parameters, 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.

THE LEARNING PROCESS

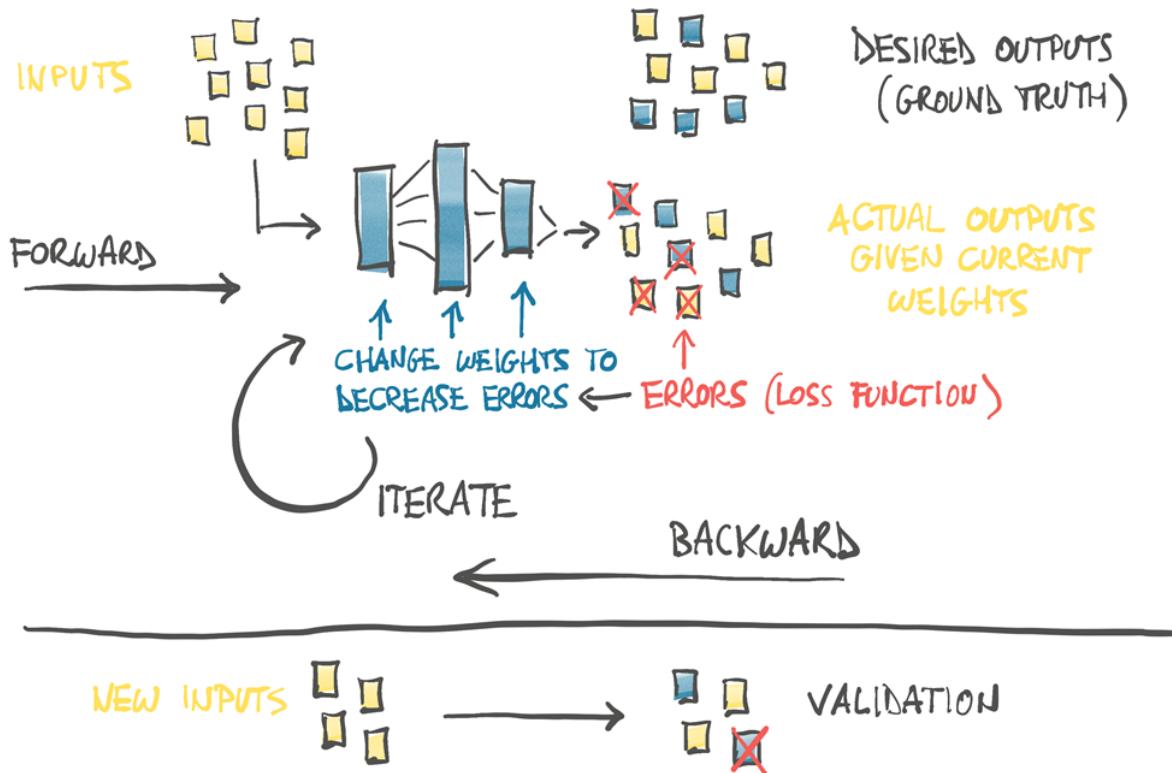


Figure 6.1 Our mental model of the learning process, as implemented in chapter 5.

In this chapter we are only going to be making changes to our model architecture. We're going to implement a full artificial neural network to solve our problem.

Our thermometer conversion training loop and our Fahrenheit-to-Celsius samples split into training and validation sets from last chapter will remain. We could start to use a quadratic model; re-writing our `model` as a quadratic function of its input (e.g. $y = a * x^{**2} + b * x + c$). Since such a model would be differentiable, PyTorch would take care of computing gradients and the training loop would work as usual. That wouldn't be too interesting for us, though, because we would still be fixing the shape of the function.

This is the chapter where we start to hook the foundational work we've put in together with the PyTorch features you'll be using day-in, 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 it just being so much black magic.

Before we get into the implementation of our new model, though, let's cover what we mean by *artificial neural network*.

6.1 Artificial Neurons

At the core of deep learning are neural networks, mathematical entities capable of representing complicated functions through a composition of simpler functions. The term *neural network* is obviously suggestive of a link to the way our brain works. As a matter of fact, although the initial models were inspired by neuroscience⁷¹, modern artificial neural networks bear only vague resemblance to the mechanisms of neurons in the brain. It seems likely that both artificial and physiological neural networks use vaguely similar mathematical strategies for approximating complicated functions because that family of strategies work very effectively.

NOTE

We are going to drop the "artificial" and refer to these constructs as just "neural networks" from here forward.

The basic building block of these complicated functions is the *neuron*, as pictured in Figure-6.2. At its core, it is nothing but a linear transformation of the input (e.g. multiplication of the input by a number, the *weight*, and addition of a constant, the *bias*) followed by the application of a fixed non-linear function (referred to as the "activation function").

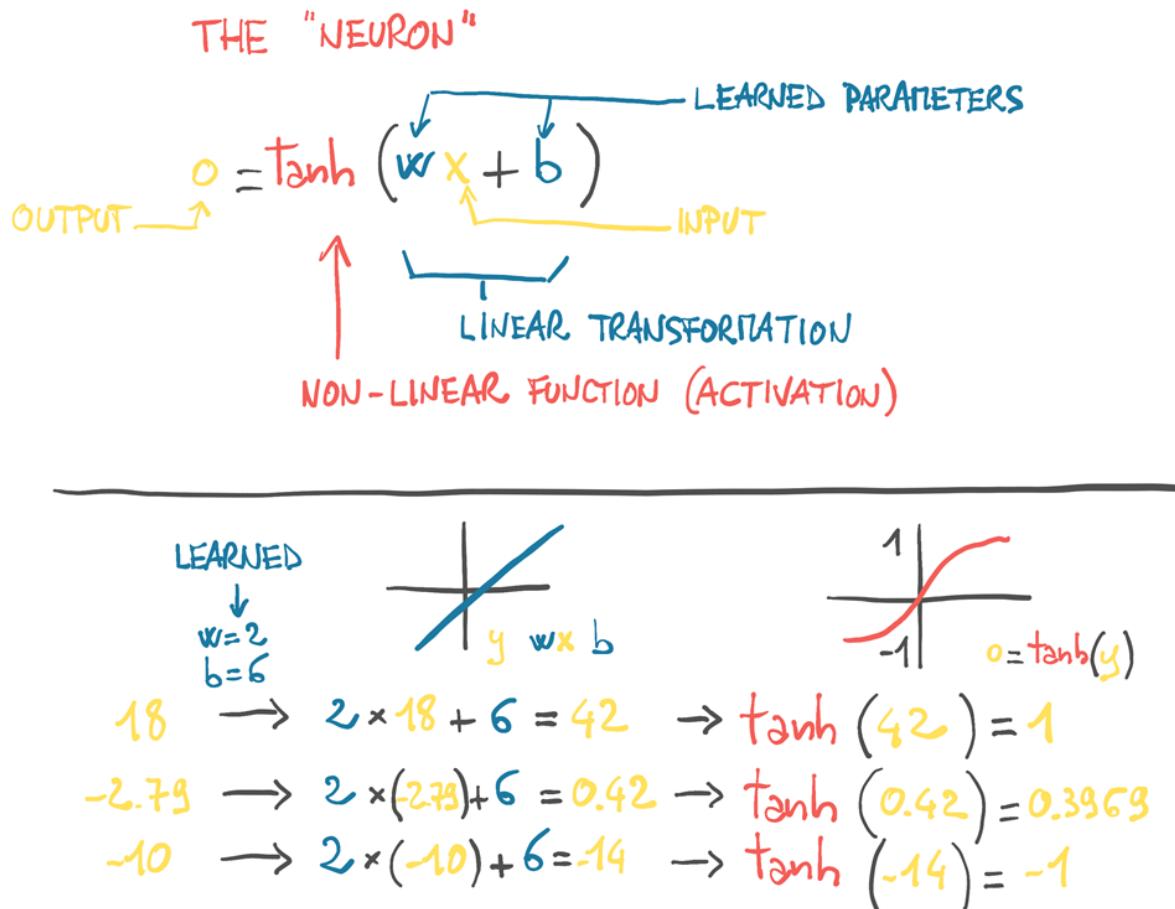


Figure 6.2 An artificial neuron: a linear transformation enclosed in a non-linear function.

Mathematically we can write this out as $o = f(w * x + b)$, with x as our input, w our weight or scaling factor, and b is our bias or offset. f is our activation function, set to the hyperbolic tangent, or "tanh" function here. In general, x and hence o can be simple scalars, or vector-valued (meaning holding many scalar values), and similarly w can be a single scalar, or a matrix, while b is a scalar or vector (the dimensionality of the inputs and weights must match, however). In the latter case, the expression above is referred to as a *layer* of neurons, since it represents many neurons via the multi-dimensional weights and biases.

A multi-layer neural network, as represented in Figure-6.3, is made up by a composition of the above functions, that is

```

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.

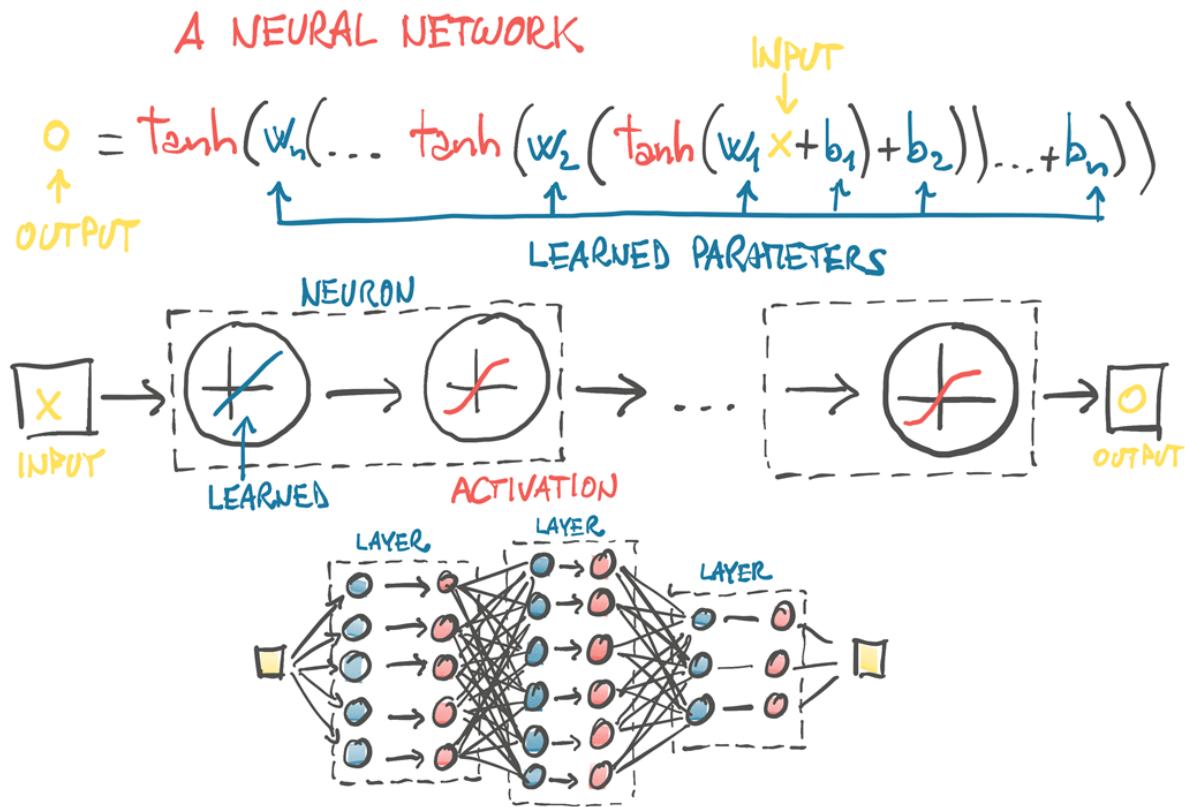


Figure 6.3 A neural network with three layers.

An important difference between our earlier linear model and what we'll actually be using for deep learning is the shape of the error function. Our linear model and error-squared loss function had a convex error curve, with a singular, clearly defined minimum. If we were to use other

methods, we could solve for it automatically and definitively. That means that our parameter updates were attempting to *estimate* that singular correct answer as best we could.

Neural networks do not have that same property of a convex error surface, even when using the same error-squared loss function! There's no singular right answer for each parameter that we're attempting to approximate. Instead, we are trying to get all of the parameters, when acting *in concert*, to produce a useful output. Since that useful output is only going to *approximate* the truth, there will be some level of imperfection. Where and how 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 results in neural network training looking very much like parameter estimation from a mechanical perspective, but we must remember that the theoretical underpinnings are actually quite different.

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

6.1.1 All We Need is Activation

As we have seen above, the simplest unit in (deep) neural networks is a linear operation (scaling + offset) followed by an activation function. We already had our linear operation in our latest model — the linear operation *was* the entire model. The activation function has the role of concentrating the outputs of the preceding linear operation into a given range.

Let's talk about what that means. Pretend that we're assigning a "good doggo" score to images. Pictures of retrievers and spaniels should have a high score, while images of airplanes and garbage trucks should have a low score. Bear pictures should have a low-ish score too, though higher than garbage trucks.

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

What we want to do is firmly constrain the output of our 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 just cap the output values. Anything below zero is set to zero, and anything above 10 is set to 10. That's a simple activation function called `torch.nn.Hardtanh`⁷².

Another family of functions that work well is `torch.nn.Sigmoid`, which is $1 / (1 + e^{-x})$, `torch.tanh`, and others that we'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 have a mostly-constant slope at $x == 0$. Conceptually, functions shaped this way work well because it means that there's an area in the middle of our linear function's output that our neuron (which again is just a linear function followed by an activation) will be sensitive to, while everything else gets lumped up next to the boundary values. As we can see in Figure-6.4, our garbage truck gets a score of -0.97 , while bears and foxes and wolves might end up somewhere in the -0.3 to 0.3 range.

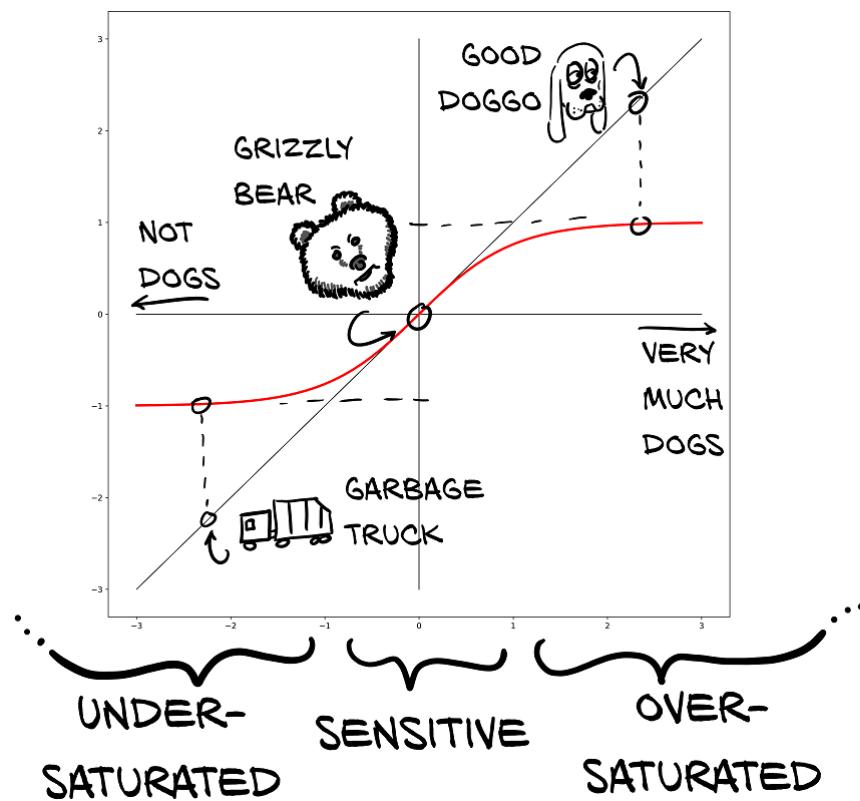


Figure 6.4 Dogs, bears, and garbage trucks being mapped to "how dog-like" via the tanh activation function.

This results in garbage trucks being flagged as "not dogs," our good dog mapping to "clearly a dog" and our bear ending up somewhere in the middle. In code, we can see the exact values:

```
>>> import math
>>> math.tanh(-2.2)      ①
-0.9757431300314515
>>> math.tanh(0.1)       ②
0.09966799462495582
>>> math.tanh(2.5)       ③
0.9866142981514303
```

- ① Garbage truck
- ② Bear

③ Good doggo

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

There are quite a few activation functions, some of which are pictured in Figure-6.5. In the first column, we see the smooth functions `Tanh` and `Softplus`, while second column has "hard" versions of the activation functions to their left, `Hardtanh` and `ReLU`. `ReLU`, or "Rectified Linear Unit" deserves special note, as it is currently considered 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 function, was widely used in early deep learning work, but has since 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 shown here with slope 0.1 for clarity).

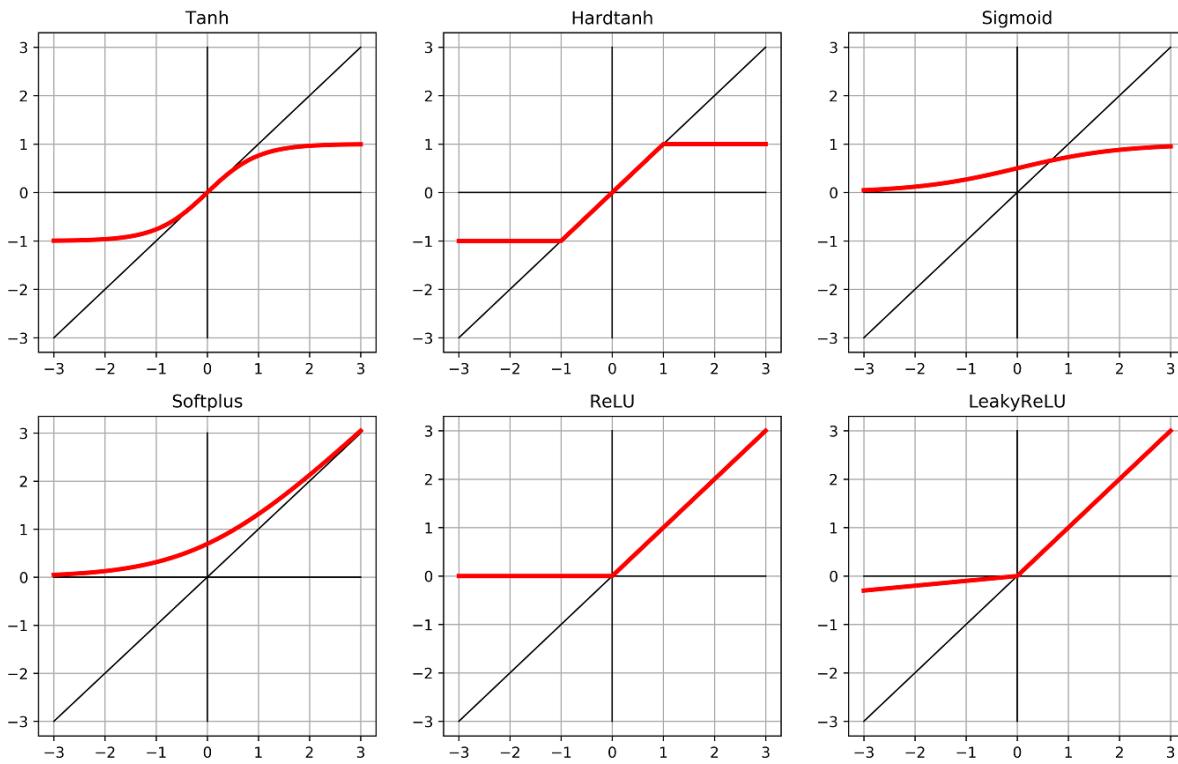


Figure 6.5 A collection of common and not-so-common activation functions.

Activation functions are curious, because with such a wide variety of proven successful ones (many more than pictured above), it's clear that there are few, if any, strict requirements. As

such, we're going to discuss some generalities about activation functions that can probably be trivially disproven in the specific. That said, by definition, activation functions:⁷³

- Are non-linear. Repeated applications of $w \cdot x + b$ without an activation function results in a function of the same (affin linear) form. The non-linearity allows the overall network to approximate more complex functions.
- Are differentiable, so that gradients can be computed through them. Point discontinuities, as we can see in Hardtanh or ReLU, are fine.

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

Also generally true (though less so) is that the functions:

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

- 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 we know about how back-propagation works, we can figure out that the errors will propagate backwards through the activation more effectively when the inputs are in the response range, while errors will not greatly affect neurons for which the input is saturated (since the gradient will be close to zero, due to the flat area around the output).

All put together, this results in a pretty powerful mechanism: 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 will respond in different ranges for the same inputs and b) the errors associated to 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 we have seen previously.

We are starting to get a deeper intuition for how joining many linear + activation units in parallel and stacking them one after the other leads us to a mathematical object that is capable of approximating complicated functions. Different combinations of units will respond to inputs in

different ranges, and for those parameters are relatively easy to optimize through gradient descent, since learning will behave a lot like that of a linear function until the output saturates.

6.1.2 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 non-linear processes and whose parameters we can estimate surprisingly well through gradient descent. This remains true, even when dealing with models with millions of parameters. What makes using deep neural networks so attractive is that it allows us not to worry too much about the exact function that represents our data - whether it is quadratic, piecewise polynomial, or something else. With a deep neural network model we have a universal approximator and a method to estimate its parameters. This approximator can be customized to our needs, in terms of model capacity and its ability to model complicated input/output relationships, just by composing simple building blocks. We can see some examples of this in Figure-6.6.

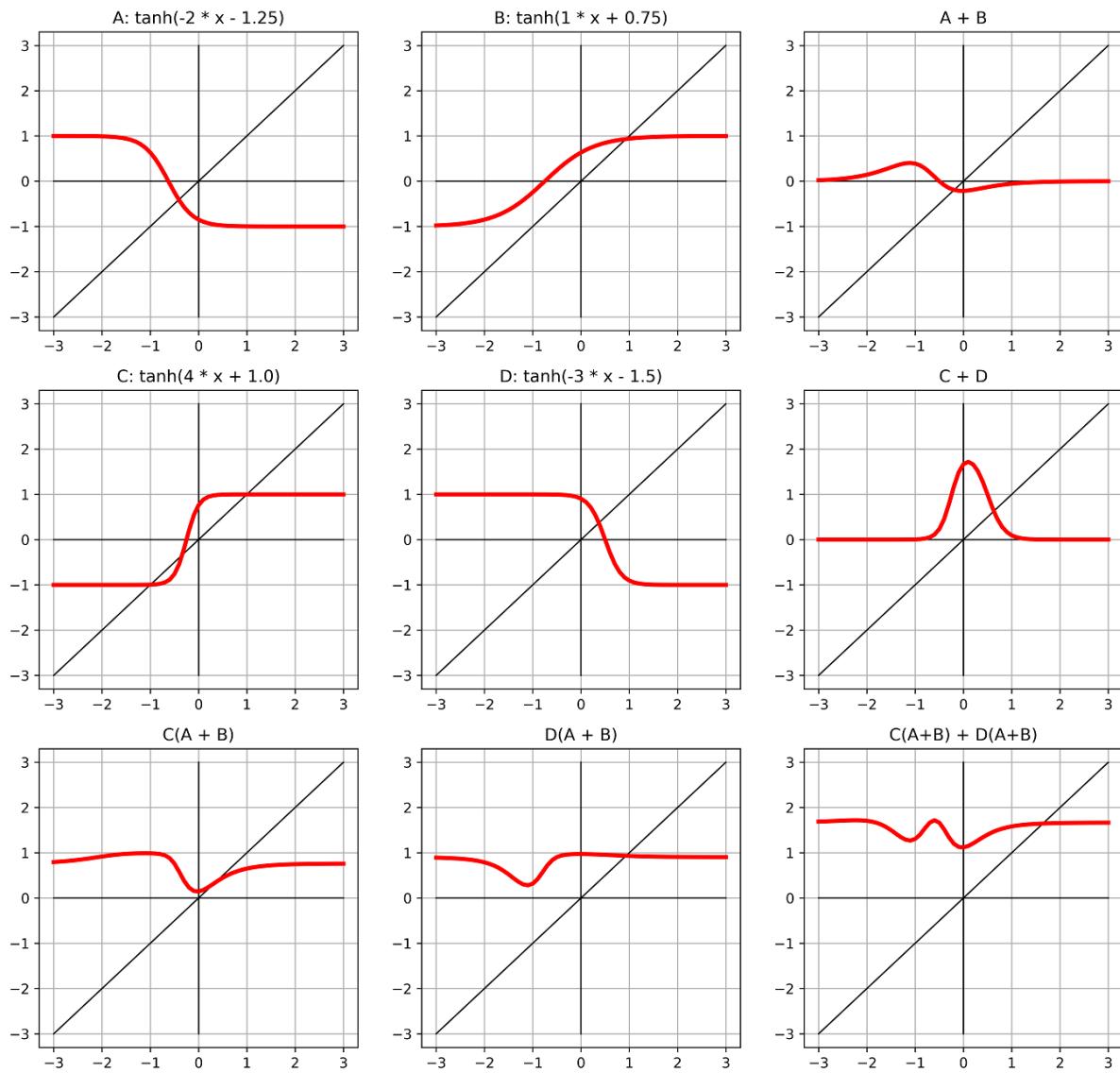


Figure 6.6 Composing multiple linear units and tanh activation functions to produce nonlinear outputs.

The four upper-left graphs show four neurons, A, B, C, and D, each with their own (arbitrarily chosen) weight and bias. Each neuron uses the Tanh activation function, with a min of -1 and a max of 1. The varied weights and biases will move the centerpoint and change how drastically the transition from min to max goes, but they clearly are all of the same general shape. The columns to the right of those show the both pairs of neurons added together (A+B and then C+D). Here, we start to see some interesting properties, which 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 our single-neuron max of 1.

In the third row we start to compose our neurons, as they would be in a two-layer network. Both C(A+B) and D(A+B) have the same positive-and-negative-bumps that A+B showed, 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 very subtle second positive peak as well to the left of the main area of interest. All this with only four neurons in two layers!

Again, these neurons' parameters were chosen only to have a visually interesting result. Training consists in finding acceptable values for these weights and biases so that the resulting network correctly carries out a task, such as predicting likely temperatures given geographic coordinates and time of the year. By *carrying out a task successfully* we mean obtaining a correct output on unseen data produced by the same data-generating process used for training data. A successfully trained network, through the value of its weights and biases, will capture the inherent structure of the data in the form of meaningful numerical representations that work correctly for previously unseen data.

Let's take another step in our realization of the mechanics of learning: deep neural networks give us the ability to approximate highly non-linear phenomena without having an explicit model for them. Instead, starting from a generic, untrained model, we specialize it on a task by providing it a set of inputs and outputs and a loss function to back-propagate from. Specializing a generic model to a task using examples is what we refer to as *learning*, because the model wasn't built with that specific task in mind - no rules describing how that task worked were encoded in the model.

For our thermometer experience we 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 (i.e. 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 one can estimate the unknown parameters from measurements and get an accurate model of the world. Deep neural networks, on the other end, are families of functions that have the ability to 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, we're renouncing an explanation in exchange for the possibility to tackle increasingly complicated problems. In another way, we sometimes lack the ability, information or computational resources to build an explicit model of what we're presented with, so data-driven methods are our only way forward.

6.2 The PyTorch `nn` module

All this talking about neural networks is getting us really curious about building one from scratch with PyTorch. Our first step will be to replace our linear model with a neural network unit. This will be a somewhat useless step backwards from a correctness perspective, since we've already verified that our calibration only required a linear function, but it will still be instrumental for starting on a sufficiently simple problem and scaling up later.

PyTorch has a whole submodule dedicated to neural networks, called `torch.nn`. It contains the building blocks needed to create all sorts of neural network architectures. Those building blocks are called *modules* in PyTorch parlance (these building blocks are often referred to as *layers* in other frameworks). A PyTorch module is a Python class deriving from the `nn.Module` base class. A Module can have one or more `Parameter` instances as attributes, which are tensors whose values are optimized during the training process (think `w` and `b` in our linear model). A Module can also have one or more submodules (subclasses of `nn.Module`) as attributes, and it will be able to track their Parameters as well.

NOTE

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

Unsurprisingly, we can find a subclass of `nn.Module` called `nn.Linear`, which applies an affine transformation to its input (via the parameter attributes `weight` and `bias`) and it is equivalent to what we implemented earlier in our thermometer experiments. We'll now start precisely where we left off and convert our previous code to a form that uses `nn`.

All PyTorch-provided subclasses of `nn.Module` have their `__call__` method defined. This allows one to instantiate an `nn.Linear` and call it as if it was a function, like so:

Listing 6.1 code/p1ch6/1_neural_networks.ipynb

```
# In[5]:  
import torch.nn as nn  
  
linear_model = nn.Linear(1, 1) ①  
linear_model(t_un_val)  
  
# Out[5]:  
tensor([-0.9852],  
      [-2.6876]), grad_fn=<AddmmBackward>
```

① We'll 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 is what executes the forward computation, while `__call__` does other rather important chores before and after calling `forward`. So, it is technically possible to call `forward` directly and it will produce the same output as `__call__`, but it should not be done from user code:

```
>>> y = model(x) ①  
>>> y = model.forward(x) ②
```

- ① Correct!
- ② Silent error. Don't do it!

Here's the implementation of `Module.__call__` (we left out the bits related to the JIT and made some simplifications for clarity):

Listing 6.2 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 we can see, there are a lot of hooks that won't get called properly if we just use `.forward(...)` directly.

Back to our linear model. The constructor to `nn.Linear` accepts three arguments: the number of input features, the number of output features and whether the linear model includes a bias or not (defaulting to `True`, here).

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

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

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

- ① The arguments are input size, output size, and bias defaulting to `True`.

The number of features in our case just refers to the size of the input and the output tensor for the module, so 1 and 1. If we used both temperature and barometric pressure in input, for instance, we would have two features in input and one feature in output. As we will see, for more complex models with several intermediate modules, the number of features will be associated with the capacity of the model.

We have an instance of `nn.Linear` with one input and one output feature. That only requires one weight, and one bias:

```
# In[6]:
```

```

linear_model.weight

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

# In[7]:
linear_model.bias

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

```

We 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 us get away with it, we didn't actually provide an input with the right dimensionality. We have a model that takes one input and produces one output, but PyTorch `nn.Module` and its subclasses are designed to do so on multiple samples at the same time. To accommodate multiple samples, modules expect the zeroth dimension of the input to be the number of samples in the *batch*. We have already encountered this concept in Chapter 4, when we learned how to arrange real-world data into tensors.

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

```

# 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=<AddmmBackward0>)

```

Let's dig into what's going on here, with Figure-6.7 showing a similar situation with batched image data. Our 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.

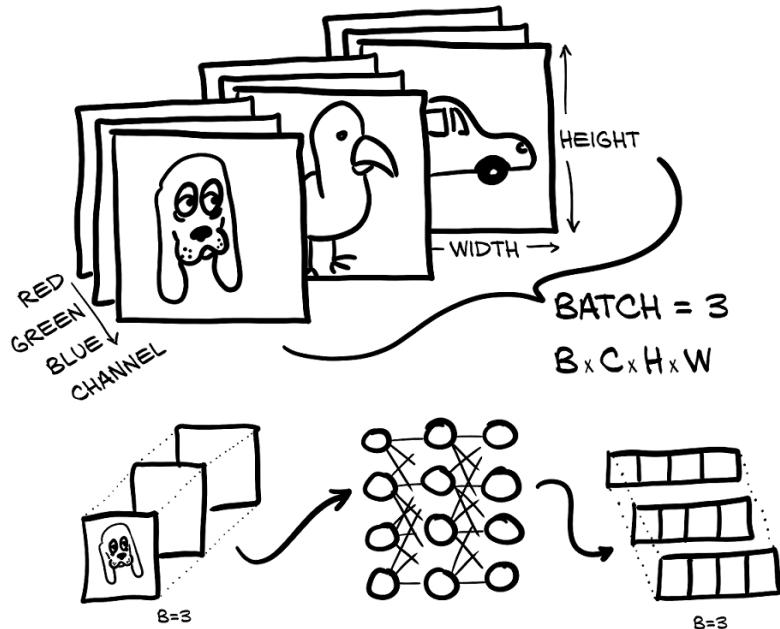


Figure 6.7 Three RGB images batched together, and fed into a neural network. The output is a batch of three vectors of size four.

As we can see, the output is a tensor of size $B \times N_{out}$, where N_{out} is the number of output features, four in this case.

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.

Back to our thermometer data, our t_u and t_c were two 1-dimensional tensors of size B . Thanks to broadcasting, we could write our linear model as $w * x + b$, where w and b were two scalar parameters. This worked because we just had one input feature: if we had two, we would need to add an extra dimension to turn that 1-dimensional tensor into a matrix with samples in the rows and features in the columns.

That's exactly what we need to do to switch to using `nn.Linear`. We reshape our B inputs to $B \times N_{in}$, where N_{in} is 1. That is easily done with `unsqueeze`:

```
# In[2]:
t_c = [0.5, 14.0, 15.0, 28.0, 11.0, 8.0, 3.0, -4.0, 6.0, 13.0, 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) ①
t_u = torch.tensor(t_u).unsqueeze(1) ①

t_u.shape
```

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

- ➊ Here we add the extra dimension at axis 1.

We're done, let's update our training code. First we replace our hand-made model with `nn.Linear(1, 1)`, then we need to pass the linear model parameters to the optimizer.

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

- ➊ This is just a redefinition from above.
- ➋ We replaced `[params]` with this method call.

Earlier it was our responsibility to create parameters and pass them as the first argument to `optim.SGD`. Now we can just ask any `nn.Module` for a list of parameters owned by it or any of its submodules 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 will recurse into submodules defined in the module's `init` constructor and return a flat list of all parameters encountered, so that we can conveniently pass it to the optimizer constructor as we did above.

We can already figure out what happens in the training loop. The optimizer is provided with a list of tensors that were defined with `requires_grad = True` - all `Parameter`s are defined this way by definition, since they need to be optimized by gradient descent. When `training_loss.backward()` is called, `grad` is accumulated on the leaf nodes of the graph, which are precisely the parameters that were passed to the optimizer.

At this point, the SGD optimizer has everything it needs. When `optimizer.step()` is called, it will iterate through each `Parameter` and change it by an amount proportional to what is stored in its `grad` attribute. Pretty clean design.

Let's take a look at the training loop now:

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

        t_p_val = model(t_u_val) ①
        loss_val = loss_fn(t_p_val, t_c_val)

        optimizer.zero_grad() ②
        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)))
```

- ① The model is now passed in, instead of the individual params.
- ② The loss function is also passed in. We'll make use of that in a moment.

It hasn't changed practically at all, except that now we don't pass `params` explicitly to `model` since the model itself holds its `Parameters` internally.

There's one last bit that we can leverage from `torch.nn`: the loss. Indeed, `nn` comes with several common loss functions, among which `nn.MSELoss` (`MSE` stands for Mean Square Error), which is exactly what we defined earlier as our `loss_fn`. Loss functions in `nn` are still subclasses of `nn.Module`, so we will create an instance and call it as a function. In our case, we get rid of the hand-written `loss_fn` and replace it:

```
# In[15]:
linear_model = nn.Linear(1, 1)
optimizer = optim.SGD(linear_model.parameters(), lr=1e-2)

training_loop(
    n_epochs = 3000,
    optimizer = optimizer,
    model = linear_model,
    loss_fn = nn.MSELoss(), ①
    t_u_train = t_u_train,
    t_u_val = t_u_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)
```

- ① We are no longer using our hand-written 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.

6.2.1 Finally a Neural Network

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

There's one last step left to take: replacing our linear model with a neural network as our approximating function. We have said it earlier: using a neural network will not result in a higher-quality model, since the process underlying our calibration problem was fundamentally linear. However, it's good to make the leap from linear to neural network in a controlled environment, so that we won't feel lost later on.

We are going to keep everything else fixed, including the loss function, and only redefine the model. Let's build the simplest possible neural network: a linear module, followed by an activation function, feeding into another linear module. The first linear + activation layer is commonly referred to as a *hidden* layer for historical reasons, since its outputs are not observed directly but fed into the output layer. While the input and 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 is usually larger than one. Recalling our earlier explanation on the role of activations, this can lead different units to respond to different ranges of the input, which increases the capacity of our model. The last linear layer will take the output of activations and combine them linearly to produce the output value.

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

```
# In[16]:
seq_model = nn.Sequential(
    nn.Linear(1, 13), ①
    nn.Tanh(),
    nn.Linear(13, 1)) ②
seq_model

# Out[16]:
Sequential(
    (0): Linear(in_features=1, out_features=13, bias=True)
    (1): Tanh()
    (2): Linear(in_features=13, out_features=1, bias=True)
)
```

- ① 13 was chosen arbitrarily. We wanted to pick a number that was a different size from the other various tensor shapes we have floating around.
- ② This 13 must match the first size, however.

The end result is a model that takes the inputs expected by the first module specified as an argument of `nn.Sequential`, passes intermediate outputs to subsequent modules and produces the output returned by the last module. The model fans out from 1 input feature to 13 hidden features, passes them through a `tanh` activation and linearly combines the resulting 13 numbers into 1 output feature.

Calling `model.parameters()` will collect 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 calling `model.backward()` all parameters will be populated with their `grad` and the optimizer will then update their value accordingly during the `optimizer.step()` call. Not that different from our previous linear model, eh? After all, they're both differentiable models that can be trained using gradient descent.

A few notes on parameters of `nn.Modules`. When inspecting parameters of a model made up of several submodules, it is handy to be able to identify parameters by their name. There's a method for that, it's 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 just the ordinal with which the module has appeared in the arguments. Interestingly, `Sequential` also accepts an `OrderedDict`⁷⁴ in which we can name each module passed to `Sequential`:

```
# In[19]:  
from collections import OrderedDict  
  
seq_model = nn.Sequential(OrderedDict([  
    ('hidden_linear', nn.Linear(1, 8)),  
    ('hidden_activation', nn.Tanh()),  
    ('output_linear', nn.Linear(8, 1))  
]))  
  
seq_model  
  
# Out[19]:  
Sequential(  
    (hidden_linear): Linear(in_features=1, out_features=8, bias=True)
```

```

        (hidden_activation): Tanh()
        (output_linear): Linear(in_features=8, out_features=1, bias=True)
    )

```

This allows us to get more explanatory names for submodules:

```

# In[20]:
for name, param in seq_model.named_parameters():
    print(name, param.shape)

# Out[20]:
hidden_linear.weight torch.Size([8, 1])
hidden_linear.bias torch.Size([8])
output_linear.weight torch.Size([1, 8])
output_linear.bias torch.Size([1])

```

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

```

# In[21]:
seq_model.output_linear.bias

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

```

This is useful for inspecting parameters or their gradients, for instance to monitor gradients during training, as we were doing at the beginning of this Chapter. Say we want to print out the gradients of `weight` of the linear portion of the hidden layer. We can run the training loop for the new neural network model, then look at the resulting gradients after the last epoch:

```

# In[22]:
optimizer = optim.SGD(seq_model.parameters(), lr=1e-3) ❶

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]))
```

- ① Note that we've dropped the learning rate a bit to help with stability.

We can also evaluate the model on the whole data and see how different that 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')
```

Which produces 6.8:

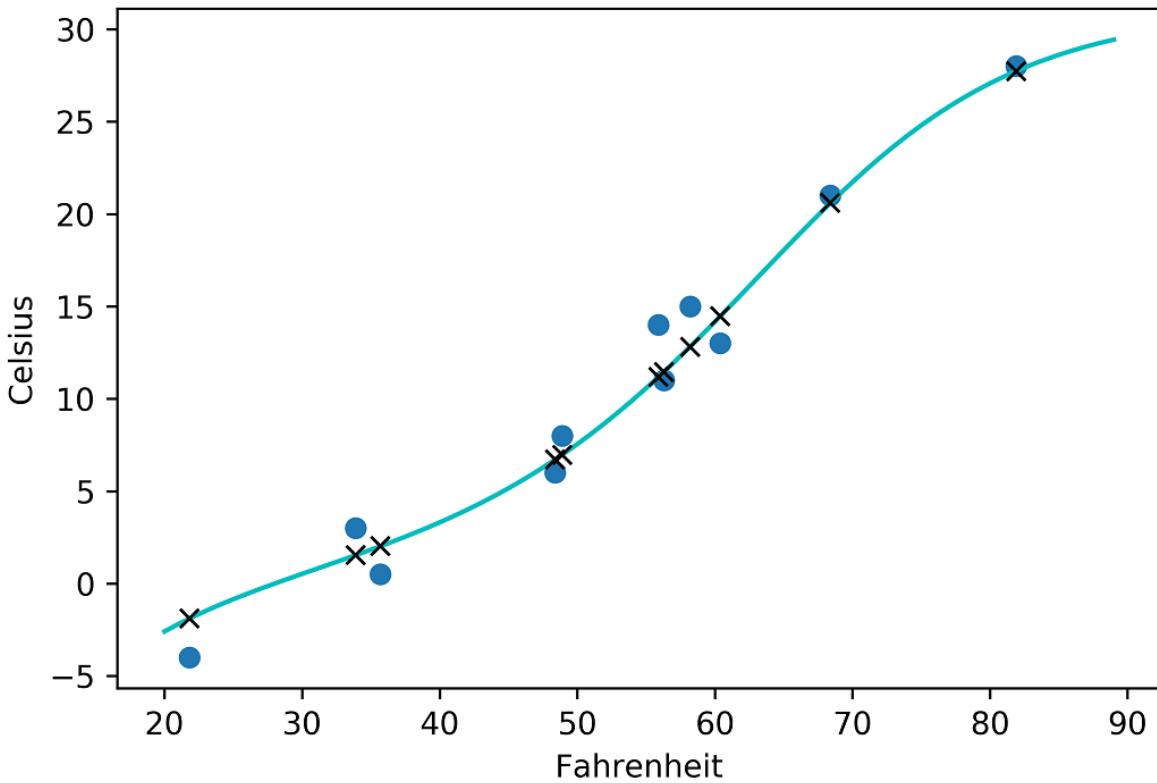


Figure 6.8 The plot of our neural network model, with input data (circles) and model output (Xs). The continuous line shows behavior between samples.

We can appreciate that the neural network has a tendency to overfit as we discussed in chapter 5, since it tries to chase the measurements, including the noisy ones. It doesn't do a bad job though, overall.

6.3 Subclassing nn.Module

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

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

Let's see three different ways to implement the same network structure, using increasingly more complex PyTorch functionality to do so. As we do so, we will vary the number of neurons in the hidden layer to make it easier to differentiate between the approaches. The first will be a simple instance of `nn.Sequential` like we've already seen:

Listing 6.3 code/p1ch6/3_nn_module_subclassing.ipynb

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

- ① The choice of 11 is somewhat arbitrary, but the sizes of the two layers must match.

While this works, we don't have any semantic information about what the various layers are intended to be. We can rectify that 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. We 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. We can take full control of the processing of input data by subclassing `nn.Module` ourselves:

```

# In[4]:
class SubclassModel(nn.Module):
    def __init__(self):
        super().__init__() ①

        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)
)

```

- ① This calls `nn.Module`'s `__init__` which sets up the housekeeping. If you forget it, you will get an error message reminding you that you should call it before using `nn.Module`'s capabilities we get into below.

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

Typically we will want to use the constructor of the module to define the submodules that we call in the `forward` function, so they can hold their parameters throughout the lifetime of our module. For instance, we instantiate two instances of `nn.Linear` in the constructor and use them in `forward`. Interestingly, assigning an instance of `nn.Module` to an attribute in a `nn.Module`,

just like we did in the constructor here, automatically registers the module as a submodule. This allows modules to have access to the parameters of its submodules without further action by the user.

You can call arbitrary methods of an `nn.Module` subclass — for example for a model where training is substantially different than it's use, say, for prediction, it may make sense to have a `predict` method. You should be aware that calling them will be similar to calling `forward` instead of the module itself — they will be ignorant of hooks and the JIT does not see the module structure when using them because we are missing the equivalent of the `__call__` bits shown in 6.2.

Going back to our non-random `SubclassModel`, we can see that the printed output for that class is very similar to the output for the sequential model with named parameters. This makes sense, since we used the same names and intended to implement the same architecture. If we look at the parameters of all three models, we will 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 happened 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, 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 will know how to change parameters so to minimize the loss. We know that story from chapter 5.

NOTE

Child modules contained inside Python `list` or `dict` instances will not be automatically registered! Subclasses can either register those modules manually with the `add_module(name, module)` method of `nn.Module`⁷⁵, 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 we can access their parameters, it appears a bit of a waste that we are also registering submodules that have no parameters, like `nn.Tanh`. Wouldn't just be easier to call them directly in the `forward` function?⁷⁶

6.3.1 The Functional API

It sure would! And that's why PyTorch has *functional* counterparts of every `nn` module. By "functional" here we mean "having no internal state", or, in other words, "whose output value is solely and fully determined by the value input arguments". Indeed, `torch.nn.functional` provides the many of the same modules we find in `nn`, but with all eventual parameters moved as an argument to the function call. For instance, the functional counterpart of `nn.Linear` 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.

Back to our model, it makes sense to keep using `nn` modules for `nn.Linear`, so that `SubclassModel` will be able to manage all of its `Parameter` instances during training. However, we can safely switch to the functional counterparts of `Tanh`, since it has no parameters.

```
# In[6]:
class SubclassFunctionalModel(nn.Module):
    def __init__(self):
        super().__init__()

        self.hidden_linear = nn.Linear(1, 14)           ①
        self.output_linear = nn.Linear(14, 1)

    def forward(self, input):
        hidden_t = self.hidden_linear(input)
        activated_t = torch.tanh(hidden_t)  ②
        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)
)
```

- ① The `self.hidden_activation = ...` line is missing here.
- ② It 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 will start to add up!). Note that it would still make sense to instantiate modules that require arguments for their initialization in the constructor. For example, `HardTanh` takes optional `min_val` and `max_val` arguments and defining those in the `__init__` saves stating them in the body of `forward`.

TIP

While 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 will remain in `torch.nn.functional`.

In this way, the functional way also sheds light on what the `nn.Module` API is all about: A `Module` is a container for state in forms of `Parameter`s and submodules combined with the instructions to do a forward.

Whether to use the functional or the modular API is a bit of style and taste decision. When we have a part of a network that is so simple that we want to use `nn.Sequential`, we would be in the modular realm. When we are writing our own forwards, it may be more natural to use the functional interface for things that do not need state in the form of parameters.

In [chapter 15](#) we will see that with quantization, stateless bits like activations suddenly become stateful because information on the quantization needs to be captured. This means that if we aim to quantize our model, it might be worthwhile to stick with the modular API if we go for non-JITed quantization. There is one style matter that will help you avoid surprises with (originally unforeseen) uses: if you need several applications of stateless modules (like `nn.HardTanh` or `nn.ReLU`), it is likely a good idea to have a separate instance for each. Re-using the same module appears to be clever and will give correct results with our standard Python usage here, but tools analysing your model might trip over it.

6.4 Conclusion

We have covered a lot in these last two chapters, although we only dealt with a very simple problem. We dissected building differentiable models and training them using gradient descent, using raw autograd first and then relying on `nn`. By now we should have confidence in our understanding of what's going on behind the scenes.

Hopefully this taste of PyTorch has given you an appetite for more!

6.5 Exercises

- Experiment with the number of hidden neurons in our 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 4, 4.3 and create a new model with the appropriate number of input parameters.
 - How long does it take to train compared to the temperature data we have 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?

6.6 Summary

- Neural networks can be automatically adapted to specialize themselves on the problem at hand.
- Neural networks allow easy access to the analytical derivatives of the loss with respect to any parameter in the model, which makes evolving the parameters very efficient. Thanks to its automated differentiation engine, PyTorch provides such derivatives effortlessly.
- Activation functions around linear transformations make neural networks capable of approximating highly non-linear functions, at the same time keeping them simple enough to optimize.
- The `nn` module, together with the tensor standard library, provide all the building blocks for creating neural networks.
- To recognize overfitting, it's essential to maintain the training set of data points separate from the validation set. There's no one recipe to combat overfitting, but getting more data, or more variability in the data, and resorting to simpler models are good starts.
- Anyone doing data science should be plotting data all the time.

Telling Birds from Airplanes: Learning from Images

This chapter covers:

- working through an image recognition problem, step by step
- building a feed forward neural network to classify images
- loading data using Datasets and DataLoaders
- why and how to use a classification loss
- describing how convolution works
- building a convolutional neural network to classify images

The last chapter gave us the opportunity to dive into the inner mechanics of learning through gradient descent, and the facilities that PyTorch offers to build models and optimize them. We have done so on a simple regression model of one input and one output, which allowed us to have everything in plain sight, but admittedly was only borderline exciting.

In this chapter we'll keep moving ahead on building our neural network foundations. This time, we'll turn our attention to images. Image recognition is arguably the task that made the world realize the potential of deep learning.

We will now approach a simple image recognition problem step by step, building from a simple neural network like we defined in the last chapter. This time, instead of a tiny dataset of numbers, we'll use a more extensive dataset of tiny images. Let's download the dataset first and get to work preparing it for use.

7.1 A dataset of tiny images

There is nothing like an intuitive understanding of a subject, and there is nothing to achieve that like working on simple data. There's a dataset for image recognition that we first encountered in Chapter 2, the hand-written digit recognition dataset known as MNIST. There's another dataset that is similarly simple and a bit more fun. It's called CIFAR-10, and, together with its sibling CIFAR-100, it has been a computer vision classic for a decade.

CIFAR-10 consists of 60000 tiny 32x32 color (RGB) images, labeled with an integer corresponding to 10 classes, namely airplane (0), automobile (1), bird (2), cat (3), deer (4), dog (5), frog (6), horse (7), ship (8), truck (9).



Figure 7.1 Image samples from all CIFAR-10 classes.

The images were collected and labeled by Krizhevsky, Nair and Hinton from CIFAR (the Canadian Institute For Advanced Research), and were drawn from a larger collection of unlabeled 32x32 color images, the "80 million tiny images dataset", from CSAIL (the Computer Science and Artificial Intelligence Laboratory) at the Massachusetts Institute of Technology.

Nowadays, CIFAR-10 is considered too simple for developing or validating new research, but it serves our learning purposes just fine. Just as we did with the MNIST dataset in Chapter 2, we will use the `torchvision` module to automatically download the dataset and load it as a collection of PyTorch tensors.

7.1.1 Downloading CIFAR10

As we anticipated, let's import `torchvision` and use the `datasets` module to download the CIFAR-10 data:

```
# In[2]:  
from torchvision import datasets  
data_path = '../data-unversioned/plch6/'  
cifar10 = datasets.CIFAR10(data_path, train=True, download=True)  
cifar10_val = datasets.CIFAR10(data_path, train=False, download=True)
```

The first argument we just provided to the `CIFAR10` function is the location where the data will

be downloaded, the second whether we're interested in the training or the validation set and the third whether we allow PyTorch to download the data if it is not found in the location specified as the first argument.

Just like CIFAR10, the datasets submodule gives us pre-canned access to the most popular computer vision datasets, such as MNIST, FashionMNIST, CIFAR-100, SVHN, Coco, Omniglot. In all cases, they are returned as a subclass of `torch.utils.data.Dataset`. We can see the method resolution order of our `cifar10` instance includes it as a base class:

```
# In[4]:  
type(cifar10).__mro__  
  
# Out[4]:  
(torchvision.datasets.cifar.CIFAR10, torch.utils.data.dataset.Dataset, object)
```

7.1.2 The Dataset class

It's a good time to discover what being a subclass of `torch.utils.data.Dataset` means in practice. A PyTorch Dataset is an object that is required to implement two methods: `{uu}len{uu}` and `{uu}getitem{uu}`. The former should return the number of items in the dataset, while the latter should return the item, consisting of a sample and its corresponding label (an integer index).

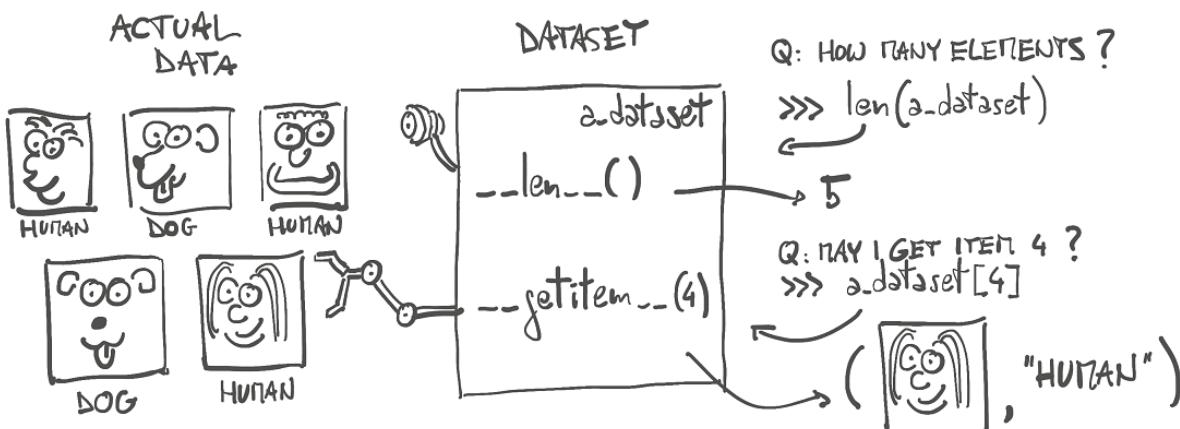


Figure 7.2 Concept of a PyTorch Dataset: it doesn't necessarily hold the data, but it provides a uniform access it through `{uu}len{uu}` and `{uu}getitem{uu}`.

In practice, when a Python object is equipped with the `{uu}len{uu}` method, we can pass it as an argument to the `len` Python built-in function, like

```
# In[5]:  
len(cifar10)  
  
# Out[5]:  
50000
```

Similarly, since the loader is equipped with the `{uu}getitem{uu}` method, we can use the

standard subscript for indexing tuples and lists to access individual items. Here, we get a `PIL` image, with our desired output, an integer with value 1, corresponding to "automobile":

```
# In[6]:
img, label = cifar10[99]
img, label, class_names[label]

# Out[6]:
(<PIL.Image.Image image mode=RGB size=32x32 at 0x18AA30C40F0>, 1, 'automobile')
```

So, the sample in the `data.CIFAR10` dataset is an instance of a RGB PIL image. We can plot it right away:

```
# In[7]:
plt.imshow(img)
plt.show()
```

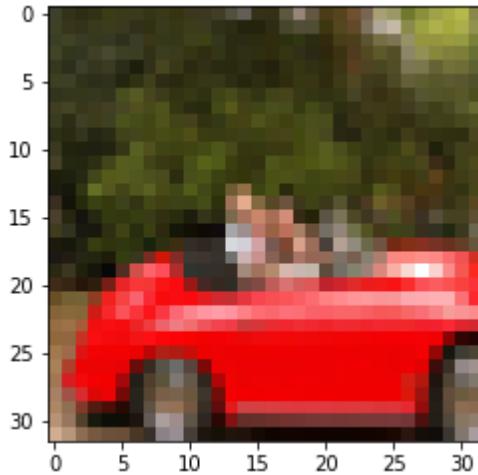


Figure 7.3 The 99th image from the CIFAR-10 dataset, an automobile.

It's a red car! ⁷⁷

7.1.3 Dataset transforms

That's all very nice, but we'll very likely need a way to convert the PIL image to a PyTorch tensor before we can do anything with it. That's where `torchvision.transforms` comes in. This module defines a set of composable function-like objects that can be passed as an argument to a `torchvision` dataset such as `datasets.CIFAR10(...)`, and that perform transformations on the data after it is loaded but before it is returned by `{uu}getitem{uu}`. We can see the list available with:

```
# In[8]:
from torchvision import transforms
dir(transforms)

# Out[8]:
['CenterCrop',
 'ColorJitter',
 ...]
```

```
'Normalize',
'Pad',
'RandomAffine',
...
'RandomResizedCrop',
'RandomRotation',
'RandomSizedCrop',
...
'TenCrop',
'ToPILImage',
'ToTensor',
...
]
```

Among those transforms we can spot `ToTensor`, that turns NumPy arrays and PIL images to tensors. It also takes care to lay out the dimensions of the output tensor as `CxHxW` (channel, height, width; just as we covered in Chapter 4).

Let's try out the `ToTensor` transform: once instantiated, it can be called like a function with the PIL image as the argument, returning a tensor in output:

```
# In[9]:
from torchvision import transforms

to_tensor = transforms.ToTensor()
img_t = to_tensor(img)
img_t.shape

# Out[9]:
torch.Size([3, 32, 32])
```

The image has been turned into a `3x32x32` tensor, therefore a three-channel (RGB), `32x32` image. Note that nothing has happened to label, it is still an integer.

As we anticipated, we can pass the transform directly as an argument to `dataset.CIFAR10`:

```
# In[10]:
tensor_cifar10 = datasets.CIFAR10(data_path, train=True, download=False,
                                    transform=transforms.ToTensor())
```

At this point, accessing an element of the Dataset will return a tensor, rather than a PIL image:

```
# In[11]:
img_t, _ = tensor_cifar10[99]
type(img_t)

# Out[11]:
torch.Tensor
```

As expected, the shape has channel as the first dimension, while the scalar type is `float32`:

```
# In[12]:
img_t.shape, img_t.dtype

# Out[12]:
(torch.Size([3, 32, 32]), torch.float32)
```

While the values in the original PIL image ranged from 0 to 255 (8-bit per channel), the `ToTensor` transform turned the data into 32-bit floating point per channel, scaling values down from 0.0 to 1.0. Let's verify that:

```
# In[13]:  
img_t.min(), img_t.max()  
  
# Out[13]:  
(tensor(0.), tensor(1.))
```

And let's verify that we're getting the same image out:

```
# In[14]:  
plt.imshow(img_t.permute(1, 2, 0))  
plt.show()  
  
# Out[14]:  
<Figure size 432x288 with 1 Axes>
```

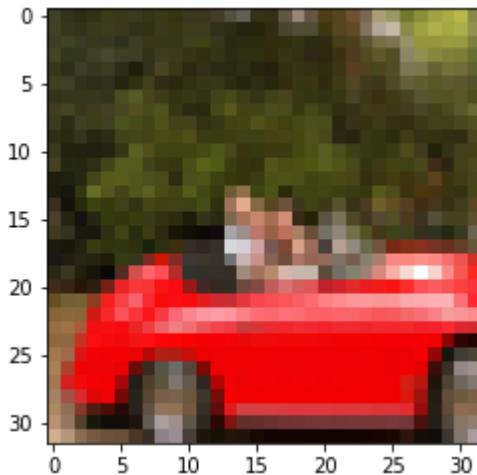


Figure 7.4 We've seen this one already.

It checks. Note how we had to use `permute` to change the order of the axes from `CxHxW` to `HxWxC` to match what Matplotlib expects.

7.1.4 Normalizing data

Transforms are really handy, because we can chain them using `transforms.Compose`, and they can handle normalization and data augmentation transparently, directly in the data loader.

For instance, it's good practice to normalize the dataset so that each channel has zero mean and unitary standard deviation. We've already mentioned this in Chapter 4, but now, after going through Chapter 5, we can also have an intuition for why: by choosing activation functions that are linear around zero plus or minus one (or two), keeping the data in the same range means that

it's more likely that neurons have non-zero gradients, hence they will learn sooner. Also, normalizing each channel so that it has the same distribution will ensure channel information can be mixed and updated through gradient descent using the same learning rate.

In order to make it so that each channel has zero mean and unitary standard deviation we can compute the mean value and the standard deviation of each channel across the dataset, and apply the following transform: $v_n[c] = (v[c] - \text{mean}[c]) / \text{stdev}[c]$. This is what `transforms.Normalize` does. The values of `mean` and `stdev` must be computed offline (they are not computed by the transform). Let's compute those for the CIFAR-10 training set.

Since the CIFAR-10 dataset is small, we'll be able to manipulate it all in memory. Let's stack all tensors returned by the dataset along an extra dimension:

```
# In[15]:
imgs = torch.stack([img_t for img_t, _ in tensor_cifar10], dim=3)
imgs.shape

# Out[15]:
torch.Size([3, 32, 32, 50000])
```

Now we can easily compute the mean per channel:

```
# In[16]:
imgs.view(3, -1).mean(dim=1)

# Out[16]:
tensor([0.4915, 0.4823, 0.4468])
```

and similarly for the standard deviation

```
# In[17]:
imgs.view(3, -1).std(dim=1)

# Out[17]:
tensor([0.2470, 0.2435, 0.2616])
```

With these numbers in our hands, we can initialize the `Normalize` transform:

```
# In[18]:
transforms.Normalize((0.4915, 0.4823, 0.4468), (0.2470, 0.2435, 0.2616))

# Out[18]:
Normalize(mean=(0.4915, 0.4823, 0.4468), std=(0.247, 0.2435, 0.2616))
```

and concatenate it after the `ToTensor` transform

```
# In[19]:
transformed_cifar10 = datasets.CIFAR10(data_path, train=True, download=False,
                                         transform=transforms.Compose([
                                             transforms.ToTensor(),
                                             transforms.Normalize((0.4915, 0.4823, 0.4468),
                                                                 (0.2470, 0.2435, 0.2616))
                                         ]))
```

Note that, at this point, plotting an image drawn from the dataset won't provide us with a faithful representation of the actual image:

```
# In[21]:
img_t, _ = transformed_cifar10[99]

plt.imshow(img_t.permute(1, 2, 0))
plt.show()

# Out[21]:
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integ

<Figure size 432x288 with 1 Axes>
```

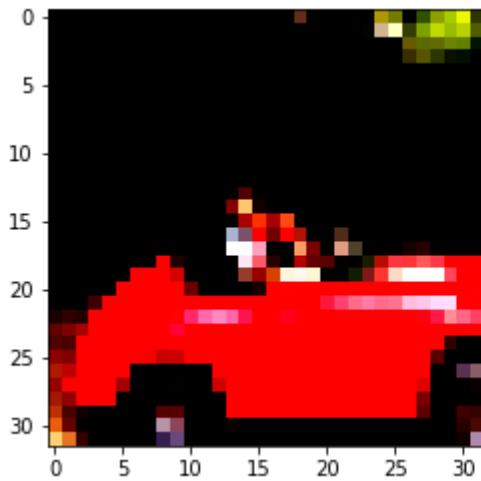


Figure 7.5 Our random CIFAR-10 image after normalization.

This is because normalization has shifted the RGB levels outside the 0.0 to 1.0 range and changed the overall magnitudes of the channels. All of the data is still there, it's just that Matplotlib renders it as black. Let's just keep this in mind for the future.

Still, we have a fancy, tens-of-thousands images dataset loaded! That's quite convenient, because we were going to need something exactly like it.

7.2 Distinguishing birds from airplanes

Jane, our friend at the birdwatching club, has set up a fleet of cameras in the woods, south of the airport. They are supposed to save a shot when something gets into the frame and upload it to the club's real-time bird watching blog. Problem is: there are a lot of planes coming and going from the airport that end up triggering the camera, so our friend needs to spend a lot of time deleting pictures of airplanes from the blog.

No worries, we jump up. We'll take care of that no problem, we just got the perfect dataset for it (what a coincidence, right?). We'll pick out all birds and airplanes from our CIFAR-10 dataset and build a neural network that can tell birds and airplanes apart.

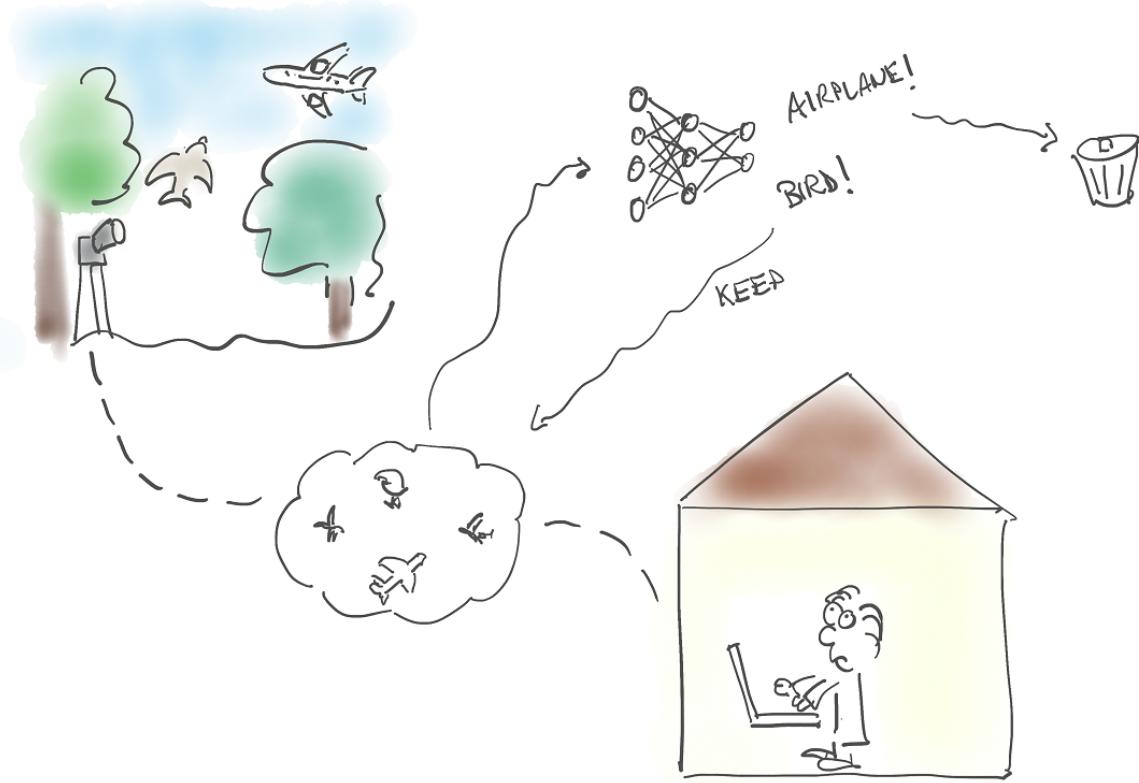


Figure 7.6 The problem at hand: we're going to help our friend tell birds from airplanes for her blog, by training a neural network to do the job.

7.2.1 Building the dataset

First step, get the data in the right shape. We could create our `Dataset` subclass that only includes birds and airplanes. However, the dataset is small and we only need indexing and `len` to work on our `Dataset`. It doesn't actually have to be a subclass of `torch.utils.data.Dataset`! Well, then why not take a little shortcut and just filter the data in `cifar10` and remap the labels so that they are contiguous? Here's how:

```
# In[5]:
label_map = {0: 0, 2: 1}
class_names = ['airplane', 'bird']
cifar2 = [(img, label_map[label]) for img, label in cifar10 if label in [0, 2]]
cifar2_val = [(img, label_map[label]) for img, label in cifar10_val if label in [0, 2]]
```

The `cifar2` object satisfies the basic requirements for a `Dataset`, that is `{uu}len{uu}` and `{uu}getitem{uu}` being defined, so we're going to just use that.

7.2.2 A fully connected classifier

We learned how to build a neural network in Chapter 5. We know that it's a tensor of features in, a tensor of features out. After all, an image is just a set of numbers laid out in some spatial configuration. Ok, we don't know how to handle the spatial configuration part just yet, but in theory if we just take the image pixels and straighten them up into a long 1D vector, we could consider those numbers as input features, right?

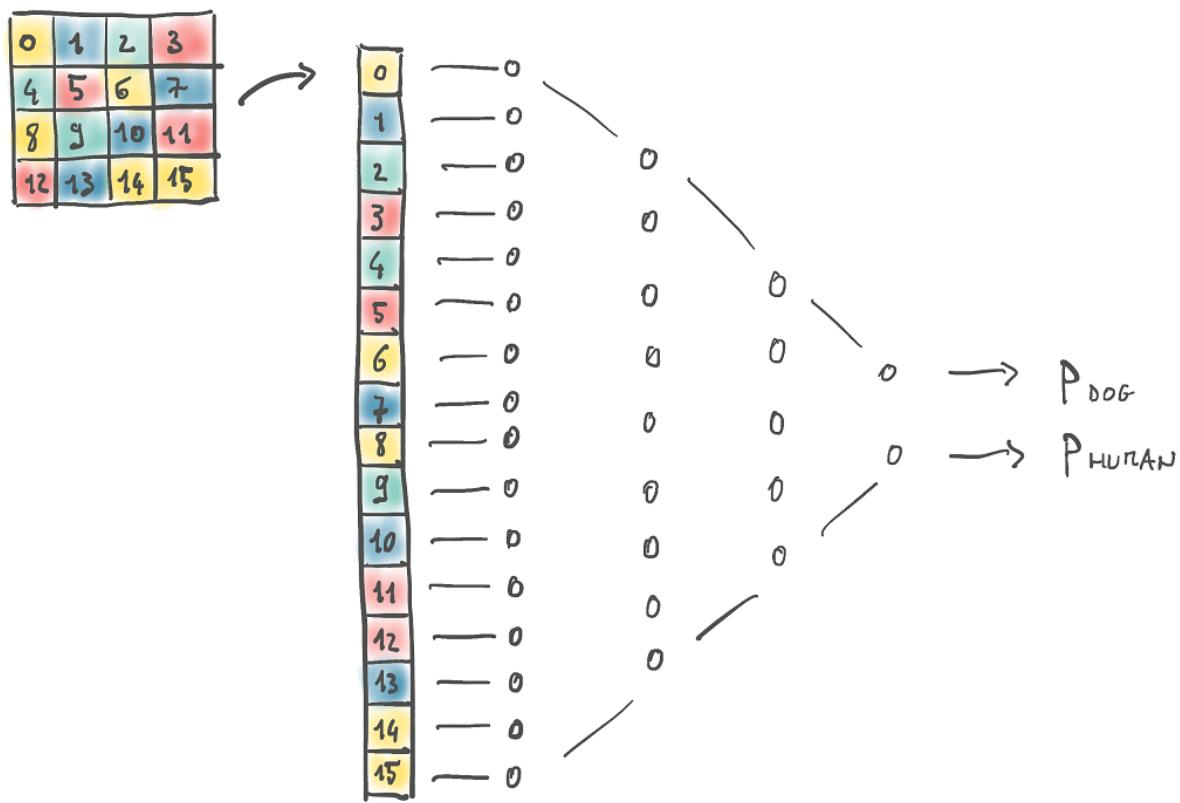


Figure 7.7 Treating our image as a 1D vector of values and training a fully connected classifier on it.

Let's try that. How many features per sample? Well, $32 \times 32 \times 3$, that is 3072 input features per sample. Starting from the model we built in Chapter 5, our new model would be a `nn.Linear` with 3072 input features and some number of hidden features (we pick 512 somewhat arbitrarily here), followed by an activation, then another `nn.Linear` that tapers the network down to an appropriate output number of features (which is 2 for this use case).

```
# In[6]:  
import torch.nn as nn  
  
n_out = 2  
  
model = nn.Sequential(  
    nn.Linear(  
        3072,      ①  
        512,      ②  
    ),  
    nn.Tanh(),
```

```
    nn.Linear(  
        512,  
        n_out,  
    )  
)
```

- ① input features
- ② hidden layer size
- ③ output classes

In Chapter 5, the network was producing the predicted temperature (a number with a quantitative meaning) in output. We could do something similar: make our network output a single scalar value (so `n_out = 1`), cast the labels to a float (`0.0` for airplane and `1.0` for bird) and use those as a target for a `MSELoss` (i.e. the average of squared differences in the batch). Doing so, we would have casted the problem into a regression problem. However, looking more closely, we are now dealing with something a bit different in nature.

We need to recognize that the output is categorical: it's either a bird or an airplane (or something else if we had all ten of the original classes). As we've learned in Chapter 4, when we have to represent a categorical variable, we should switch to a one-hot encoding representation of that variable, i.e. `[1, 0]` for airplane, `[0, 1]` for bird (the order is arbitrary). This will still work if we had 10 classes, like in the full CIFAR-10 dataset; we'd just have a vector of length ten.

In the ideal case, the network would output `torch.tensor([1.0, 0.0])` for an airplane and `torch.tensor([0.0, 1.0])` for a bird. Practically speaking, since our classifier will not be perfect, we can expect the network to output something "in-between". The key realization in this case is that we can interpret our output as probabilities: the first entry is the probability of "airplane" and the second the probability of "bird". In the ideal case, for an airplane the first entry is `1.0`, the second is `0.0`.

Casting the problem in terms of probabilities imposes a few extra constraints on the outputs of our network, namely:

- each element of the output must be in the `[0.0, 1.0]` range (a probability of an outcome cannot be lower than zero and higher than one)
- the elements of the output must add up to 1.0 (we're certain that one of the two outcomes will occur)

It sounds like a tough constraint to enforce in a differentiable way on a vector of numbers. Yet, there's a very smart trick that does exactly that, and it's differentiable: it's called *softmax*.

Softmax is a function that takes a vector of values and produces another vector of the same dimension, where the values satisfy the constraints we listed above for them to represent probabilities. The expression for softmax is

$$0 \leq \frac{e^{x_1}}{e^{x_1} + e^{x_2}} \leq 1$$

EACH ELEMENT
BETWEEN
0 AND 1

$$\frac{e^{x_1}}{e^{x_1} + e^{x_2}} + \frac{e^{x_2}}{e^{x_1} + e^{x_2}} = \frac{e^{x_1} + e^{x_2}}{e^{x_1} + e^{x_2}} = 1$$

SUM OF ELEMENTS
EQUALS 1

$$\text{softmax}(x_1, x_2) = \left(\frac{e^{x_1}}{e^{x_1} + e^{x_2}}, \frac{e^{x_2}}{e^{x_1} + e^{x_2}} \right)$$

$$\text{softmax}(x_1, x_2, x_3) = \left(\frac{e^{x_1}}{e^{x_1} + e^{x_2} + e^{x_3}}, \frac{e^{x_2}}{e^{x_1} + e^{x_2} + e^{x_3}}, \frac{e^{x_3}}{e^{x_1} + e^{x_2} + e^{x_3}} \right)$$

$$\vdots$$

$$\text{softmax}(x_1, \dots, x_n) = \left(\frac{e^{x_1}}{e^{x_1} + \dots + e^{x_n}}, \dots, \frac{e^{x_n}}{e^{x_1} + \dots + e^{x_n}} \right)$$

Figure 7.8 Handwritten softmax.

That is, we take the elements of the vector, compute the element-wise exponential, and divide each element by the sum of exponentials. In code it's something like:

```
# In[7]:
def softmax(x):
    return torch.exp(x) / torch.exp(x).sum()
```

Let's test it on an input vector

```
# In[8]:
x = torch.tensor([1.0, 2.0, 3.0])

softmax(x)

# Out[8]:
tensor([0.0900, 0.2447, 0.6652])
```

As expected, it satisfies the constraints on probability:

```
# In[9]:
softmax(x).sum()

# Out[9]:
tensor(1.)
```

Softmax is a monotone function, in that lower values in the input will correspond to lower values in the output. However, it's not "scale invariant", in that the ratio between values is not preserved. In fact, the ratio between the first and the second elements of the input is 0.5, while

the ratio between the same elements in the output is 0.3678. This is not a real issue, since the learning process will drive the parameters of the model in a way that values have appropriate ratios.

The `nn` module makes softmax available as a module. Since, as usual, input tensors may have an additional batch 0-th dimension, or have dimensions along which they encode probabilities and others in which they don't, `nn.Softmax` requires us to specify the dimension along which the softmax function is applied:

```
# In[10]:
softmax = nn.Softmax(dim=1)

x = torch.tensor([[1.0, 2.0, 3.0],
                 [1.0, 2.0, 3.0]])

softmax(x)

# Out[10]:
tensor([[0.0900, 0.2447, 0.6652],
        [0.0900, 0.2447, 0.6652]])
```

In this case we had two input vectors in two rows (just like when we work with batches), so we initialized `nn.Softmax` to operate along dimension 1.

Excellent, we can now add a softmax at the end of our model, and our network will be equipped at producing probabilities.

```
# In[11]:
model = nn.Sequential(
    nn.Linear(3072, 512),
    nn.Tanh(),
    nn.Linear(512, 2),
    nn.Softmax(dim=1))
```

We can actually try running the model before even training it. Let's do it, just to see what comes out of it. We first build a batch of one image, our bird.

```
# In[12]:
img, _ = cifar2[0]

plt.imshow(img.permute(1, 2, 0))
plt.show()
```

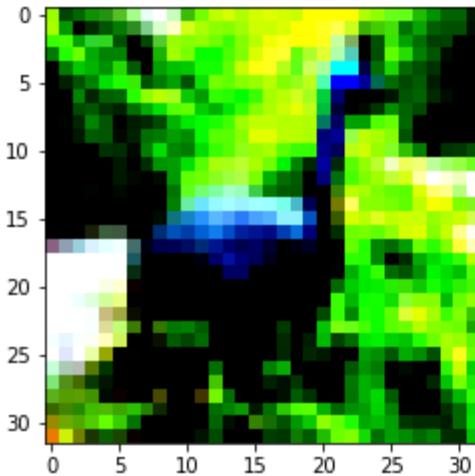


Figure 7.9 A random bird from the CIFAR-10 dataset (after normalization).

Oh, hello there. In order to call the model we need to make the input of the right dimensions. We recall that our model expects 3072 features in input, and that `nn` works with data organized into batches along the 0-th dimension. So we need to turn our 3x32x32 image into a 1D tensor, then add an extra dimension in the 0-th position. We know how to do this since Chapter 3:

```
# In[13]:
img_batch = img.view(-1).unsqueeze(0)
```

Now we're ready to invoke our model:

```
# In[14]:
out = model(img_batch)
out

# Out[14]:
tensor([[0.4784, 0.5216]], grad_fn=<SoftmaxBackward>)
```

So, we got probabilities! Well, we know we shouldn't get too excited: weights and biases of our linear layers have not been trained at all. Their elements are initialized randomly by PyTorch between -1.0 and 1.0. Interestingly, we also see the `grad_fn` for the output, which is the tip of the backward computation graph (it will be used as soon as we need to back propagate).

In addition, while we know which output probability is supposed to be which (recall our `class_names` above), our network has no indication of that. Is the first entry "airplane" and the second "bird", or the other way around? It can't even tell that at this point. It's the loss function that associates a meaning to these two numbers, after back-propagation. If the labels are provided as index 0 for "airplane" and index 1 for "bird", than that's the order the outputs will be induced to take. Therefore, after training we will be able to get the label as an index by computing the `argmax` of the output probabilities, that is, the index at which we get the maximum probability. Conveniently, when supplied a dimension, `torch.max` returns the maximum element along that dimension as well as the index at which that value occurs. In our case, we need to take the max along the probability vector (not across batches), therefore dimension 1:

```
# In[15]:
_, index = torch.max(out, dim=1)

index

# Out[15]:
tensor([1])
```

It says it's a bird. Pure luck. But we've now run our model against an input image and verified that our plumbing works. Time to get training.

7.2.3 A loss for classifying

We just mentioned that the loss is what gives probabilities a meaning. In Chapter 5 we used mean square error as our loss. We could still use MSE and make our output probabilities converge to [0.0, 1.0] and [1.0, 0.0]. However, thinking about it, we're not really interested in reproducing these values exactly. Looking back at the *argmax* operation we used to extract the index of the predicted class, what we're really interested in is that the first probability is higher than the second for airplanes and vice-versa for birds. In other words, we want to penalize misclassifications, rather than painstakingly penalize everything that doesn't look exactly like a 0.0 or 1.0.

What we need to maximize in this case is the probability associated with the correct class, `out[class_index]`, where `out` is the output of softmax and `class_index` is a vector containing 0 for "airplane" and 1 for "bird" for each sample. This quantity, that is, the probability associated to the correct class, is referred to as *likelihood*. In other words, we want a loss function that is very high when the likelihood is low, that is, so low that the alternatives have a higher probability. Conversely, the loss should be low when the likelihood is higher than the alternatives, and we're not really fixated on driving the probability up to one.

There's a loss function that behaves that way, and it's called *negative log likelihood* (NLL). It has the expression `NLL = - sum(log(out_i[c_i]))`, where the sum is taken over `n` samples and `c_i` is the correct class for sample `i`. Let's take a look at Figure TODO, that shows the NLL as a function of likelihood.

The Figure shows that for low likelihoods NLL grows to infinity, while it decreases with a rather shallow rate as likelihoods are greater than 0.5. Remember that NLL takes probabilities in input, so as the likelihood grows the other probabilities will necessarily decrease.

Summing up, our loss for classification can be computed as follows. For each sample in the batch:

- run a forward and obtain the output values from the last (linear) layer
- compute their Softmax and obtain probabilities
- take the probability corresponding to the correct class (the likelihood); note that we know what the correct class is because it's a supervised problem
- compute its logarithm, slap a minus sign in front of it and add it to the loss

So, how do we do this in PyTorch? PyTorch has a `nn.NLLLoss` class. However (gotcha ahead), as opposed to what the name suggests, it doesn't take a logarithm of the likelihood, but it expects a tensor of log probabilities in input. There's a good reason behind it: taking the logarithm of a probability gets tricky when the probability gets close zero. The workaround is to use `nn.LogSoftmax` instead of `nn.Softmax`, which takes care to make the calculation numerically stable.

We can now modify our model to use `nn.LogSoftmax` as the output module:

```
>>> model = nn.Sequential(  
>>>     nn.Linear(3072, 512),  
>>>     nn.Tanh(),  
>>>     nn.Linear(512, 2),  
>>>     nn.LogSoftmax(dim=1))
```

Then we instantiate our negative log likelihood loss:

```
>>> loss = nn.NLLLoss()
```

The loss takes the output of `nn.LogSoftmax` for a batch as the first argument and a tensor of class indices (0's and 1's in our case) as the second argument. We can now test it with our birdie:

```
>>> img, label = cifar2[0]  
>>  
>>> out = model(img.view(-1).unsqueeze(0))  
>>  
>>> loss(out, torch.tensor([label]))  
  
tensor(0.6509, grad_fn=<NllLossBackward>)
```

7.2.4 Training the classifier

Alright! We're ready to bring back the training loop we wrote in Chapter 5 and see how it trains:

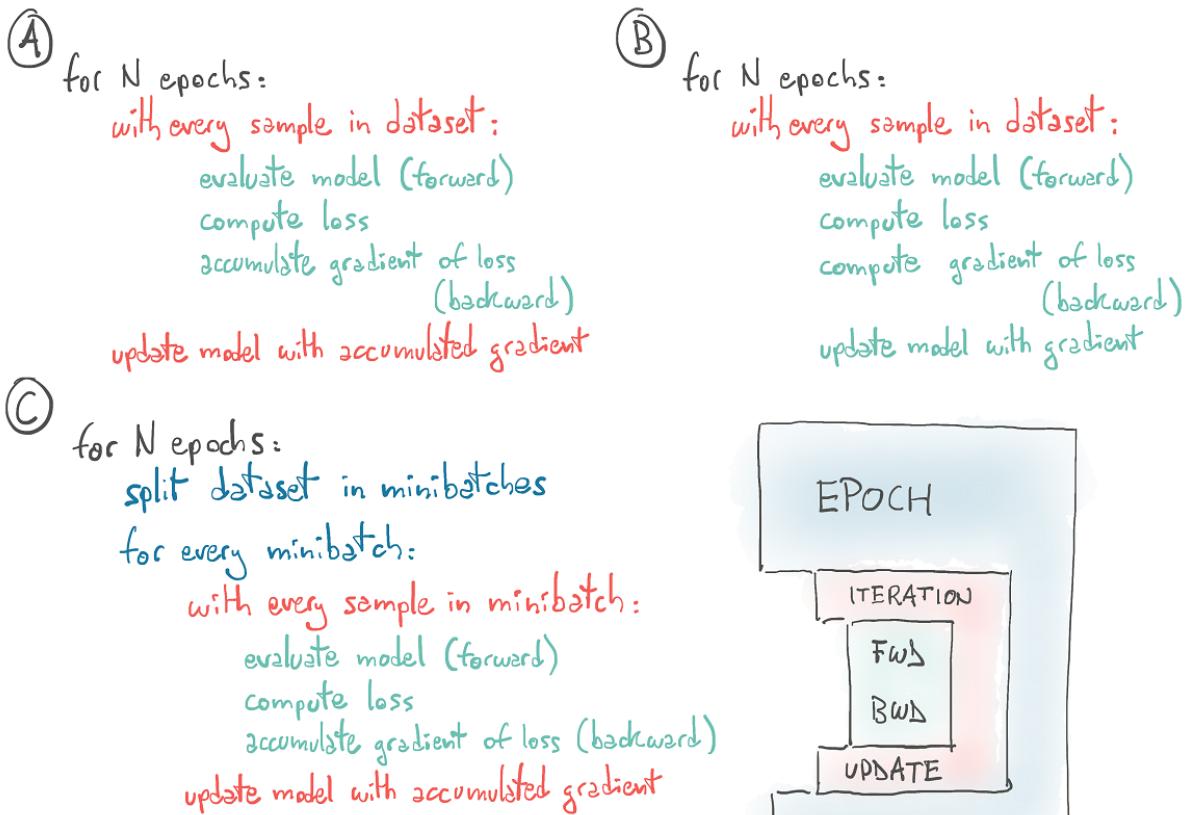


Figure 7.10 Training loops: A) averaging updates over the whole dataset; B) updating model at each sample; C) averaging updates over minibatches.

```

>>> import torch
>>> import torch.nn as nn
>>>
>>> model = nn.Sequential(
>>>     nn.Linear(3072, 512),
>>>     nn.Tanh(),
>>>     nn.Linear(512, 2),
>>>     nn.LogSoftmax(dim=1))
>>>
>>> learning_rate = 1e-2
>>>
>>> optimizer = optim.SGD(model.parameters(), lr=learning_rate)
>>>
>>> loss_fn = nn.NLLLoss()
>>>
>>> n_epochs = 100
>>>
>>> for epoch in range(n_epochs):
>>>     for img, label in cifar2:
>>>         out = model(img.view(-1).unsqueeze(0))
>>>         loss = loss_fn(out, torch.tensor([label]))
>>>
>>>         optimizer.zero_grad()
>>>         loss.backward()
>>>         optimizer.step()
>>>
>>>     print("Epoch: %d, Loss: %f" % (epoch, float(loss)))
  
```

Looking more closely, we made a small change to the training loop. In Chapter 5 we had just one

loop, over the epochs (recall that an epoch ends when all samples in the training set have been evaluated). We figured that evaluating all 10000 images in a single batch would be too much, so we decided to have an inner loop where we evaluate one sample at a time and back-propagate over that single sample.

While in the first case the gradient is accumulated over all samples before being applied, in this case we apply changes to parameters based on a very partial estimation of the gradient, on a single sample. However, what is a good direction for descending the loss based on one sample might not be a good direction for others. By shuffling samples at each epoch and estimating the gradient on one or (preferably, for stability) a few samples at a time, we are effectively introducing randomness in our gradient descent. Remember SGD? It stands for Stochastic Gradient Descent, and this is what the S is about: working on small batches (aka minibatches) of shuffled data. It turns out that following gradients estimated over minibatches, which are poorer approximations of the gradients estimated across the whole dataset, helps convergence and prevents the optimization process from getting stuck in local minima it encounters along the way. This way, gradients will be randomly off the ideal trajectory, which is part of the reason why we want to use a reasonably small learning rate. Shuffling the dataset at each epoch helps ensuring that the sequence of gradients estimated over minibatches is representative of the gradients computed across the full dataset.

Typically the size of minibatches is a constant that we need to set prior to training, just like the learning rate. These are called *hyperparameters*, to distinguish them from the parameters of a model.

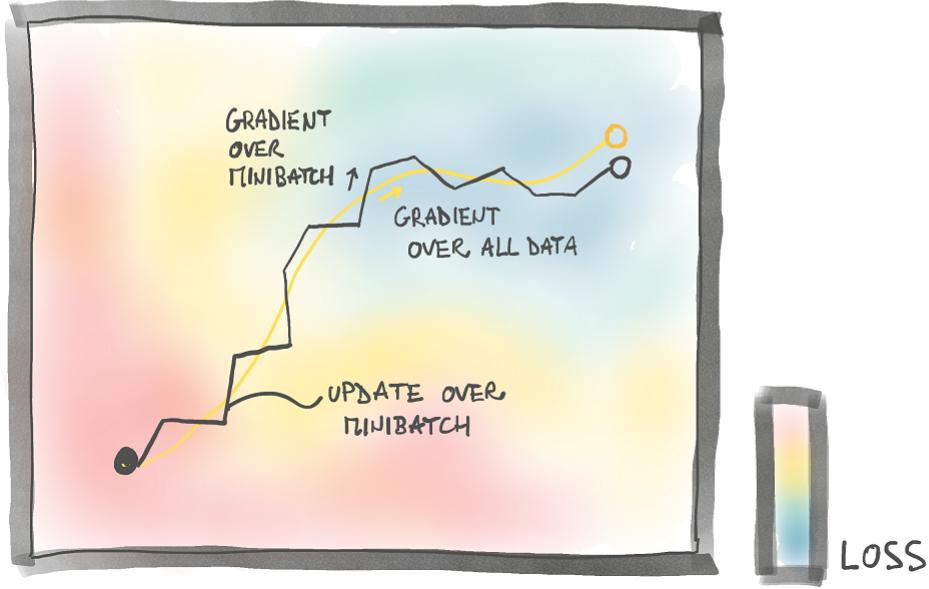


Figure 7.11 Gradient descent averaged over the whole dataset (light path) vs stochastic gradient descent, where gradient is estimated on randomly picked minibatches.

In our training code above we chose minibatches of 1 by just picking one item at a time from the Dataset. The `torch.utils.data` module has a class that helps with shuffling and organizing the data in minibatches: the `DataLoader`. The job of a `DataLoader` is to sample minibatches from a Dataset, giving the flexibility to choose from different sampling strategies, a very common one being uniform sampling after shuffling the data at each epoch.

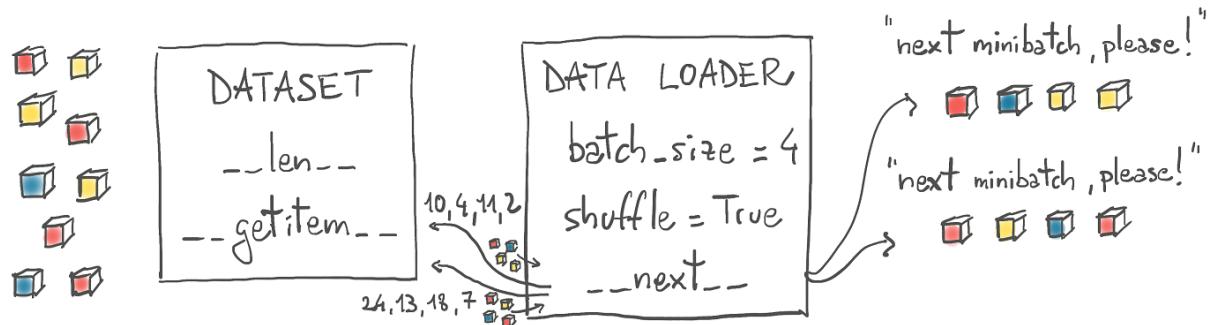


Figure 7.12 DataLoader, dispensing minibatches by using Dataset to sample individual data items.

Let's see how this is done. At a minimum, the `DataLoader` constructor takes a `Dataset` in input,

along with a `batch_size` and a `shuffle` boolean that indicates whether the data needs to be shuffled at the beginning of each epoch:

```
>>> train_loader = torch.utils.data.DataLoader(cifar2, batch_size=64, shuffle=True)
```

A `DataLoader` can be iterated over, so we can use it directly in the inner loop of our new training code:

```
>>> import torch
>>> import torch.nn as nn
>>>
>>> train_loader = torch.utils.data.DataLoader(cifar2, batch_size=64, shuffle=True)
>>>
>>> model = nn.Sequential(
>>>     nn.Linear(3072, 512),
>>>     nn.Tanh(),
>>>     nn.Linear(512, 2),
>>>     nn.LogSoftmax(dim=1))
>>>
>>> learning_rate = 1e-2
>>>
>>> optimizer = optim.SGD(model.parameters(), lr=learning_rate)
>>>
>>> loss_fn = nn.NLLLoss()
>>>
>>> n_epochs = 100
>>>
>>> for epoch in range(n_epochs):
>>>     for imgs, labels in train_loader:
>>>         batch_size = imgs.shape[0]
>>>         outputs = model(imgs.view(batch_size, -1))
>>>         loss = loss_fn(outputs, labels)
>>>
>>>         optimizer.zero_grad()
>>>         loss.backward()
>>>         optimizer.step()
>>>
>>>     print("Epoch: %d, Loss: %f" % (epoch, float(loss)))
```

At each inner iteration, `imgs` will be a tensor of size $64 \times 3 \times 32 \times 32$, i.e. a minibatch of 64 32x32 RGB images, while `labels` a tensor of size 64 containing label indices.

Let's run our training.

```
Epoch: 0, Loss: 0.523478
Epoch: 1, Loss: 0.391083
Epoch: 2, Loss: 0.407412
Epoch: 3, Loss: 0.364203
...
Epoch: 96, Loss: 0.019537
Epoch: 97, Loss: 0.008973
Epoch: 98, Loss: 0.002607
Epoch: 99, Loss: 0.026200
```

Alright, we see that the loss decreases somehow, yet we have no idea on whether it's low enough or not. Since our goal here is to correctly assign classes to images, and preferably do that on an independent dataset, we can compute the accuracy of our model on the validation set in terms of number of correct classifications over the total.

```

>>> val_loader = torch.utils.data.DataLoader(cifar2_val, batch_size=64, shuffle=False)
>>>
>>> correct = 0
>>> total = 0
>>>
>>> with torch.no_grad():
>>>     for imgs, labels in val_loader:
>>>         batch_size = imgs.shape[0]
>>>         outputs = model(imgs.view(batch_size, -1))
>>>         _, predicted = torch.max(outputs, dim=1)
>>>         total += labels.shape[0]
>>>         correct += int((predicted == labels).sum())
>>>
>>> print("Accuracy: %f", correct / total)

Accuracy: 0.794000

```

Not a great performance, but still quite a lot better than random. In our defense, our model was quite a shallow classifier, it's a miracle that it even worked at all. It did because our dataset is really simple, a lot of the samples in the two classes likely have systematic differences (e.g. color of the background) that helps the model telling birds from airplanes from a few pixels.

We can certainly add some bling to our model by including more layers, which would increase the depth and the capacity of the model. One, rather arbitrary, possibility is:

```

>>> model = nn.Sequential(
>>>     nn.Linear(3072, 1024),
>>>     nn.Tanh(),
>>>     nn.Linear(1024, 512),
>>>     nn.Tanh(),
>>>     nn.Linear(512, 128),
>>>     nn.Tanh(),
>>>     nn.Linear(128, 2),
>>>     nn.LogSoftmax(dim=1))

```

Here we are trying to taper the number of features more gently towards the output, in the hope that intermediate layers better squeeze information in increasingly shorter intermediate outputs.

Since we are at it, we should also mention that the combination of `nn.LogSoftmax` and `nn.NLLLoss` is equivalent to using `nn.CrossEntropyLoss`. In fact, the combined mathematical expression for taking the negative sum of log softmax of a vector of values corresponds to the mathematical expression for *cross entropy*. We won't go into details on what cross entropy is from information theory, as it would take us too far from our trajectory. For us at this time, it's just another name (and another underlying interpretation) for the same quantities. At this point we can remove `nn.LogSoftmax` from the model above and switch to the new loss.

```

>>> model = nn.Sequential(
>>>     nn.Linear(3072, 1024),
>>>     nn.Tanh(),
>>>     nn.Linear(1024, 512),
>>>     nn.Tanh(),
>>>     nn.Linear(512, 128),
>>>     nn.Tanh(),
>>>     nn.Linear(128, 2))

>>> loss_fn = nn.CrossEntropyLoss()

```

Note that the numbers will be *exactly* the same as with `nn.LogSoftmax` and `nn.NLLLoss`. It's just more convenient to do it all in one pass, with the only gotcha that the output of our model will not be interpretable as probabilities (or log probabilities). We'll need to explicitly pass the output through a softmax to obtain those.

Training this model and evaluating the accuracy on the validation set (0.802000) lets us appreciate that a larger model bought us an increase in accuracy, but not that much. The accuracy on the training set is practically perfect (0.998100). What is this telling us? That we are overfitting our model in both cases. Our fully connected model is finding a way to discriminate birds and airplanes on the training set by memorizing the training set, but the performance on the validation set is not all that great, even if we choose a larger model.

PyTorch offers a quick way to determine how many parameters a model has through the `parameters()` method of `nn.Model` (the same method we use to provide the parameters to the optimizer). To find out how many elements are in each tensor instance we can call the `numel` method. Summing those gives us our total count. Depending on our use case, counting parameters might require us to check if a parameter actually has `requires_grad` set to `True`, as well. We might want to differentiate the number of *trainable* parameters from the overall model size. Let's take a look at what we have right now:

```
# In[7]:
numel_list = [p.numel() for p in connected_model.parameters() if p.requires_grad == True]
sum(numel_list), numel_list

# Out[7]:
(3737474, [3145728, 1024, 524288, 512, 65536, 128, 256, 2])
```

Wow, 3.7 million parameters! Not a small network for such a small input image, isn't it? Even our first network was pretty large:

```
# In[9]:
numel_list = [p.numel() for p in first_model.parameters()]
sum(numel_list), numel_list

# Out[9]:
(1574402, [1572864, 512, 1024, 2])
```

The number of parameters in our first model is roughly half of our latest model. Well, from the list of individual parameter sizes we start having an idea for who's responsible: it's the first module with 1.5 million parameters. In our full network we had 1024 output features, which lead the first linear module to have 3 million parameters. This shouldn't be unexpected: we know that a linear layer computes $y = \text{weight} * \mathbf{x} + \text{bias}$, and if \mathbf{x} has length 3072 (disregarding the batch dimension for simplicity) and y has to have length 1024, then the `weight` tensor needs to be of size 1024x3072 and the `bias` size 1024. And $1024 * 3072 + 1024$ equals 3146752, as we found above. We can verify these quantities directly:

```
# In[10]:
```

```

linear = nn.Linear(3072, 1024)

linear.weight.shape, linear.bias.shape

# Out[10]:
(torch.Size([1024, 3072]), torch.Size([1024]))

```

What is this telling us? That our neural network won't scale very well with the number of pixels. What if we had a 1024x1024 RGB image? That's 3.1 million input values. Even abruptly going to 1024 hidden features (which is not going to work for our classifier), we would have over 3 *billion* parameters. Using 32-bit floats, we're already at 12GB of RAM, and we haven't even hit the second layer, much less computed and stored the gradients. That's just not going to fit on most present-day GPUs.

7.2.5 The limits of going fully connected

Let's reason about what using a linear module on a 1D view of our image entails. It's like taking every single input value, that is, every single component in our RGB image, and computing a linear combination of it with all the other values for every output feature. On one hand, we are allowing for the combination of any pixel with every other pixel in the image being potentially relevant for our task. On the other hand, we aren't utilizing the relative position of neighboring or far-away pixels, since we are treating the image as one big vector of numbers.

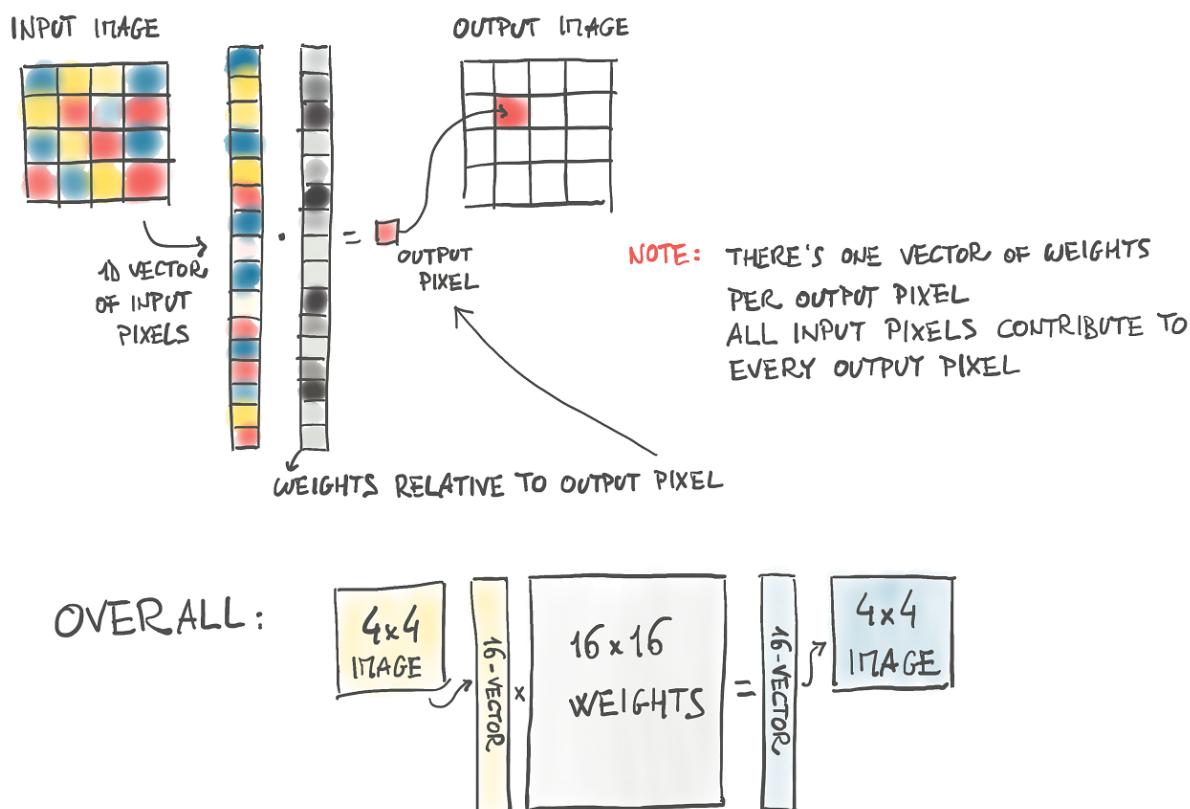


Figure 7.13 Using a fully connected module with an input image: every input pixel is combined with every other to produce each element in the output.

An airplane flying in the sky captured in a 32x32 image will look very roughly like a dark, cross-like shape on a blue background. A fully connected network like the above would need to learn that when pixel 0,1 is dark, pixel 1,1 is also dark, and so on, that's a good indication that there's an airplane there. However, shift the same airplane by one pixel or more, and the relationship between pixels will have to be re-learnt from scratch: this time it's when 0,2 is dark, pixel 1,2 is dark, etc. In more technical terms, a fully connected network is not *translation-invariant*. This means that a network that has been trained to recognize a Spitfire starting at position 4,4 will not be able to recognize the *exact same* Spitfire starting at position 8,8.

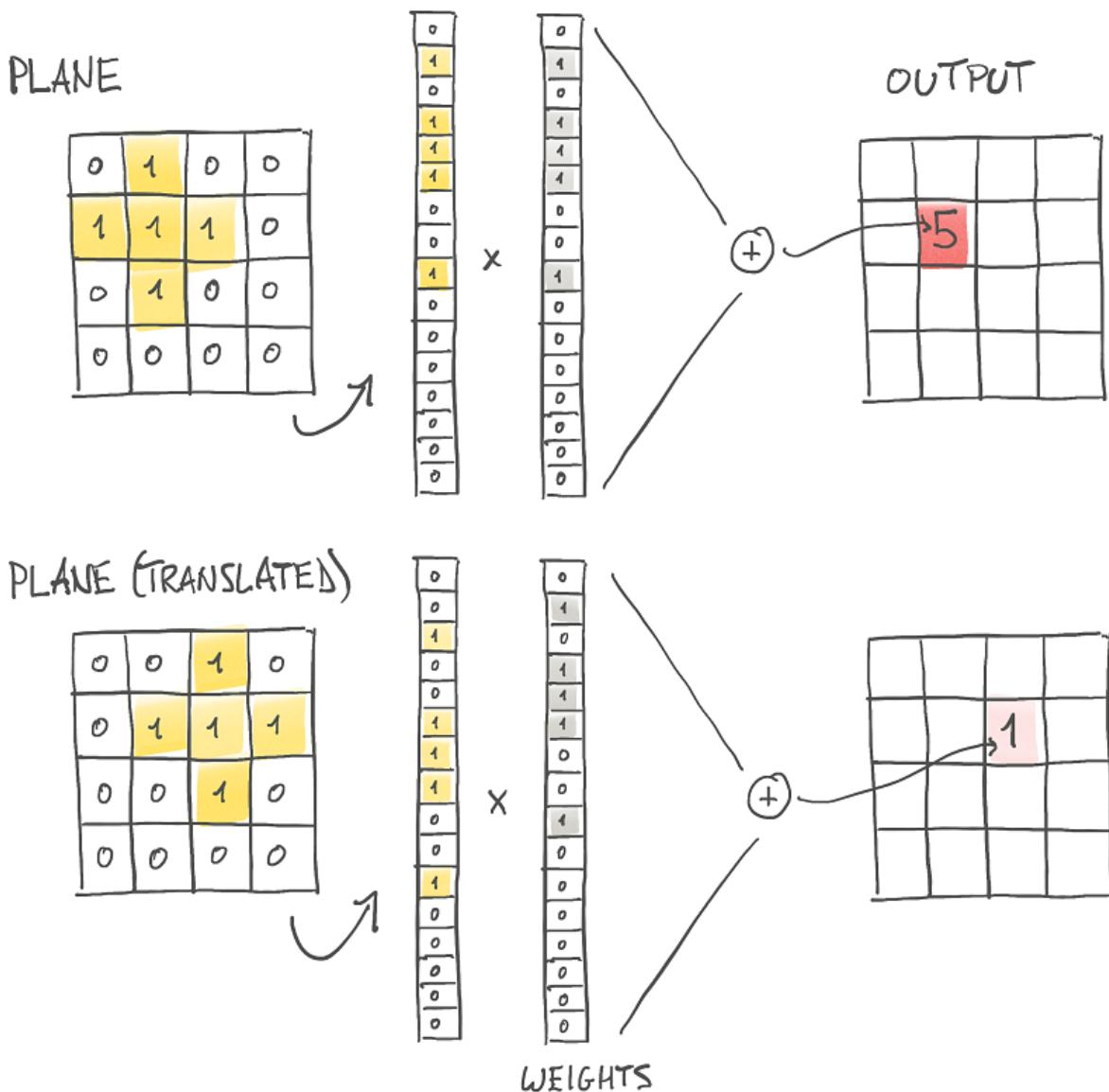


Figure 7.14 Translation invariance, or the lack thereof with fully connected layers.

We would then have to *augment* the dataset, i.e. apply random translations to images during training, so that the network has a chance to see Spitfires all over the image, and this should be done for every image in the dataset (for the record, we could concatenate a transform from

`torchvision.transforms` to do this transparently). However, this data augmentation strategy comes at a cost: the number of hidden features, i.e. of parameters, has to be large enough to store the information about all these translated replicas.

7.3 Conclusion

In this chapter, we have obtained a standard dataset and adapted it for our use case by screening out all images that were not a bird or airplane. We've then normalized that data, and fed it into a fully-connected classification model. That is done in a loop, along with computing the loss and updating the model's parameters, resulting in an (eventually) trained network. All of these things are going to be standard tools for your PyTorch toolbelt, and the skills needed to use them will be useful throughout your PyTorch tenure. However, due to a mismatch between our problem and our network structure, we end up over-fitting our training data, rather than learning the generalized features of what we want the model to detect.

We created a model that allows for relating every pixel to every other pixel in the image, regardless of their spatial arrangement. We have a reasonable assumption that pixels that are closer together are in theory a lot more related, though. This means that we are training a classifier that is not translation-invariant, so we're forced to use a lot of capacity for learning translated replicas if we want to hope to do well on the validation set. There has to be a better way, right?

Of course, most questions like that in a book like this are rhetorical. The solution to our current set of problems is to change our model to use convolutional layers. We'll cover what that means in the next chapter.

7.4 Exercises

- Use `torchvision` to implement random cropping of the data.
 - How are the resulting images different from the uncropped originals?
 - What happens when you request the same image a second time?
 - What is the result of training using randomly cropped images?
- Switch loss functions (perhaps MSE).
 - Does training behavior change?
- Is it possible to reduce capacity of the network enough so that it stops overfitting?
 - What is the performance of the model on the validation set when doing so?

7.5 Summary

- Computer vision is one of the most extensive applications of deep learning.
- Several datasets of annotated images are publicly available, many of them can be accessed via `torchvision`.
- Datasets and DataLoaders provide a simple, yet effective abstraction for loading datasets and sampling them.
- For a classification task, using the softmax function on the output of a network produces values that satisfy the requirements for being interpreted as probabilities. The ideal loss function for classification in this case is obtained by feeding the output of softmax as the input of a non-negative log likelihood function. The combination of softmax and such loss is mathematically equivalent to cross entropy.
- There's nothing preventing images to be treated as vectors of pixel values and be dealt with using a fully connected network, just like any other numerical data. However, this makes leveraging spatial relationships in the data much harder.
- Simple models can be created either using `nn.Sequential`.

8

Using Convolutions To Generalize

This chapter covers:

- how convolution works
- building a convolutional neural network to classify images
- The difference between the module and functional APIs

In the previous chapter, we built a simple neural network that could fit (or overfit) the data, thanks to the many parameters available for optimization in the linear layers. We had issues with our models, however, in that it was better able to memorize the training set than it was at generalizing properties of birds and airplanes. Based on our model architecture, we've got a guess as to why. Due to the fully connected setup needed to be able to detect the various possible translations of the bird or airplane in the image, we have both too many parameters (making it easier for the model to memorize the training set), and no position independence (making it harder to generalize).

As discussed last chapter, we could augment our training data by using a wide variety of recropped images to try and force generalization, but that won't address the issue of having too many parameters.

8.1 The case for convolutions

There is a better way! It consists in replacing the dense, fully-connected affine transformation in our neural network unit with a different linear operation: convolution. Let's get to the bottom of what convolutions are and how we can use them in our neural networks. Yes, yes, we were in the middle of our quest to tell birds from airplanes, and our friend is still there waiting for our solution, but this diversion is worth the extra time spent. We'll develop an intuition for this foundational concept in computer vision and we'll return to our problem equipped with superpowers.

We said above that taking a 1D view of our input image and multiplying it with a `n_output_features x n_input_features` weight matrix, as done in `nn.Linear`, means taking all pixels in the image, and for each channel computing a weighted sum of all pixels multiplied by a set of weights, one per output feature.

We also said that, if we ought to recognize patterns corresponding to objects, like an airplane in the sky, we will likely need to look at how nearby pixels are arranged, and we would be less interested at how pixels that are far from each other appear in combination. Essentially, it doesn't matter if our image of a Spitfire has a tree or cloud or kite in the corner or not.

In order to translate this intuition in mathematical form, we could compute the weighted sum of a pixel with its immediate neighbors, rather than with all other pixels in the image. This would be equivalent to building weight matrices, one per output feature and output pixel location, in which all weights beyond a certain distance from a center pixel are zero. This will still be a weighted sum, i.e. a linear operation.

There's one more desired property we identified above: we would like these localized patterns to have an effect on the output no matter their location in the image, i.e. to be translation-invariant. To do that, we would need to force the weights in each per-output-pixel family of patterns to have same values, regardless of pixel location. To achieve this goal, we would need to initialize all weight matrices in a family with the same values, and, during back-propagation, average the gradients for all pixel locations and apply that average as the update to all weights in the family.

This way we would have ensured that a) weights operate in neighborhoods to respond to local patterns, b) local patterns are identified no matter where they occur in the image.

The explanation above is a way to realize that it's possible to turn a fully-connected linear operation on individual values of an image into a local, translation-invariant operation on the image, a *convolution*. In fact, we can come up with a more compact description of a convolution, but we'll realize that what we are going to describe is exactly what we just delineated, only taken from a different angle.

Convolution, or more precisely discrete convolution (there's an analogous continuous version

that we won't go into here), is defined for a 2D image as the scalar product of a weight matrix, the *kernel*, with every neighborhood in the input. Consider a 3x3 kernel (in deep learning we typically use small kernels; we'll see why later on) as a 2D tensor:

```
weight = torch.tensor([[w00, w01, w02],
                      [w10, w11, w12],
                      [w20, w21, w22]])
```

and a 1-channel, MxN image:

```
image = torch.tensor([[i00, i01, i02, i03, ..., i0N],
                      [i10, i11, i12, i13, ..., i1N],
                      [i20, i21, i22, i23, ..., i2N],
                      [i30, i31, i32, i33, ..., i3N],
                      ...
                      [iM0, iM1, iM2, iM3, ..., iMN]])
```

We can compute an element of the output image (without bias) as:

```
o11 = i11 * w00 + i12 * w01 + i22 * w02 +
      i21 * w10 + i22 * w11 + i23 * w12 +
      i31 * w20 + i32 * w21 + i33 * w22
```

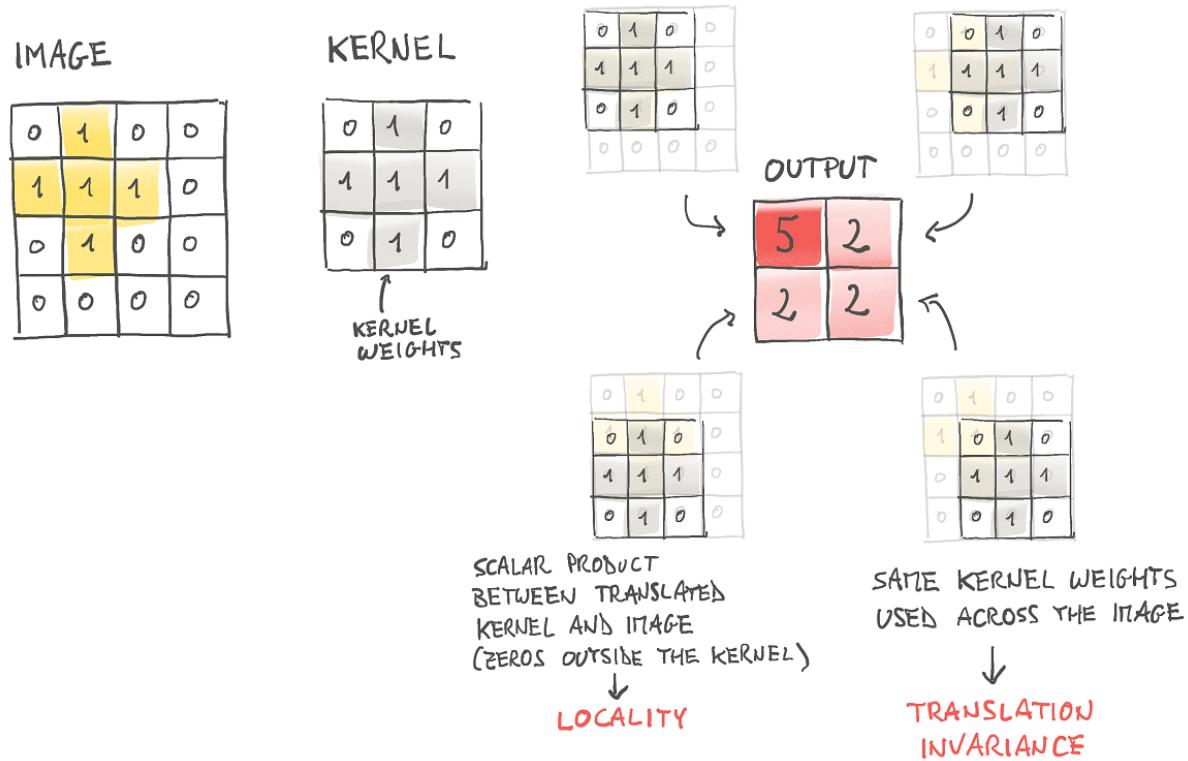


Figure 8.1 Convolution: locality and translation invariance.

That is, we "translate" the kernel on the i_{11} location of the input image, and we multiply each weight with the value of the input image at the corresponding location. Thus, the output image is

created by translating the kernel on all input locations and performing the weighted sum above. For a multi-channel image, like our RGB image, the weight matrix would be a 3x3x3 matrix, i.e. one set of weights for every channel, contributing together to the output values.

Note that, just like the elements in the `weight` matrix of `nn.Linear`, the weights in the kernel are not known in advance, but they are initialized randomly and updated through back-propagation.

Note also that the same kernel, thus each weight in the kernel, is re-used across the whole image. Thinking back to autograd, this means that the use of each weight has an history spanning the entire image. Thus, the derivative of the loss with respect to a convolution weight includes contributions from the entire image.

It's now possible to see the connection to what we were stating above: a convolution is equivalent to having multiple linear operations, whose weights are zero almost everywhere except around individual pixels and which receive equal updates during training.

Summarizing, by switching to convolutions we got:

1. local operations on neighborhoods
2. translation-invariance
3. models with a lot fewer parameters

The key insight underlying the third point is that, with a convolution layer, the number of parameters does not depend on the number of pixels in the image, as it was in our fully connected model, but on the size of the convolution kernel (3x3, 5x5 etc) and on how many convolution filters (or output channels) we decide to use in our model.

8.2 Convolutions in action

Well, it looks like we rabbit holed enough! It's now time to see some PyTorch in action on our birds vs airplanes challenge. The `torch.nn` module provides convolution for 1, 2 and 3 dimensions: `nn.Conv1d` for time series, `nn.Conv2d` for images, and `nn.Conv3d` for volumes or videos.

For our CIFAR-10 data we'll resort to `nn.Conv2d`. At a minimum, the arguments we provide to `nn.Conv2d` are the number of input features (or *channels*, since we're dealing with so-called multi-channel images, i.e. more than one value per pixel), the number of output features, and the size of the kernel. For instance, for our first convolutional module we'll have 3 input features per pixel (the RGB channels) and an arbitrary number of channels in output, say 16. The more channels in the output image, the more the capacity of the network. Let's stick to a kernel size of 3x3.

```
# In[11]:
```

```
conv = nn.Conv2d(3, 16, kernel_size=3) ❶
conv

# Out[11]:
Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1))
```

- ❶ The `kernel_size=3` here is a shortcut for "3 in every dimension"; here 2D, so 3x3. Otherwise, a tuple of length 2 is needed.

We can expect a weight tensor sized the following way: we'll have as many kernels, sized `n_input_channels x3x3`, as the number of output channels. That is, we expect the weight tensor to be sized `n_input_channels x3x3x n_output_channels`, so `3x3x3x16`. The bias (we haven't talked about it for a while for simplicity, but just like in the linear module case it's a constant value we add to each channel of the output image) will have size 16. Let's verify our assumptions:

```
# In[12]:
conv.weight.shape, conv.bias.shape

# Out[12]:
(torch.Size([16, 3, 3, 3]), torch.Size([16]))
```

We can see how convolutions are a convenient choice for learning from images. We have smaller models looking for local patterns, whose weights are optimized across the entire image.

A 2D convolution pass produces a 2D image in output, whose pixels are a weighted sum over neighborhoods of the input image. In our case, both the kernel weights and the bias `conv.weight` are initialized randomly, so the output image will not be particularly meaningful. As usual, we need to add the zero-th batch dimension with `unsqueeze` if we want to call the `conv` module with one input image, since `nn.Conv2d` expects a `BxCxHxW` shaped tensor in input:

```
# In[13]:
img, _ = cifar2[0]
output = conv(img.unsqueeze(0))
img.unsqueeze(0).shape, output.shape

# Out[13]:
(torch.Size([1, 3, 32, 32]), torch.Size([1, 16, 30, 30]))
```

We're curious, so we can display the output:

```
# In[15]:
plt.imshow(output[0, 0].detach(), cmap='gray')
plt.show()
```

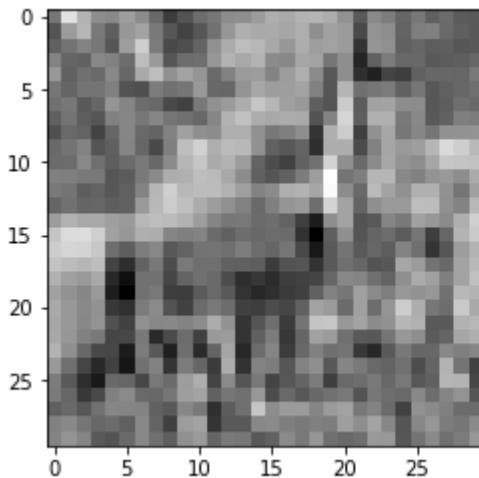


Figure 8.2 Our bird after a random convolution treatment.

Wait a minute, let's take a look at the size of `output`, it was `torch.Size([1, 16, 30, 30])`. Huh, we lost a few pixels in the process, how's that?

It's a side effect of deciding what to do at the boundary of the image. Applying a convolution kernel as a weighted sum of pixels in a 3x3 neighborhood requires that there are neighbors in all directions. If we are at `i00`, we only have pixels right of and below us. By default, PyTorch will skip pixels at the boundary, thereby producing images that are one half of the convolution kernel width (in our case, $3 // 2 = 1$) smaller on each side. This explains why we're missing two pixels in each dimension.

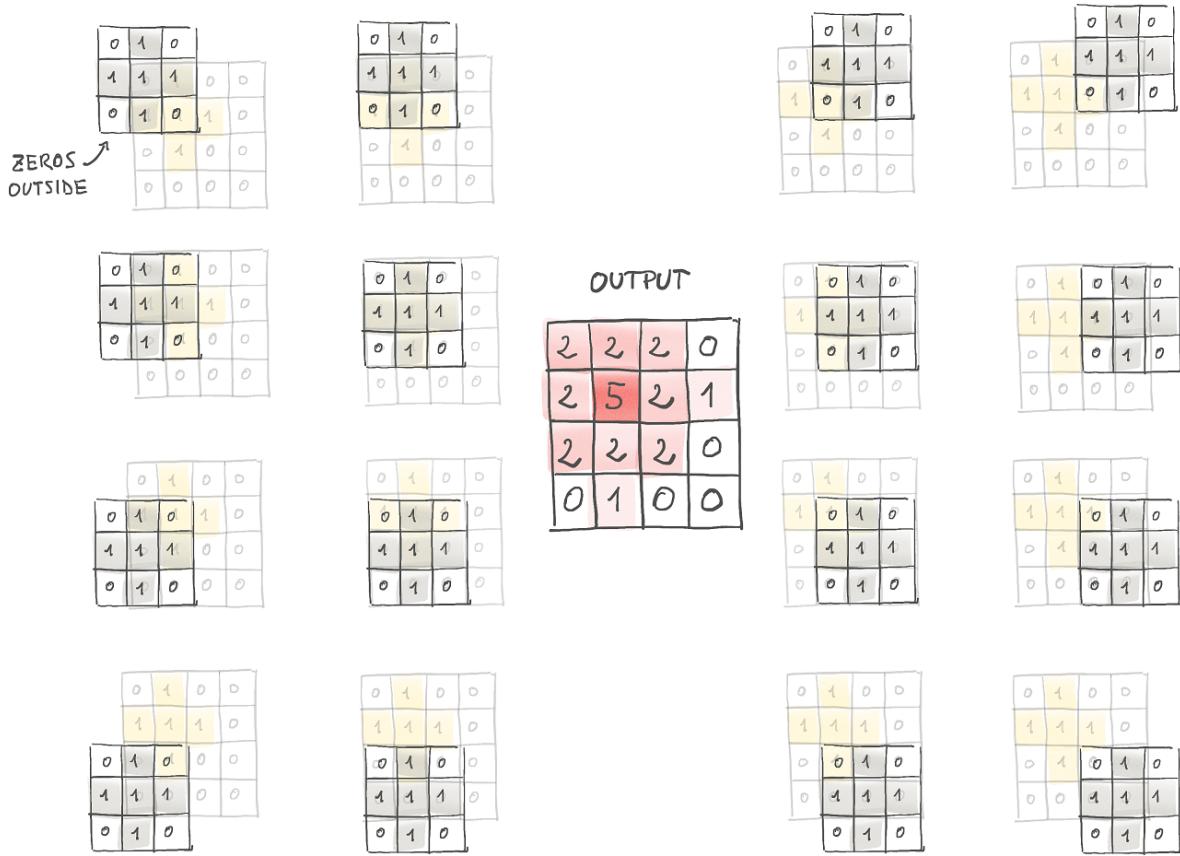


Figure 8.3 Zero padding to preserve image size to the output.

However, PyTorch gives us the possibility of *padding* the image, creating *ghost pixels* around the border that have value zero as far as the convolution is concerned. For instance, specifying `padding=1` when `kernel_size=3` means that now `i00` has an extra set of neighbors above and left, so that an output of the convolution can be computed even in the corner of our original image. The net result is that the output has now the exact same size as the input:

```
# In[16]:
conv = nn.Conv2d(3, 1, kernel_size=3, padding=1) ①
output = conv(img.unsqueeze(0))
img.unsqueeze(0).shape, output.shape

# Out[16]:
(torch.Size([1, 3, 32, 32]), torch.Size([1, 1, 32, 32]))
```

① Now with padding.

Note that the size of `weight` and `bias` don't change whether padding is used or not.

We said above that `weight` and `bias` are `Parameters` that will be learned through back-propagation, exactly as it happens for `weight` and `bias` in `nn.Linear`. However, we can play with convolution by setting weights by hand and see what happens.

Let's first zero out the `bias`, just to remove any confounding factor, then set `weights` so that

each pixel in the output gets the mean of its neighbors, for each 3x3 neighborhood:

```
# In[17]:
with torch.no_grad():
    conv.bias.zero_()

with torch.no_grad():
    conv.weight.fill_(1.0 / 9.0)
```

We could have gone with `conv.weight.one_()` - this would have resulted in each pixel in the output to be the *sum* of the pixels in the neighborhood. Not a big difference, except that the values in the output image would have been nine times larger.

Anyway, let's see the effect we get on our CIFAR image:

```
# In[18]:
output = conv(img.unsqueeze(0))
plt.imshow(output[0, 0].detach(), cmap='gray')
plt.show()

# Out[18]:
<Figure size 432x288 with 1 Axes>
```

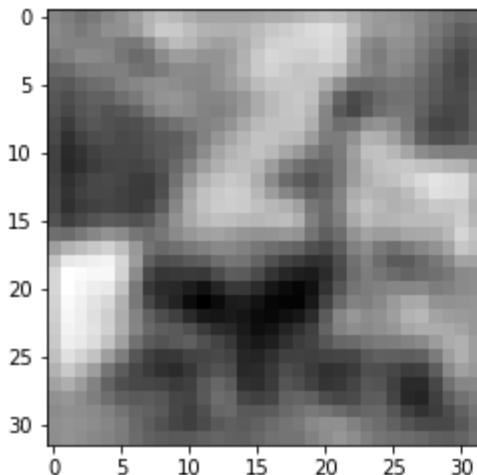


Figure 8.4 Our bird, this time blurred thanks to a constant convolution kernel.

As we could have predicted, the filter has produced a blurred version of the image. After all, every pixel of the output is the average of a neighborhood of the input, so output pixels of the output will be correlated and change more smoothly.

We can try with something different, like

```
# In[19]:
conv = nn.Conv2d(3, 1, kernel_size=3, padding=1)

with torch.no_grad():
    conv.weight[:] = torch.tensor([[-1.0, 0.0, 1.0],
                                  [-1.0, 0.0, 1.0],
                                  [-1.0, 0.0, 1.0]])
    conv.bias.zero_()
```

Working out the weighted sum for an arbitrary pixel in position $2, 2$, as we did above for the generic convolution kernel, we get

```
o22 = i13 - i11 +
      i23 - i21 +
      i33 - i31
```

which performs the difference of all pixels on the right of i_{22} , minus the pixels on the left of i_{22} . In case there's a vertical boundary between two adjacent regions of different intensity, o_{22} will have a high value. If the kernel is applied on a region of uniform intensity, o_{22} will be zero. It's a so-called *edge detection* kernel: the kernel highlights the vertical edge between two horizontally adjacent regions.

Applying the convolution kernel to our image we see:

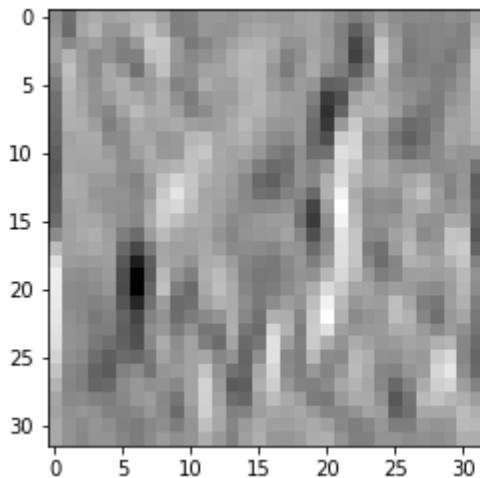


Figure 8.5 Vertical edges throughout our bird, courtesy of a hand-crafted convolution kernel.

As expected, the convolution kernel enhances vertical edges. We could build lots of more elaborate filters, e.g. for detecting horizontal or diagonal edges, cross-like or checkerboard patterns, whereby "detecting" means that the output has a high magnitude. In fact, the job of a computer vision expert has historically been to come up with the most effective combination of filters so that certain features were highlighted in images and objects could be recognized.

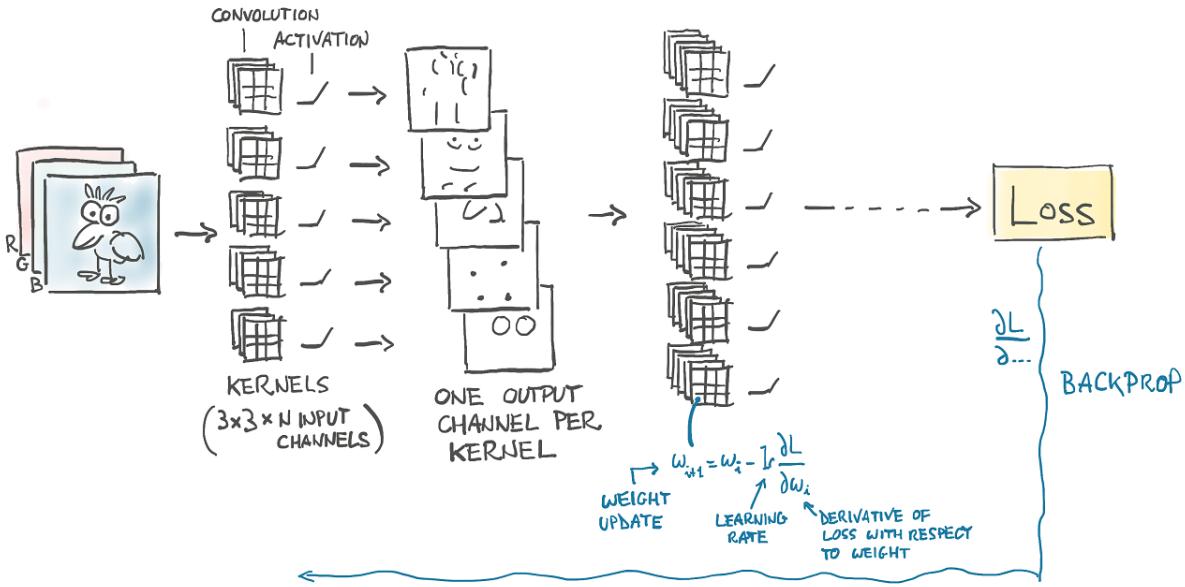


Figure 8.6 The process of learning with convolutions, by estimating the gradient at the kernel weights and updating them individually in order to optimize for the loss.

With deep learning, we let kernels get estimated from data, in a way that the discrimination is most effective. For instance, in terms of non-negative log likelihood of the output compared to ground truth. From this angle, the job of a convolutional neural network is to estimate the kernel of a set of filter banks in successive layers, that will transform a multi-channel image into another multi-channel image, where different channels will correspond to different features (e.g. one channel for the average, another channel for vertical edges, etc).

8.2.1 Looking further with depth and pooling

This is all well and good, but conceptually there's an elephant in the room here. We got all excited that by moving from fully connected layers to convolutions we achieve locality and translation-invariance. Then we recommended the use of small kernels, like 3x3, or 5x5: that's peak locality, alright. What about the *big picture* though? How do we know that all structures in our images are 3 pixels or 5 pixels wide? Well, we don't, because they aren't. And so if they aren't, how are our networks going to be equipped to see those patterns with larger scope?

This is something we'll really need if we want to solve our bird vs airplanes problem effectively, since although CIFAR-10 images are small, the objects still have a (wing-)span several pixels across.

One possibility could be to use large convolution kernels. Well, sure, at the limit we could get a 32x32 kernel for a 32x32 image, but we would converge to the old fully connected, affine transformation and lose all the nice properties of convolution. Another option, which is what is used in convolutional neural networks, is stacking one convolution after the other, and at the same time downsampling the image in-between successive convolutions.

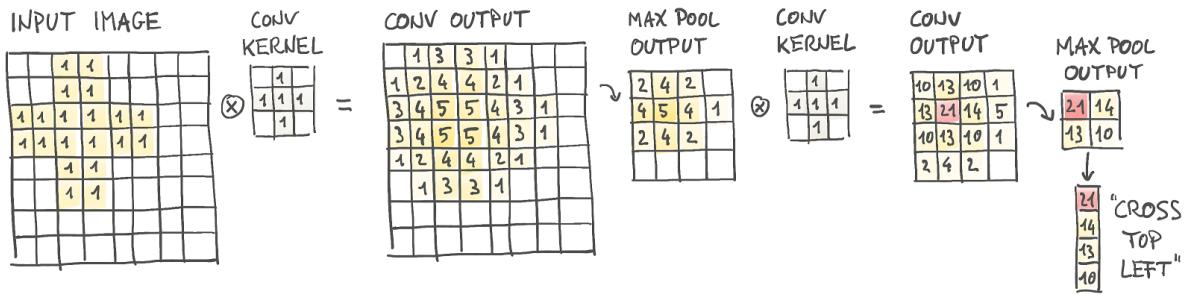


Figure 8.7 Some more convolutions by hand, showing the effect of stacking convolutions and downsampling: a large cross is highlighted using two small cross-shaped kernels and max pooling.

In Figure-6.22 we start with applying a set of 3x3 kernels on our 8x8 image, obtaining a multi-channel output image of the same size. Then we scale down the output image by a half, obtaining a 4x4 image and apply another set of 3x3 kernels to it. This second set of kernels operates on a 3x3 neighborhood of something that has been scaled down by a half, so it effectively maps back to 8x8 neighborhoods of the input. In addition, the second set of kernels takes the output of the first set of kernels (so features like averages, edges, etc) and extracts additional features on top of those.

NOTE

The second 3x3 "conv kernel" filter corresponds to a 6x6 set of pixels from the "conv output" pixels, but each of those pixels is drawn from a 3x3 window from the "input image" pixels. There's a lot of overlap, but each of the edge pixels in the 6x6 is informed by pixels 1 pixel outside of the 6x6 window, which results in a total receptive field of 8x8.

So, on one hand, the first set of kernels operates on small neighborhoods on first-order, low-level features, while the second set of kernels effectively operates on wider neighborhoods, producing features that are compositions of the previous features. This is a very powerful mechanism that provides convolutional neural networks with the ability to see into very complex scenes - much more complex than our 32x32 images from the CIFAR-10 dataset.

Downsampling could in principle occur in different ways. Scaling an image by a half is the equivalent of taking 4 neighboring pixels in input and producing one pixel in output. How we compute the value of the output based on the values of the input is up to us. We could:

- Average the four pixels. This was a common approach early on, but has since fallen out of favor somewhat.
- Take the maximum of the four pixels. This is currently the most commonly used approach, but has a downside of discarding the other 3/4ths of the data.
- Perform a *strided* convolution, where only every Nth pixel is calculated. A 3x4 convolution with stride 2 still incorporates input from all pixels from the previous layer. Current literature shows promise for this approach, but it has not yet supplanted maxpool.

We will be focusing on max pooling going forward.

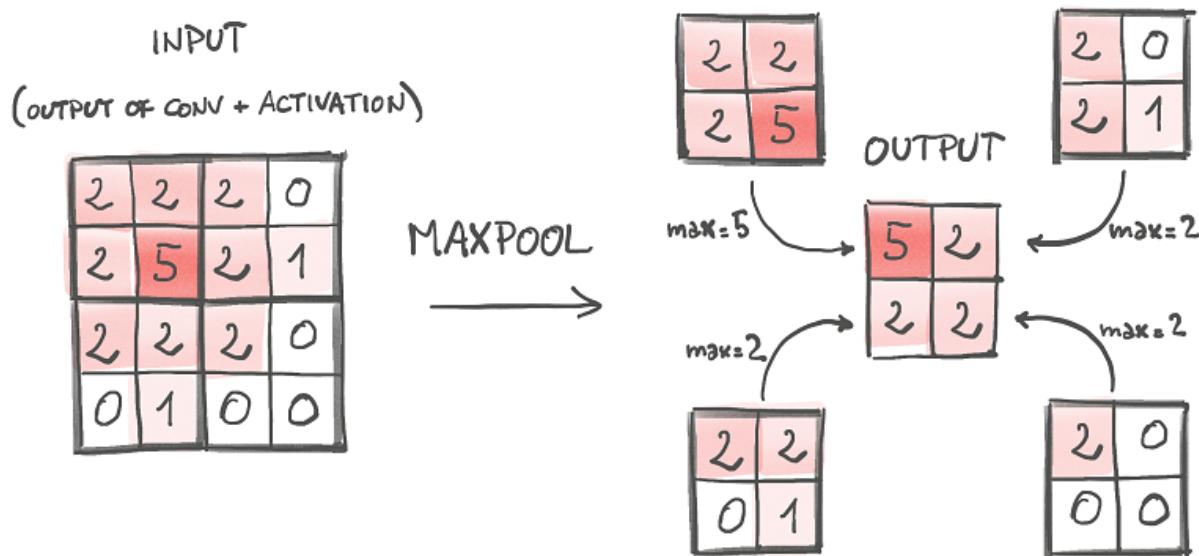


Figure 8.8 Max pooling in detail.

Intuitively, the output images from a convolution layer, especially since they are followed by an activation just like any other linear layer, will tend to have a high magnitude where certain features corresponding to the estimated kernel are detected (e.g. vertical lines). By keeping the highest value in the 2×2 neighborhood as the downsampled output, we ensure that the features that have been found *survive* the downsampling, at the expense of the weaker responses.

Max-pooling is provided by the `nn.MaxPool2d` module (as with convolution, there are versions for 1D and 3D data). It takes in input the size of the neighborhood over which to operate the pooling operation. If we wish to downsample our image by a half, we'll want to use a size of 2. Let's verify it works as expected directly on our input image:

```
# In[21]:
pool = nn.MaxPool2d(2)
output = pool(img.unsqueeze(0))

img.unsqueeze(0).shape, output.shape

# Out[21]:
(torch.Size([1, 3, 32, 32]), torch.Size([1, 3, 16, 16]))
```

With building blocks in our hands, we can now proceed to building our convolutional neural network for detecting birds and airplanes. Let's take our previous fully-connected model as a starting point and introduce `nn.Conv2d` and `nn.MaxPool2d` as described above:

```
# In[22]:
model = nn.Sequential(
    nn.Conv2d(3, 16, kernel_size=3, padding=1),
    nn.Tanh(),
    nn.MaxPool2d(2),
    nn.Conv2d(16, 8, kernel_size=3, padding=1),
    nn.Tanh(),
    nn.MaxPool2d(2),
```

```
# ...
)
```

We have gone from three RGB channels to 16, thereby giving the network a chance to generate 16 independent features that will operate to hopefully discriminate low-level features of birds and airplanes. After the activation function is applied, the 16-channel 32x32 image will be pooled to a 16-channel 16x16 image. At this point, the downsampled image will undergo another convolution that will generate a 8-channel 16x16 output, which, again after activation, will be pooled to a 8-channel 8x8 output.

Where does this end? After the input image has been reduced to a set of 8x8 features, we expect to be able to output some probabilities from the network, so that we can feed them to our negative log likelihood. However, probabilities are a pair of numbers in a 1D vector (one for airplane, one for bird), while here we're still dealing with multi-channel 2D features.

Thinking back at the beginning of this chapter, we already know what we need to do: turn the 8-channel 8x8 image into a 1D vector and complete our network with a set of fully connected layers.

```
# In[23]:
model = nn.Sequential(
    nn.Conv2d(3, 16, kernel_size=3, padding=1),
    nn.Tanh(),
    nn.MaxPool2d(2),
    nn.Conv2d(16, 8, kernel_size=3, padding=1),
    nn.Tanh(),
    nn.MaxPool2d(2),
    # ... ❶
    nn.Linear(8 * 8 * 8, 32),
    nn.Tanh(),
    nn.Linear(32, 2))
```

- ❶ WARNING: Something important is missing here!

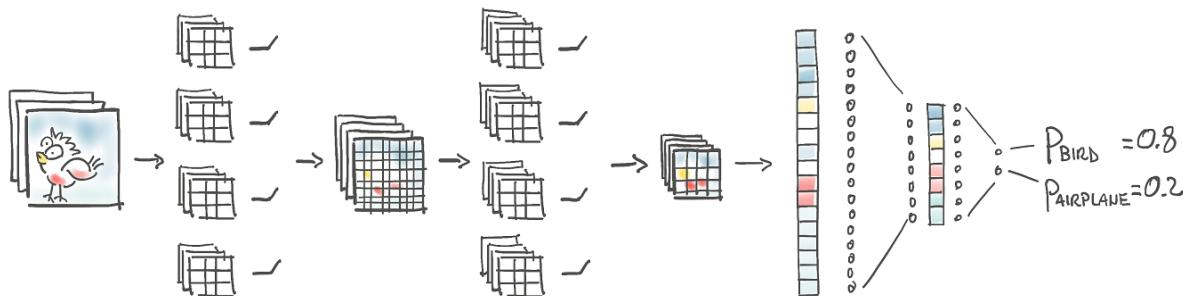


Figure 8.9 Shape of a typical convolutional network, including the one we're building. An image is fed to a series of convolutions and max pooling modules, then straightened into a 1D vector and fed into fully connected modules.

Ignore the "something missing" comment for a minute. Let's first notice that the size of the linear layer is dependent on the expected size of the output of `MaxPool2d`, i.e. $8 \times 8 \times 8 = 512$. Let's count the number of parameters for this small model:

```
# In[24]:
numel_list = [p.numel() for p in model.parameters()]
sum(numel_list), numel_list

# Out[24]:
(18090, [432, 16, 1152, 8, 16384, 32, 64, 2])
```

That's very reasonable for a limited dataset of such small images. In order to increase the capacity of the model we could increase the number of output channels for the convolution layers (i.e. the number of features that each convolution layers generates), which would lead the linear layer to increase its size as well.

The `WARNING` comment was there for a reason. The model above has zero chances of running without complaining.

```
# In[25]:
model(img.unsqueeze(0))

# Out[25]:
...
RuntimeError: size mismatch, m1: [64 x 8], m2: [512 x 32] at c:\...\THTensorMath.cpp:940
```

Admittedly, the error message is a bit obscure, but not too much. We find references to `linear` in the traceback and looking back at the model the only module that has to have a 512×32 tensor is `nn.Linear(512, 32)`, the first linear module after the last convolution block.

What's missing there is the reshaping step from a 8-channel 8x8 image to a 512-element, 1D vector (1D if we ignore the batch dimension, that is). This could be achieved by calling `view` on the output of the last `nn.MaxPool2d`, but unfortunately we don't have any explicit visibility of the output of each module when we use `nn.Sequential`.⁷⁸

8.3 Subclassing `nn.Module`

We need to leave `nn.Sequential` for something that gives us more flexibility, that is, subclassing `nn.Module`. In order to subclass `nn.Module`, at a minimum we need to define a `forward` function, that takes the input to the module and returns the output. If we use standard torch operations, autograd will take care of the backward pass automatically.

Typically we will want to define the submodules that we use in the `forward` function in the constructor, so they can be called in `forward` and can hold their parameters throughout the lifetime of our module. For instance, we instantiate `nn.Conv2d` and `nn.Linear` in the constructor and use their instances in `forward`:

```
# In[26]:
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.act1 = nn.Tanh()
        self.pool1 = nn.MaxPool2d(2)
```

```

self.conv2 = nn.Conv2d(16, 8, kernel_size=3, padding=1)
self.act2 = nn.Tanh()
self.pool2 = nn.MaxPool2d(2)
self.fc1 = nn.Linear(8 * 8 * 8, 32)
self.act3 = nn.Tanh()
self.fc2 = nn.Linear(32, 2)

def forward(self, x):
    out = self.pool1(self.act1(self.conv1(x)))
    out = self.pool2(self.act2(self.conv2(out)))
    out = out.view(-1, 8 * 8 * 8) ①
    out = self.act3(self.fc1(out))
    out = self.fc2(out)
    return out

```

- ① This reshape is what we were missing earlier.

The `Net` class is equivalent to the `nn.Sequential` model we built above in terms of submodules, but, by writing the `forward` function explicitly, we were able to manipulate the output of `self.pool3` directly and call `view` on it to turn it into a BxN vector. Note that we leave the batch dimension as `-1` in the call to `view`, since in principle we don't know how many samples there'll be in the batch.

Interestingly, assigning an instance of `nn.Module` to an attribute in a `nn.Module`, just like we did in the constructor here, automatically registers the module as a submodule. This allows `Net` to have access to the Parameters of its submodules without further action by the user:

```

# In[27]:
model = Net()

numel_list = [p.numel() for p in model.parameters()]
sum(numel_list), numel_list

# Out[27]:
(18090, [432, 16, 1152, 8, 16384, 32, 64, 2])

```

What happened here is that the `parameters()` call delves into all submodules assigned as attributes in the constructor and recursively calls `parameters()` on them. No matter how nested the submodule, 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 will know how to change parameters so to minimize the loss. We know that story from chapter 5.

Looking back at the implementation of the `Net` class, and thinking about the utility of registering submodules in the constructor so that we can access their parameters, it appears a bit of a waste that we are also registering submodules that have no parameters, like `nn.Tanh` and `nn.MaxPool2d`. Wouldn't just be easier to call directly in the `forward` function, just like we called `view`?

8.3.1 The Functional API

It sure would! And that's why PyTorch has *functional* counterparts of every `nn` module. By "functional" here we mean "having no internal state", or, in other words, "whose output value is solely and fully determined by the value input arguments". Indeed, `torch.nn.functional` provides the many of the same modules we find in `nn`, but with all eventual parameters moved as an argument to the function call. For instance, the functional counterpart of `nn.Linear` 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.

Back to our model, it makes sense to keep using `nn` modules for `nn.Linear` and `nn.Conv2d`, so that `Net` will be able to manage their `Parameter`'s during training. However, we can safely switch to the functional counterparts of pooling and activation, since they have no parameters.

```
# In[28]:
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(16, 8, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(8 * 8 * 8, 32)
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        out = F.max_pool2d(torch.tanh(self.conv1(x)), 2)
        out = F.max_pool2d(torch.tanh(self.conv2(out)), 2)
        out = out.view(-1, 8 * 8 * 8)
        out = torch.tanh(self.fc1(out))
        out = self.fc2(out)
        return out
```

A lot more concise and fully equivalent to the above. Note that it would still make sense to instantiate modules that require several parameters for their initialization in the constructor. For instance, one may want to still initialize `nn.MaxPool2d` as we did before, to reinforce the fact that we're using a kernel of 2. We could actually have just one instance of `nn.MaxPool2d` that we use everywhere - since `nn.MaxPool2d` has no parameters, `self.pool1`, `self.pool2` and `self.pool3` we initialized earlier were essentially the same thing and could be used interchangeably. It's really a matter of style, there's no downside either way.

TIP

While 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 like `max_pool2d` will remain in `torch.nn.functional`.

Let's double check that our model runs at all and then we'll get to the training loop.

```
# In[29]:
```

```

model = Net()
model(img.unsqueeze(0))

# Out[29]:
tensor([[-0.0157,  0.1143]], grad_fn=<AddmmBackward>)

```

We've got two numbers out! Information flows correctly. We might not realize it right now, but in more complex models getting the size of the first linear layer right is sometimes a source of frustration. We've heard stories of famous practitioners banging numbers at random and then relying on error messages from PyTorch to backtrack the correct sizes for their linear layers. Lame, eh? Nah, it's all legit!

8.4 Training our Convnet

We're now at the point where we can assemble our complete training loop. We've added some tracking for accuracy, but the overall structure should feel very similar to what we saw in chapter 5:

```

# In[30]:
import datetime

def training_loop(n_epochs, optimizer, model, loss_fn, train_loader):
    for epoch in range(1, n_epochs + 1):
        loss_train = 0.0
        for imgs, labels in train_loader:
            outputs = model(imgs)
            loss = loss_fn(outputs, labels)

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

            loss_train += loss.item()

        if epoch == 1 or epoch % 10 == 0:
            print('{} Epoch {}, Training loss {}'.format(
                datetime.datetime.now(), epoch, float(loss_train)))

```

- ➊ This will give us the index of the highest value in output.

The substantial changes in our model from last chapter are the fact that now our model is a custom subclass of `nn.Module` and that we're using convolutions. Let's run training for a hundred epochs, while printing the loss. This will probably take twenty minutes or more to finish!

```

# In[31]:
train_loader = torch.utils.data.DataLoader(cifar2, batch_size=64, shuffle=True)

model = Net()
optimizer = optim.SGD(model.parameters(), lr=1e-2)
loss_fn = nn.CrossEntropyLoss()

training_loop(
    n_epochs = 100,
    optimizer = optimizer,
    model = model,
    loss_fn = loss_fn,

```

```

        train_loader = train_loader,
    )

# Out[31]:
2019-07-25 01:34:43.202195 Epoch 1, Training loss 88.46652862429619
2019-07-25 01:34:59.160216 Epoch 10, Training loss 51.45938600599766
2019-07-25 01:35:16.652595 Epoch 20, Training loss 47.65262024104595
2019-07-25 01:35:34.646300 Epoch 30, Training loss 44.35324889421463
2019-07-25 01:35:53.409290 Epoch 40, Training loss 40.993075758218765
2019-07-25 01:36:11.110690 Epoch 50, Training loss 37.8431678712368
2019-07-25 01:36:28.465513 Epoch 60, Training loss 34.52699891477823
2019-07-25 01:36:46.122946 Epoch 70, Training loss 31.98214814811945
2019-07-25 01:37:04.062207 Epoch 80, Training loss 29.737521782517433
2019-07-25 01:37:21.525515 Epoch 90, Training loss 27.143563382327557
2019-07-25 01:37:39.210988 Epoch 100, Training loss 25.34625903889537

```

And then we can take a look at our accuracies on the training and validation datasets:

```

# In[32]:
train_loader = torch.utils.data.DataLoader(cifar2, batch_size=64, shuffle=False)
val_loader = torch.utils.data.DataLoader(cifar2_val, batch_size=64, shuffle=False)

for loader in [train_loader, val_loader]:
    correct = 0
    total = 0

    with torch.no_grad():
        for imgs, labels in loader:
            outputs = model(imgs)
            _, predicted = torch.max(outputs, dim=1) ①
            total += labels.shape[0]
            correct += int((predicted == labels).sum())

    print("Accuracy: %f" % (correct / total))

# Out[32]:
Accuracy: 0.930300
Accuracy: 0.930300

```

Quite a lot better than the fully connected model, and with fewer parameters. This is telling us that this model does a better job at generalizing its task to recognize the subject of images from a new sample, through locality and translation-invariance. We could now let it run for more epochs and see what performance we could squeeze out.

Since we're quite satisfied with our model so far, it would be nice to actually save it, right? Very easy, let's save the model to a file:

```
# In[33]:
torch.save(model.state_dict(), data_path + 'birds_vs_airplanes.pt')
```

The `birds_vs_airplanes.pt` file now contains all the parameters of `model`, that is, weights and biases for the two convolution modules and the two linear modules. So, no structure, just the weights. This means that when we deploy the model in production for our friend, we'll need keep the `model` class handy, create an instance and then load parameters back in it:

```
# In[34]:
loaded_model = Net() ①
loaded_model.load_state_dict(torch.load(data_path + 'birds_vs_airplanes.pt'))
```

```
# Out[34]:  
IncompatibleKeys(missing_keys=[], unexpected_keys=[])
```

- ➊ We will have to make sure that we don't change the definition of `Net` between saving and later loading the model state.

We have also included a pre-trained model in our code repository, saved to `../data/p1ch7/birds_vs_airplanes.pt`.

8.5 Model Design

We reached the point where we can build a feed-forward convolutional neural network and train it successfully to classify images. The natural question is, what now? What if we are presented with some more complicated problem? Admittedly our birds vs airplanes dataset wasn't that complicated: images were very small, the object under investigation is centered and takes most of the viewport.

If we moved to ImageNet, for instance, there we would find larger, more complex images, where the right answer would depend on multiple visual clues, often hierarchically organized - for instance looking for something looking like a screen to predict whether that dark brick is a remote control or a cell phone.

Plus images may not be our sole focus in the real world, where we have tabular data, sequences, text. The promise of neural networks is to be flexible enough to be able to solve problems on all these kinds of data given the proper architecture (i.e. the interconnection of layers, or modules) and the proper loss function.

PyTorch ships with a very comprehensive collection of modules and loss functions to implement state of the art architectures, ranging from feed-forward to LSTM and transformer networks (two very popular architectures for sequential data). Several models are available through the PyTorch Hub or as part of the `torchvision` and other vertical community efforts.

We'll see a few more advanced architectures in Part 2, where we'll walk through an end-to-end problem on analyzing CT scans, but in general it is beyond the scope of this book to explore variations on neural network architectures. However, what we can do now is build upon the knowledge we've accumulated thus far to understand how we can implement close to any architecture, thanks to the expressivity of PyTorch. The purpose of this section is precisely to provide those conceptual tools that will allow us to read the latest paper and start implementing it in PyTorch. Or, since authors often release PyTorch implementations of their papers, to read through the implementations without chocking on our coffee.

8.5.1 Width

Given our feed-forward architecture, there's a first couple of dimensions we'd likely want to explore before getting into further complications. The first dimension is the *width* of the network, that is, the number of neurons per layer, or channels per convolution. We can make a model wider very easily in PyTorch. We just specify a larger number of output channels in the first convolution, and increase the subsequent layers accordingly, taking care to change the forward function to reflect the fact that we'll now have a longer vector once we switch to fully-connected layers:

```
# In[35]:
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 16, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(8 * 8 * 16, 32)
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        out = F.max_pool2d(torch.tanh(self.conv1(x)), 2)
        out = F.max_pool2d(torch.tanh(self.conv2(out)), 2)
        out = out.view(-1, 8 * 8 * 16)
        out = torch.tanh(self.fc1(out))
        out = self.fc2(out)
        return out
```

If we want to avoid hard-coding numbers in the definition of the model, we can easily pass a parameter to *init* and parameterize width, taking care to also parameterize the call to *view* in the *forward* function.

```
# In[36]:
class Net(nn.Module):
    def __init__(self, n_chans1=32):
        super(Net, self).__init__()
        self.n_chans1 = n_chans1
        self.conv1 = nn.Conv2d(3, n_chans1, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(n_chans1, n_chans1 // 2, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(8 * 8 * n_chans1 // 2, 32)
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        out = F.max_pool2d(torch.tanh(self.conv1(x)), 2)
        out = F.max_pool2d(torch.tanh(self.conv2(out)), 2)
        out = out.view(-1, 8 * 8 * self.n_chans1 // 2)
        out = torch.tanh(self.fc1(out))
        out = self.fc2(out)
        return out

model = Net(n_chans1=32)
```

The numbers specifying channels and features for each layer are directly related to the number of parameters in a model, and, all other things being equal, they increase the *capacity* of the model. As we did previously, we can look at how many parameters our model has now:

```
# In[37]:
sum(p.numel() for p in model.parameters())
```

```
# Out[37]:  
38386
```

The greater the capacity, the more variability in the inputs the model will be able to manage, but at the same time the more likely overfitting will be, since the model can leverage a greater number of parameters to memorize unessential aspects of the input. We already went into ways to combat overfitting, the best being increasing the sample size, or, in the absence of new data, augmenting existing data through artificial modifications of the same data.

There are a few more tricks we can play at the model level (i.e. without acting on the data) in order to control overfitting, we'll review the most common ones.

REGULARIZATION

The first way to stabilize training is adding a regularization term to the loss. This term is crafted so that the weights of the model tend to be small, to be somehow bounded. In other words, it is a penalty on larger values of the weights. This makes loss have smoother topography and there's relatively less to gain from fitting individual samples.

The most popular regularization terms of this kind are L2 regularization, which is the sum of squares of all weights in the model, and L1 regularization, the sum of the absolute values of all weights in the model. Both of them are scaled by a (small) factor, which is a hyperparameter we set prior to training.

L2 regularization is also referred to as *weight decay*. The reason for this name is that, thinking about SGD and back-propagation, the negative gradient of the L2 regularization term with respect to a parameter `w_i` is $-2 * \text{lambda} * w_i$, where `lambda` is the aforementioned hyperparameter, simply named *weight decay* in PyTorch. So, adding L2 regularization to the loss function is equivalent to decreasing each weight by an amount proportional to its current value during the optimization step (hence the name "weight decay").

In PyTorch we could implement regularization pretty easily by adding a term to the loss. After computing the loss, whatever the loss function is, we can iterate the parameters of the model, sum their respective `sqr` (for L2) or `abs` (for L1) and back-propagate

```
# In[38]:  
def training_loop(n_epochs, optimizer, model, loss_fn, train_loader):  
    for epoch in range(1, n_epochs + 1):  
        loss_train = 0.0  
        for imgs, labels in train_loader:  
            outputs = model(imgs)  
            loss = loss_fn(outputs, labels)  
  
            l2_lambda = 0.9  
            l2_norm = sum(p.pow(2.0).sum() for p in model.parameters())  
            loss = loss + l2_lambda * l2_norm  
  
            optimizer.zero_grad()  
            loss.backward()
```

```
    optimizer.step()  
  
    loss_train += loss.item()
```

- ① replace `pow(2.0)` with `abs()` for L1 regularization

However, the `SGD` optimizer in PyTorch already has a `weight_decay` parameter, that corresponds to `2 * lambda`, that directly performs weight decay during the update as described previously. It is fully equivalent to adding the L2 norm of weights to the loss, without the need for accumulating terms in the loss and involving autograd.

DROPOUT

An effective strategy for combating overfitting was originally proposed in 2014 by Nitish Srivastava and co-authors from Geoff Hinton's group in Toronto, in a paper aptly entitled "Dropout: a simple way to prevent neural networks from overfitting". Sounds pretty much exactly what we're looking for uh? The idea behind dropout is indeed simple: zero out a random fraction of outputs from neurons across the network, where the randomization happens at each training iteration.

This procedure effectively generates slightly different models with different neuron topologies at each iteration, giving less chance to neurons in the model to coordinate in the memorization process happening during overfitting. An alternative point of view is that dropout perturbs the features being generated by the model, exerting an effect that is close to augmentation, but this time throughout the network.

In PyTorch we can implement dropout in a model by adding a `nn.Dropout` module between the non-linear activation function and the linear or convolutional module of the subsequent layer. As an argument, we need to specify the probability with which inputs will be zeroed out. In case of convolutions we'll use the specialized `nn.Dropout2D` or `nn.Dropout3D`, which zero out entire channels of the input.

```
# In[39]:  
class Net(nn.Module):  
    def __init__(self, n_chans1=32):  
        super(Net, self).__init__()  
        self.n_chans1 = n_chans1  
        self.conv1 = nn.Conv2d(3, n_chans1, kernel_size=3, padding=1)  
        self.conv1_dropout = nn.Dropout2D(p=0.4)  
        self.conv2 = nn.Conv2d(n_chans1, n_chans1 // 2, kernel_size=3, padding=1)  
        self.conv2_dropout = nn.Dropout2D(p=0.4)  
        self.fc1 = nn.Linear(8 * 8 * n_chans1 // 2, 32)  
        self.fc2 = nn.Linear(32, 2)  
  
    def forward(self, x):  
        out = F.max_pool2d(torch.tanh(self.conv1(x)), 2)  
        out = self.conv1_dropout(out)  
        out = F.max_pool2d(torch.tanh(self.conv2(out)), 2)  
        out = self.conv2_dropout(out)  
        out = out.view(-1, 8 * 8 * self.n_chans1 // 2)
```

```
out = torch.tanh(self.fc1(out))
out = self.fc2(out)
return out
```

Note that dropout is normally active during training, while during the evaluation of a trained model in production dropout is bypassed, or equivalently, it is assigned a probability equal to 0. This is controlled through the `train` property of the `Dropout` module. We recall that PyTorch lets us switch between the two modalities by calling

```
model.train()
```

or

```
model.eval()
```

on any `nn.Model` subclass. The call will be automatically replicated on the submodules, so that if `Dropout` is among them it will behave accordingly in subsequent forward and backward passes.

BATCH NORMALIZATION

Dropout was all the rage when, in 2015, another seminal paper was published by Sergey Ioffe and Christian Szegedy from Google, entitled "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". The paper described a technique that had multiple beneficial effects on training, namely allowing to increase the learning rate, make training less dependent on initialization and act as a regularizer, thus representing an alternative to Dropout.

The main idea behind batch normalization is to rescale the inputs to the activations of the network so that minibatches have a certain desirable distribution. Recalling the mechanics of learning and the role of non-linear activation functions, this helps avoiding that the inputs to activation functions are too far into the saturated portion of the function, thereby killing gradients and slowing training altogether.

In practical terms, batch normalization shifts and scales an intermediate input using the mean and standard deviation collected at that intermediate location over the samples of the minibatch. The regularization effect is a result of the fact that an individual sample and its downstream activations are always seen by the model shifted and scaled depending on the statistics across the randomly extracted minibatch. This is in itself a form of *principled* augmentation. The authors of the paper suggest that when using batch normalization eliminates or at least alleviates the need for dropout.

Batch normalization in PyTorch is provided through the `nn.BatchNorm1D`, `nn.BatchNorm2D` and `nn.BatchNorm3D` modules, according to the dimensionality of the input. Since the aim for batch normalization is to rescale the inputs of the activations, the natural location is after the linear transformation (convolution in this case) and the activation, as shown here:

```
# In[39]:
class Net(nn.Module):
    def __init__(self, n_chans1=32):
        super(Net, self).__init__()
        self.n_chans1 = n_chans1
        self.conv1 = nn.Conv2d(3, n_chans1, kernel_size=3, padding=1)
        self.conv1_dropout = nn.Dropout2D(p=0.4)
        self.conv2 = nn.Conv2d(n_chans1, n_chans1 // 2, kernel_size=3, padding=1)
        self.conv2_dropout = nn.Dropout2D(p=0.4)
        self.fc1 = nn.Linear(8 * 8 * n_chans1 // 2, 32)
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        out = F.max_pool2d(torch.tanh(self.conv1(x)), 2)
        out = self.conv1_dropout(out)
        out = F.max_pool2d(torch.tanh(self.conv2(out)), 2)
        out = self.conv2_dropout(out)
        out = out.view(-1, 8 * 8 * self.n_chans1 // 2)
        out = torch.tanh(self.fc1(out))
        out = self.fc2(out)
        return out
```

Just like for dropout, batch normalization needs to behave differently during training and inference. In fact, at inference time we want to avoid the output for a specific input to depend on the statistics of the other inputs we're presenting to the model. As such, we need a way to still normalize, but this time fixing the normalization parameters once and for all.

As minibatches are processed, in addition to estimating mean and standard deviation for the current minibatch, PyTorch also updates running estimates for mean and standard deviation that is representative of the whole dataset, as an approximation. This way, when the user specifies

```
model.eval()
```

and the model contains a batch normalization module, the running estimates are frozen and used for normalization.

8.5.2 Depth

Earlier we talked about width as the first dimension to act upon in order to make a model larger, in a way more capable. The second fundamental dimension is obviously *depth*. Since this is a deep learning book, depth is something we're supposedly into. After all, deeper models are always better than shallow ones, aren't they? Well, it depends. With depth, the complexity of the function the network is able to approximate generally increases. Talking about computer vision, a shallower network could identify a person's shape in a photo, while a deeper network could identify the person, the face on top half of it and the mouth within the face. Depth allows a model to deal with hierarchical information, where we need to understand context in order to say something about some input.

There's another way to think about depth: increasing depth is related to increasing the length of the sequence of operations that the network will be able perform when processing some input. This view, of a deep network that performs sequential operations to carry out a task, is likely

fascinating to a software developer, who is used to think about algorithms as sequences of operations, like find the persons's boundaries, look for the head on top of the boundaries, look for the mouth within the head.

SKIP CONNECTIONS

Depth comes with some additional challenges, which prevented deep learning models from exceeding the 10-something number of layers until late 2015. Adding depth to a model generally makes training harder to converge. Let's recall back-propagation and think about it in the context of a very deep network. The derivatives of the loss function with respect to the parameters, especially those in early layers, need to be multiplied by a lot of other numbers originating from the chain of derivative operations between the loss and the parameter. Those chains of numbers being multiplied could be small, generating ever smaller numbers, or large, swallowing smaller numbers due to floating point approximation. The bottomline is that a long chain of multiplications will tend to make the contribution of the parameter to the gradient *vanish*, making training of that layer ineffective, since that parameter and the others like it won't be properly updated.

In December 2015, Kaiming He and co-authors presented Residual Networks, an architecture that used a simple trick to allow very deep networks to be successfully trained. That work opened the door to networks ranging from tens to a hundred layers in depth, surpassing the then state of the art in computer vision benchmark problems. We have already encountered Residual Networks when we were playing with pre-trained models in Chapter 2. The trick we were mentioning is the following: use a so-called *skip connection* to short-circuit blocks of layers, as shown in Figure TODO.

A skip connection is nothing but an addition of the input to the output of a block of layers. This is exactly how it is done in PyTorch. Let's add one layer to our simple convolutional model, and let's use ReLU as the activation for a change.

```
# In[41]:
class Net(nn.Module):
    def __init__(self, n_chans1=32):
        super(Net, self).__init__()
        self.n_chans1 = n_chans1
        self.conv1 = nn.Conv2d(3, n_chans1, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(n_chans1, n_chans1 // 2, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(n_chans1 // 2, n_chans1 // 2, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(8 * 8 * n_chans1 // 2, 32)
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        out = F.max_pool2d(torch.relu(self.conv1(x)), 2)
        out = F.max_pool2d(torch.relu(self.conv2(out)), 2)
        out = F.max_pool2d(torch.relu(self.conv3(out)), 2)
        out = out.view(-1, 8 * 8 * self.n_chans1 // 2)
        out = torch.tanh(self.fc1(out))
        out = self.fc2(out)
        return out
```

Adding a skip connection *a-la* ResNet to this model amounts to adding the output of the first layer in the `forward` function to the input of the third layer.

```
# In[42]:
class Net(nn.Module):
    def __init__(self, n_chans1=32):
        super(Net, self).__init__()
        self.n_chans1 = n_chans1
        self.conv1 = nn.Conv2d(3, n_chans1, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(n_chans1, n_chans1 // 2, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(n_chans1 // 2, n_chans1 // 2, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(8 * 8 * n_chans1 // 2, 32)
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        out = F.max_pool2d(torch.relu(self.conv1(x)), 2)
        out1 = out
        out = F.max_pool2d(torch.relu(self.conv2(out)), 2)
        out = F.max_pool2d(torch.relu(self.conv3(out + out1)), 2)
        out = out.view(-1, 8 * 8 * self.n_chans1 // 2)
        out = torch.tanh(self.fc1(out))
        out = self.fc2(out)
        return out
```

In other words, we're using the output of the first activations as inputs to the last, in addition to the standard feed-forward path. This is also referred to as *identity mapping*. So, how is this alleviating the issues with vanishing gradients we were mentioning earlier?

Thinking back about back-propagation, we can appreciate that a skip connection, or a sequence of skip connections in a deep network, creates a direct path from the deeper parameters to the loss. This makes their contribution to the gradient of the loss more direct, as partial derivatives of the loss with respect to those parameters have a change not to be multiplied by a long chain of other operations.

It has been observed that skip connections have a beneficial effect on convergence especially in the initial phases of training. Also, the loss landscape of deep Residual Networks is a lot smoother than feed-forward networks of the same depth and width.

It is worth noting that skip connections were not new to the world when Residual Networks came along. Highway networks, U-net, all made use of skip connections one form or the other. However, the way Residual Networks used skip connections enabled models of very high depths, exceeding 100, to be amenable to training.

Since the advent of ResNets, other architectures have taken skip connections to the next level. One in particular, DenseNet, proposed to connect each layer with several other layers downstream through skip connections, achieving state of the art results with fewer parameters. By now we know how to implement something like DenseNets: just arithmetically add earlier intermediate outputs to downstream intermediate outputs.

8.5.3 Building very deep models in PyTorch

We talked about exceeding 100 layers in a convolutional neural network. How can we build that network in PyTorch without losing our minds in the process? The standard strategy is to define a building block, such as a (Conv2D, ReLU, Conv2D) + skip connection block, and then build the network dynamically in a `for` loop. Let's see it done in practice. We first create a module subclass whose sole job is to provide the computation for one *block*, that is, one group of convolutions, activation and skip connection.

```
# In[43]:
class ResBlock(nn.Module):
    def __init__(self, n_chans):
        super(ResBlock, self).__init__()
        self.conv = nn.Conv2d(n_chans, n_chans, kernel_size=3, padding=1)
        self.batch_norm = nn.BatchNorm2d(num_features=n_chans)

    def forward(self, x):
        out = self.conv(x)
        out = self.batch_norm(out)
        out = torch.relu(out)
        return out + x
```

Since we're planning to generate a deep model, we are includinig batch normalization in the block, since this will help avoiding that gradients vanish during training. we'd now like to generate a 100-block network, does this mean we'll have to get our left hand ready for some serious cutting and pasting? Not at all, we already have all the ingredients for imagining how this could look like.

First, in `init`, we create a `nn.Sequential` containing a list of `ResBlock` instances. `nn.Sequential` will ensure that the output of one block goes as input to the next. It will also ensure that all the parameters in the block are visible to `Net`. Then, in `forward`, we just call the sequential to traverse the 100 blocks and generate the output.

```
# In[44]:
class Net(nn.Module):
    def __init__(self, n_chans1=32, n_blocks=10):
        super(Net, self).__init__()
        self.n_chans1 = n_chans1
        self.conv1 = nn.Conv2d(3, n_chans1, kernel_size=3, padding=1)
        self.resblocks = nn.Sequential(* [ResBlock(n_chans=n_chans1)] * n_blocks)
        self.fc1 = nn.Linear(8 * 8 * n_chans1, 32)
        self.fc2 = nn.Linear(32, 2)

    def forward(self, x):
        out = F.max_pool2d(torch.relu(self.conv1(x)), 2)
        out = self.resblocks(out)
        out = F.max_pool2d(out, 2)
        out = out.view(-1, 8 * 8 * self.n_chans1)
        out = torch.tanh(self.fc1(out))
        out = self.fc2(out)
        return out

model = Net(n_chans1=32, n_blocks=100)
```

In the implementation we parameterized the actual number of layers, which is important for

experimentation and reuse. Also, needless to say, backprop will work as expected.

All the above shouldn't encourage us to seek depth on a dataset of 32x32 images, but it clearly demonstrates how this can be achieved on more challenging datasets, like ImageNet. The above also provides the key elements for understanding existing implementations for models like ResNet, for instance in `torchvision`.

8.5.4 Now it's already outdated

The curse and blessing of a deep learning practitioner is that neural network architectures evolve at a very rapid pace. This is not to say that what we've seen in this chapter is necessarily old school, but a thorough illustration of the latest and greatest architectures is matter for another book (and they would cease to be the latest and the greatest pretty quickly anyway). The take home message here is that we should make every effort to proficiently translate the math behind a paper into actual PyTorch code, or at least to understand code that others have written with the same intention. In the last few chapters we hopefully gathered quite a few of the fundamental skills to go in and translate ideas into implemented models in PyTorch.

8.6 Conclusion

Right, so, after quite some we now have a model that our fictional friend Jane can use to filter images for her blog. All we have to do is take an image coming in, crop-and-resize it to 32x32 and see what the model has to say about it. Admittedly we have solved only a part of the problem, but it was quite a journey in itself.

We have solved just a part of it because there are a few interesting unknowns we still would have to face. One is picking out the bird or the airplane from a larger image. Creating bounding boxes around objects in an image is something that a model such as the one we created is not capable of doing.

Another hurdle concerns what happens when Fred the cat walks in front of the camera. Our model will not refrain from giving its opinion about how bird-like the cat is! It will happily output airplane or bird, perhaps with 0.99 probability. This problem, that of being very confident about samples that are far from the training distribution, is called *overgeneralization*. It's one of the main problems when one takes a (presumably good) model to production, in those cases where we can't really trust the input (which, sadly, is the majority of real-world cases).

In this chapter we have built reasonable, working models that can learn from images in PyTorch. We did it in a way that helped us building an intuition around convolutional networks. We have also explored ways in which we can make our models wider and deeper, while controlling effects like overfitting. While we still only scratched the surface, we have taken another significant step ahead from the previous chapter. The steps we have taken provide us with a solid basis for facing the challenges we'll encounter while working on deep learning projects.

Now that we've gotten familiar with both PyTorch conventions and common features, we're ready to tackle something bigger. We're going to transition from a mode where each chapter or two presents a small problem into spending multiple chapters breaking down a bigger, real-world problem. Part 2 is going to use automatic detection of lung cancer as the motivating example, and we will go from being familiar with the PyTorch API to being able to implement entire projects using PyTorch. We'll start with the next chapter explaining the problem from a high level, and then getting into the details of the data we'll be using.

8.7 Exercises

- Change the model to use a 5x5 kernel with `kernel_size=5` passed to the `nn.Conv2d` constructor.
 - What impact does this have on the number of parameters in the model?
 - Does this change improve or degrade over-fitting?
 - Read pytorch.org/docs/stable/nn.html#conv2d
 - Can you describe what `kernel_size=(1, 3)` will do?
 - How does the model behave with such a kernel?
- Can you find an image with neither a bird nor airplane in it, but that the model will claim has one or the other with more than 95% confidence?
 - Can you manually edit a neutral image to make it more airplane-like?
 - Can you manually edit an airplane image to trick the model into reporting a bird?
 - Do these tasks get easier with a network with less capacity? More capacity?

8.8 Summary

- Convolution can be used as the linear operation of a feed forward network dealing with images. Using convolution produces networks with fewer parameters, exploiting locality and featuring translation invariance.
- Stacking multiple convolutions with their activations one after the other, and using max pooling in between has the effect of applying convolutions to increasingly smaller feature images, thereby effectively accounting for spatial relationships across larger portions of the input image as depth increases.
- Any `nn.Module` subclass can recursively collect its and its children's parameters and return them. This can be used to count them, feed them into the optimizer, or inspect their values.
- The functional API provides modules that do not depend on storing internal state. It is used for operations that do not hold parameters, hence they are not trained.
- Once trained, parameters of a model can be saved to disk and loaded back in one line of code each.

Learning from Images in the Real-World: Early Detection of Lung Cancer



Part 2 is going to be structured differently than part 1; almost a book within a book. We're going to be taking a single use case and exploring it in depth over the course of several chapters, starting with the basic building blocks we've learned from part 1, and building out a more complete project than we've seen so far. Our first attempts are going to be incomplete and inaccurate, and we'll explore how to diagnose those problems, and then fix them. We're going to identify various other improvements to our solution, implement them, and measure their impact.

Our goal is to give you the tools to deal with situations where things aren't working, which is a far more common state of affairs than part 1 might have led you to believe. We can't predict every failure case, or cover every debugging technique, but hopefully we'll have given you enough to not feel stuck when you encounter some new roadblock. Similarly, we want to help you avoid situations with your own projects where you have no idea what you could do next when your projects are underperforming. Instead we hope your ideas list will be so long that the challenge will be to prioritize!

In order to be able to present these ideas and techniques, we'll need a context with some nuance and a fair bit of heft to it. We've chosen automatic detection of malignant tumors in the lungs using only a CT scan of a patient's chest as input. Detecting lung cancer early has a huge impact on survival rate, but is difficult to do manually, especially in any comprehensive, whole-population sense. Currently, the work of reviewing the data must be performed by highly-trained specialists, requires painstaking attention to detail, and is dominated by cases where no cancer exists.

Doing that job well is akin to being placed in front of a hundred haystacks and being told "determine which of these, if any, have a needle in them." The human brain just isn't built well for that kind of monotonous work.

And that, of course, is where deep learning comes in.

9

Using PyTorch To Fight Cancer

This chapter covers:

- How to break down a large problem like cancer detection into a series of smaller, easier ones.
- Exploring the constraints of an intricate deep learning problem, and deciding on a structure and approach.
- Crucial background information about cancer detection for the example project.
- Special issues when working with three dimensional image data files.

As you might have guessed, the title of this chapter is more eye-catching implied-hyperbole than anything approaching a serious statement of intent. Let us be precise: our project in part 2 will be to take three-dimensional CT scans of human torsos as input, and produce as output the location of any suspected malignant tumors, if such exist. Prior to the arrival of deep learning, this problem was exclusively tackled by humans. Radiologists had to pore over every single image manually, looking for hints that some part of the patient's body might be displaying signs of malignancy. Doing the search this way results in a lot of missed warning signs, particularly in the early stages when the hints are more subtle.

Automating this process is going to give us experience working in uncooperative environment, where we have to do more work from scratch, and there are fewer easy answers to problems that we might run into. Together, we'll get there, though! Once you're done reading part 2, we think you'll be ready to start working on a real-world, unsolved problem of your own choosing.

We chose this problem of lung tumor detection for a few reasons. The primary reason is that the problem itself is unsolved! This is important, because we want to make it clear that you can use PyTorch to tackle cutting edge projects quickly and easily. We're hoping that increases the confidence that you have in PyTorch as a framework, as well as in yourself as a developer. Another nice aspect of this problem space is that while it's unsolved, there have been a lot of

teams paying attention to it recently that have seen promising results. That means that this task is probably right at the edge of our collective ability to solve; we won't be wasting our time on a problem that's actually decades away from reasonable solutions. That attention on the problem has also resulted in a lot of high-quality papers and open source projects which are a great source of inspiration and ideas. This will be a huge help once we conclude part 2 of the book, if you are interested in continuing to improve on the solution we create. We'll provide some links to additional information in [chapter 14](#).

Part 2 will remain focused on the problem of lung cancer detection, but skills we'll teach will be general. Learning how to investigate, pre-process, and present your data for training is important no matter what project you're working on. While we'll be covering pre-processing in the specific context of lung cancer, the general idea is that *this is what you should be prepared to do* to succeed. Similarly, setting up your training loop, getting the right performance metrics, and tying the project's models together into a final application are all general skills that we'll employ as we go through the chapters of part 2.

NOTE

While the end result of part 2 will clearly be working, the output will not be accurate enough to be usable clinically. We're focusing on using this as a motivating example for *teaching PyTorch*, not on employing every last trick to solve the problem.

We have two main goals for this chapter. We'll start with covering the overall plan for part 2, so that we have a solid idea of the larger scope that the following individual chapters will be building towards. Next, in chapter 10, 10, we will start to build out the data parsing and manipulation routines that will produce data to be consumed in chapter 11, while training our first model. In order to do that well, we'll use this chapter to cover some of the context that our part 2 project will be operating in; we'll go over data formats, data sources, and exploring the constraints that our problem domain places on us. Get used to doing these tasks, since you'll have to do them for any serious deep learning project!

This project will build off of the foundational skills learned in part 1. In particular, the content covering model construction from chapter 8, 8 will be directly relevant. Repeated convolutional layers followed by a resolution-reducing downsampling layer are going to still comprise the majority of our model. We will be using three-dimensional data as input to our model, however. This is conceptually similar to the 2D image data used in the last few chapters of part 1, but we will not be able to rely on all of the 2D-specific tools available in the PyTorch ecosystem.

The main differences between the work we did with convolutional models in chapter 8 and what we'll do in part 2 are related to how much effort we put into things outside the model itself. In chapter 8, we used a provided, off-the-shelf data set, and did little data manipulation before feeding the data into a model for classification. Almost all of our time and attention was spent building the model itself, while now we're not even going to get to building the first of our two

models until chapter 11. That is a direct consequence of having non-standard data without pre-built libraries ready to hand us training samples suitable to plug into a model. We'll have to learn about our data and implement quite a bit ourselves.

Even once that's done, this will not end up being a case where we convert the CT to a tensor, feed it into a neural network, and have the answer pop out the other side. As is common for real-world use cases such as this, a workable approach is going to have to be more complicated to account for confounding factors such as limited data availability, finite computational resources, and limitations on our ability to design effective models. Please keep that in mind as we build to a high-level explanation of our project architecture.

Speaking of finite computational resources, part 2 will require access to a GPU to achieve reasonable training speeds. Trying to train the models we will build in part 2 on CPU could take weeks!⁷⁹ For readers without a GPU handy, we provide pre-trained models in chapter 14; the diagnosis script there can probably be run overnight. While we don't want to tie the book to proprietary services if we don't have to, we should note that at the time of writing, Colaboratory⁸⁰ provides free GPU instances that might be of use. PyTorch even comes pre-installed!

NOTE

Many of the code examples presented in Part 2 will have complicating details omitted. Rather than clutter the examples with logging, error handling, and edge cases, the text of this book will contain only code that expresses the core idea under discussion. Full working code samples can be found on GitHub.⁸¹

Okay, we've established that this is a hard, multifaceted problem, but what are we going to do about it? Instead of looking at the entire CT for signs of malignancy, we're going to solve a series of simpler problems that will combine to provide the end-to-end result that we're interested in. Like a factory assembly line, each step will take raw materials (data) and/or output from previous steps, perform some processing, and hand off the result to the next station down the line. Not every problem needs to be solved this way, but breaking off chunks of the problem to solve in isolation is often a great way to start. Even if it turns out to have been the wrong approach for a given project, it's likely you'll have learned enough while working on the individual chunks that you'll have a good idea of how to restructure your approach into something successful.

Before we get into the details of how we're going to break down our problem, we need to learn some details about the medical domain. While the code listings will tell you *what* we're doing, learning about radiation oncology will inform *why*. Learning about your problem space is crucial, no matter what domain it is. While deep learning is powerful, it's not magic, and trying to apply it blindly to non-trivial problems will likely fail. Instead, we'll have to combine insights into the space with intuition about neural network behavior. From there, disciplined

experimentation and refinement should give us enough information to close in on a workable solution.

9.1 What is a CT scan, exactly?

Before we get too far into the project, we need to take a moment to explain exactly what a CT scan is. We will be using data from CT scans extensively as the main data format for our project, and so having a working understanding of the data format's strengths, weaknesses, and fundamental nature will be crucial to utilizing them well. The key point is this: CT scans are essentially three-dimensional x-rays, represented as a three-dimensional array of single-channel data. As we might recall from chapter 4, 4.2, this is like a stacked set of grayscale PNG images.

SIDE BAR

Voxel

A *voxel* is the three-dimensional equivalent to the familiar two-dimensional pixel. It encloses a volume of space (hence, "volumetric pixel"), rather than an area, and will typically be arranged in a three-dimensional grid to represent a field of data. Each of those dimensions will have a measurable distance associated. Often, voxels are cubic, but for this chapter we will be dealing with voxels that are rectangular prisms.

In addition to medical data, we can see similar voxel data in fluid simulations, 3D scene reconstruction from 2D images, LIDAR data for self-driving cars, and many other problem spaces. Those spaces are going to have their individual quirks and subtleties, and while the APIs that we're going to cover here apply generally, we must also be aware of the nature of the data we're using with those APIs if we want to be effective.

Each voxel of a CT scan has a numeric value that roughly corresponds to the average mass density of the matter contained inside. Most visualizations of that data have high-density material like bones and metal implants as white, low-density air and lung tissue as black, and fat and tissue as various shades of gray in between. Again, this ends up looking somewhat similar to an x-ray, with some key differences.

The primary difference between CT scans and x-rays is that while an x-ray is a projection of 3D intensity (in this case, tissue and bone density) onto a 2D plane, a CT scan retains the 3rd dimension of the data. This allows us to render the data in a variety of ways, for example as a grayscale solid, which we can see in 9.1.

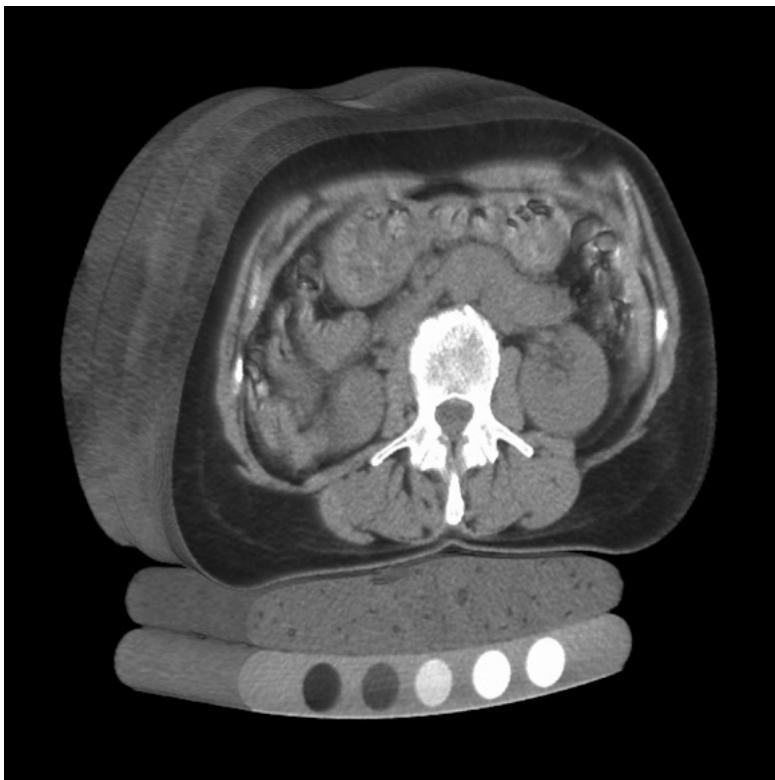


Figure 9.1 A CT scan of a human torso showing, from the top, skin, organs, spine, and patient support bed. (Image

attributioncommons.wikimedia.org/wiki/File:Image_of_3D_volumetric_QCT_scan.jpg:
MindwaysCT Software / CC BY-SA 3.0 creativecommons.org/licenses/by-sa/3.0/deed.en)

NOTE

CT scans actually measure radiodensity, which is a function of both mass density and atomic number of the material under examination. For the purposes here, the distinction isn't relevant, since the model will consume and learn from the CT data no matter what the exact units of the input happen to be.

This 3D representation also allows us to "see inside" the subject by hiding tissue types we are not interested in. For example, we can render the data in 3D and restrict visibility to only bone and lung tissue, as we see in 9.2.

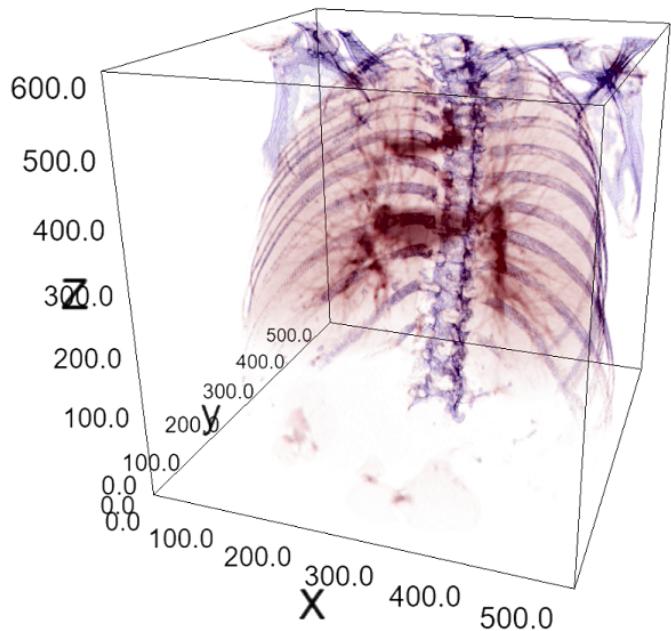


Figure 9.2 A CT scan showing ribs, spine, and lung structures.

CT scans are much more difficult to acquire than x-rays, in that doing so requires a machine like the one in 9.3 that typically runs upwards of a million dollars new, as well as needing trained staff to operate. Most hospitals and some well-equipped clinics have a CT scanner, but they aren't nearly as ubiquitous as x-ray machines. This, combined with patient privacy regulations, can make it somewhat difficult to get CT scans unless someone has already done the work of gathering and organizing a collection of them.

9.3 also shows an example bounding box for the area contained in the CT scan. The bed the patient is resting on will move back and forth, allowing the scanner to image multiple slices of the patient, and hence filling the bounding box. The scanner's darker, central ring is where the actual imaging equipment is located.



Figure 9.3 A patient inside of a CT scanner, with the CT scan's bounding box overlaid. Outside of stock photos, patients don't typically wear street clothes while in the machine.

A final difference is that the data is a digital-only format. "CT" stands for Computed Tomography⁸⁴. The raw output of the scanning process doesn't look particularly meaningful to the human eye, and requires a computer to properly reinterpret into something we can understand. The settings of the CT scanner when the scan is taken can have a large impact on the resulting data.

While this information might not seem particularly relevant, we have actually learned something that is! From 9.3 we can see that the way the CT scanner measures distance along the head-to-foot axis is different than the other two axes. The patient actually moves along that axis! This explains (or at least is a strong hint as to) why our voxels might not be cubic, and also ties into how we approach massaging our data in chapter 12, 12.5. This is a good example of how we need to understand our problem space, if we're going to make effective choices about how to solve our problem. When starting to work on your own projects, make sure that you do the same investigation into the details of your data.

9.2 The project: an end-to-end malignancy detector for lung cancer

Most of the bytes on disk are going to be devoted to storing the CT scans' 3D arrays containing density information, and our models are going to be primarily consuming various subslices of those 3D arrays. Now that we've got our heads wrapped around the basics of CT scans, let's discuss the structure of our project. We're going to use five main steps to go from examining a whole-chest CT scan to giving the patient a lung cancer diagnosis.

Our full, end-to-end solution shown in 9.4 will be to load CT data files to produce a `ct` instance that contains the full three-dimensional scan, combine that with a module that does "segmentation" (flagging voxels of interest), and then cluster the interesting voxels into potential tumors called "nodules." The nodule locations are combined back with the CT voxel data to produce nodule samples, which can then be classified by our classification model to determine if they're malignant or not. Each of those individual, per-nodule classifications can then be combined into a whole-patient diagnosis.

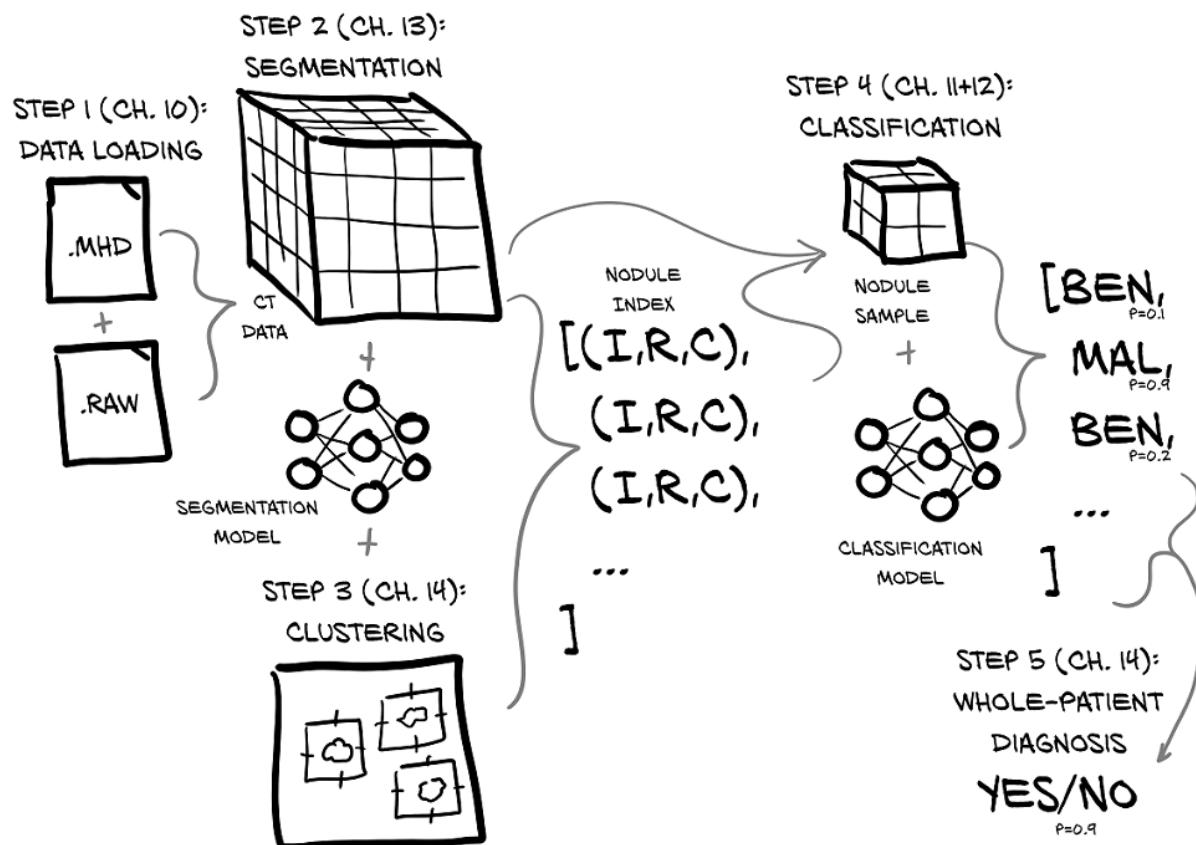


Figure 9.4 The end-to-end process of taking a full-chest CT scan and determining if the patient has a malignant tumor.

In more detail, we will:

1. Load our raw CT scan data into a form that we can use with PyTorch. [Putting raw data

into a form usable by PyTorch would be the first step in any project you face. The process is somewhat less complicated with 2D image data, and simpler still with non-image data.]

2. Identify the voxels of potentially malignant tumors in the lungs using PyTorch to implement a technique known as "segmentation." This is roughly akin to producing a heatmap of areas that should be fed into our classifier in step 3.
 - This will allow us to focus on potential malignant tumors inside the lungs, and ignore huge swathes of uninteresting anatomy (can't have lung cancer in the stomach, for example).
 - Generally, being able to focus on a single, small task is going to be best while learning. With experience, there are some situations where more complicated model structures can yield superlative results (for example, the GAN game we saw in chapter 2), but designing those from scratch requires an extensive mastery of the basic building blocks first. Gotta walk before you run, and all that.
3. Cluster interesting voxels into lumps called nodules (see 9.2.2 for more information on nodules). Here, we will be finding the rough center of each hotspot on our heatmap.
 - Each nodule can be located with by the index, row, and column of their center point. We do this to present a simple, constrained problem to the final classifier.
 - This will not involve PyTorch directly, which is why we've pulled this out into a separate step.
 - Often when working with multi-step solutions, there will be non-deep-learning glue steps between the larger, deep-learning-powered portions of the project.
4. Classify identified nodules as benign or malignant using 3D convolution.
 - This will be similar in concept to the 2D convolution we've covered previously in chapter 8.
 - The features that determine malignancy of a tumor are local to the tumor in question, so this approach should provide a good balance between limiting input data size without excluding relevant information.
 - Making scope-limiting decisions like this can help keep each individual task constrained, which can help by limiting the amount of things to examine when troubleshooting.
5. Diagnose the patient using the combined per-nodule classifications.
 - We will be taking a simple maximum of the per-tumor malignancy predictions, as a patient only needs one tumor to be malignant to have cancer.
 - Other projects might want to use different ways of aggregating the per-instance predictions into a file score.

//FL TODO 9/25/19: Can you expand on that last point a bit? What are some otherways of aggregating the per-instance predictions?

We are **standing on the shoulders of giants** when deciding on this five-step approach. We'll discuss these giants and their work more in chapter 14. There isn't any particular reason why we should know in advance that this project structure will work well for this problem. Instead, we're relying on others who have actually implemented similar things and reported success when doing so. Expect to have to experiment to find workable approaches when transitioning to a different domain, but always try and learn from earlier efforts in the space, and from those who have worked in similar areas that have discovered things that might transfer over well. Go out there,

look for what others have done and use that as a benchmark. At the same time, avoid getting code and running it *blindly*, because you need to fully understand the code you're running in order to use the results to make progress for yourself.

9.4 is only depicting the final path through the system once we've built and trained all of the requisite models. The actual work required to train the relevant models will be detailed as we get closer to actually implementing each step.

The data we'll be using for training provides human-annotated output for both step 1 and step 3. This allows us to treat steps 1 (identifying voxels) as almost a separate project from step 3 (classification). We have human experts who have annotated the data with the correct answers, and so we can work on either one in the order that we prefer. We will be focusing on step 3 first, since it requires an approach similar to what we used in chapter 8, using multiple convolutional and pooling layers to aggregate spatial information before feeding it into a linear classifier. Once we've got a handle on our classification model, then we can start working on segmentation. Since segmentation is the more complicated topic, we want to tackle it without having to learn both segmentation and the fundamentals of CT scans and malignant tumors at the same time. Instead, we'll explore the cancer detection space while working on a more familiar classification problem.

This approach of starting in the middle of the problem and working our way out probably seems odd. Starting at step 1 and working our way forward would make more intuitive sense. Being able to carve up the problem and work on steps independently is useful, however, since it can encourage more modular solutions, as well as being easier to partition the workload between members of a small team. Additionally, actual clinical users would likely prefer a system that flags suspicious nodules for review, rather than provide a single binary diagnosis. Adapting our modular solution to different use cases will probably be easier than if we'd done a monolithic, from-the-top system.

As we work our way through implementing each step, we're going to be going into a fair bit of detail about lung tumors, as well as going over a lot of fine-grained detail about CT scans. While that might seem off-topic for a book that's focused on PyTorch, we're doing so specifically so that you begin to develop an intuition about the problem space. That's crucial to have, as the space of all possible solutions and approaches is too large to effectively code, train, and evaluate.

If we were working on a different project (say, the one you tackle after finishing this book), we'd still need to do investigation to understand the data and problem space. Perhaps you're interested in satellite mapping, and your next project needs to consume pictures of our planet taken from orbit. You'd need to ask questions about the wavelengths being collected — do you get only normal RGB, or something more exotic? What about infrared or ultraviolet? In addition, there might be impacts on the images based on time of day, or if the imaged location isn't directly under the satellite, skewing the image. Will the image need correction?

Even if your hypothetical *third* project's data type remains the same, it's probable that the domain you'll be working in will change things, possibly drastically. Processing camera output for self-driving cars still involves 2D images, but the complications and caveats are wildly different. For example, it's much less likely that a mapping satellite will need to worry about the sun shining into the camera, or getting mud on the lens!

We must be able to use our intuition to guide our investigation into potential optimizations and improvements. That's true of deep learning projects in general, and so we'll practice using our intuition as we go through part 2. So, let's do that. Take a quick step back, and do a gut-check. What does your intuition say about this approach? Does it seem over-complicated to you?

9.2.1 Why can't we just throw data at a neural network until it works?

After reading the last section, I couldn't blame you for thinking "This is nothing like chapter 8!" You might be wondering why we've got two separate model architectures, or why the overall data flow is so complicated. Well, our approach is different from what we saw in chapter 8 for a reason. It's a hard task to automate, and people haven't fully figured it out yet. That difficulty translates to complexity; once we as a society have solved this problem definitively, there will probably be an off-the-shelf library package we can grab to have it Just Work(tm), but we're not there just yet.

Why so difficult, though?

Well, for starters, the majority of a CT scan is fundamentally uninteresting with regards to answering the question "does this patient have a malignant tumor?" This makes intuitive sense, since the vast majority of the patient's body will consist of healthy cells. In the cases where there is a malignant tumor, up to 99.9999% of the voxels in the CT still won't be cancer. That ratio is equivalent to a two-pixel blob of incorrectly tinted color somewhere on a high-definition television, or a single misspelled word out of a large trilogy of novels.

Can you identify the white dot in the three views of 9.5 that has been flagged as malignant? ⁸⁵

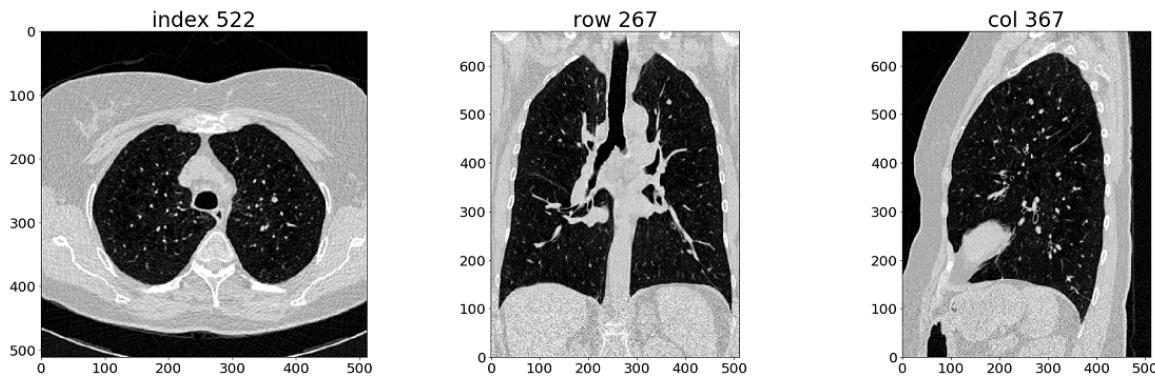


Figure 9.5 A CT scan with approximately one thousand potential tumors. Exactly one has been identified as malignant, when reviewed by a human specialist. The rest are normal anatomical structures, like blood vessels, lesions, and other non-problematic lumps.

If you need a hint, the index, row, and column values can be used to help find the relevant blob of dense tissue. Do you think you could figure out the relevant properties of malignant tumors given only images (and that means *only* the images — no index, row, and column information!) like the above? What if you were given the entire 3D scan, not just three slices that intersect the interesting part of the scan?

NOTE Don't fret if you can't locate the tumor! We're trying to illustrate just how subtle this data can be; it being hard to identify visually is the entire point of this example.

While it's certainly theoretically possible to just throw an arbitrarily large amount of data at a neural network until it learns the specifics of the proverbial lost needle, as well as how to ignore the hay, it's going to be practically prohibitive to collect enough data and wait for a long enough time to get the network trained properly. That's not going to be the *best* approach since the results are poor, and most readers won't have access to the compute resources to pull it off at all.

To come up with the best solution, we could investigate proven model designs^{86 87} that can better integrate data in an end-to-end manner. These complicated designs are capable of producing high-quality results, but they're not the *best* because understanding the design decisions behind them require having mastered fundamental concepts first. That makes these advanced models poor candidates to use while teaching those same fundamentals!

That's not to say that our multi-step design is the *best* approach, either, but that's because "best" is only relative to the criteria we chose to evaluate approaches. There are *many* "best" approaches, just as there are many goals we could have in mind as we work on a project. Our self-contained, multi-step approach has some disadvantages as well.

Recall the GAN game from chapter 2, 2.2. There, we had two networks cooperating to produce convincing forgeries of old master artists. The artist would produce a candidate work, and the

scholar would critique it, giving the artist feedback on how to improve. Put in technical terms, the structure of the model allowed gradients to back-propagate from the final classifier (fake or real) to the earliest parts of the project (the artist).

Our approach for solving the problem won't be using end-to-end gradient back-propagation to directly optimize for our end goal. Instead, we'll be optimizing discrete chunks of the problem individually, since our segmentation model and classification model won't be trained in tandem with each other. That might limit the top-end effectiveness of our solution, but we feel that this will make for a much better learning experience.

We feel that being able to focus on a single step at a time allows us to zoom in and concentrate on the smaller number of new skills we're learning. Each of our two models will be focused on performing exactly one task. Similar to the human radiologist as they review slice after slice of CT, the job gets much easier to train for if scope is well-contained. We also want to provide tools that allow for rich manipulation of the data. Being able to zoom in and focus on the detail of a particular location will have a huge impact on overall productivity while training the model compared to having to look at only the entire image at once.

Our segmentation model is forced to consume the entire image, but we will structure things so that our classification model gets a zoomed-in view of the areas of interest.

Step 3 (clustering) will produce and step 4 (classification) will consume data similar to the image in 9.6 containing sequential transverse slices of a tumor:

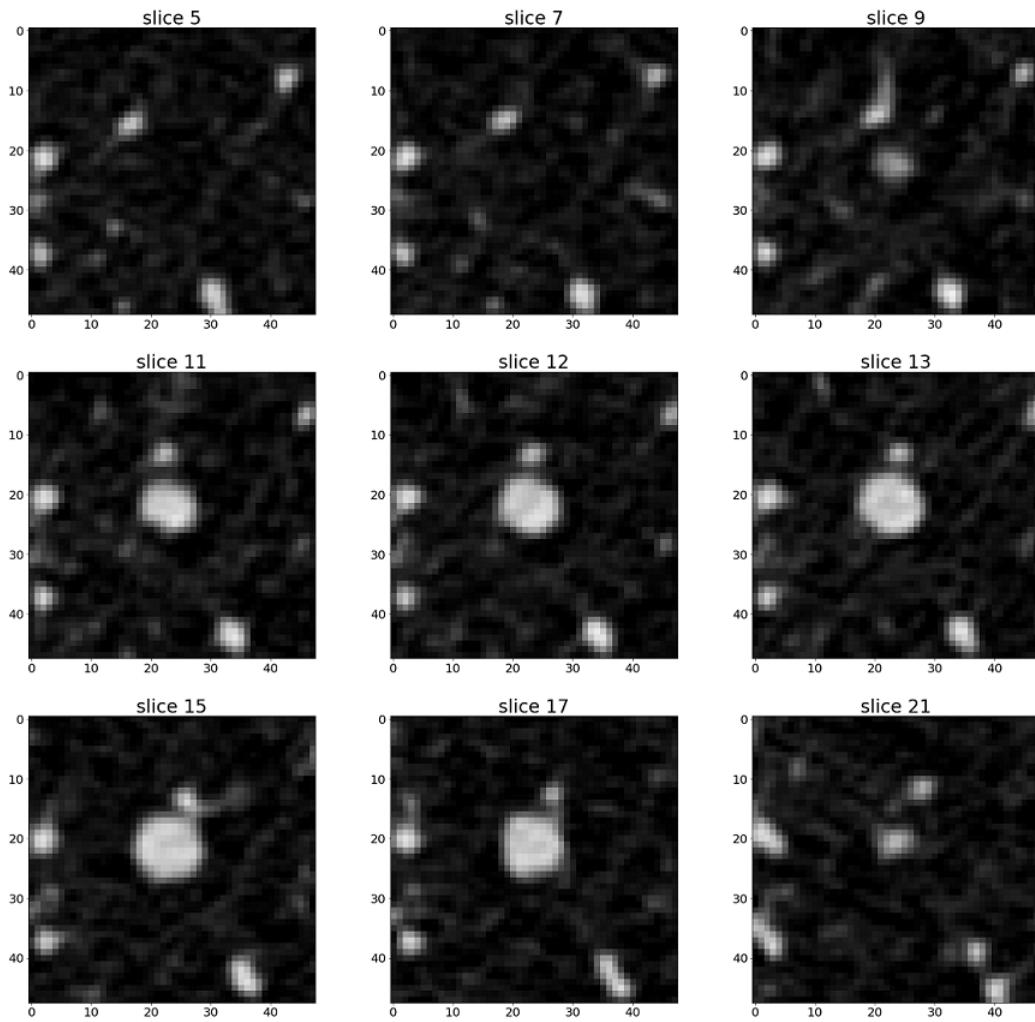


Figure 9.6 A close-up, multi-slice crop of the malignant tumor from the CT scan in .

This image is a close-up view of a potential malignant tumor, and is what we're going to be training the step 4 model to classify as either benign or malignant. While this lump may seem nondescript to an untrained eye (or untrained convolutional network), identifying the warning signs of malignancy in this sample is at least a far more constrained problem than having to consume the entire CT we saw above. Let's repeat our high-level overview in 9.7. Chapters 11 and 12 will focus on solving the problem of classifying these nodules. After that, we'll back up to work on step 2 (using segmentation to find the candidate tumors) in chapter 13, then close out the book by implementing the end-to-end project with step 3 (clustering) and step 5 (diagnosis).

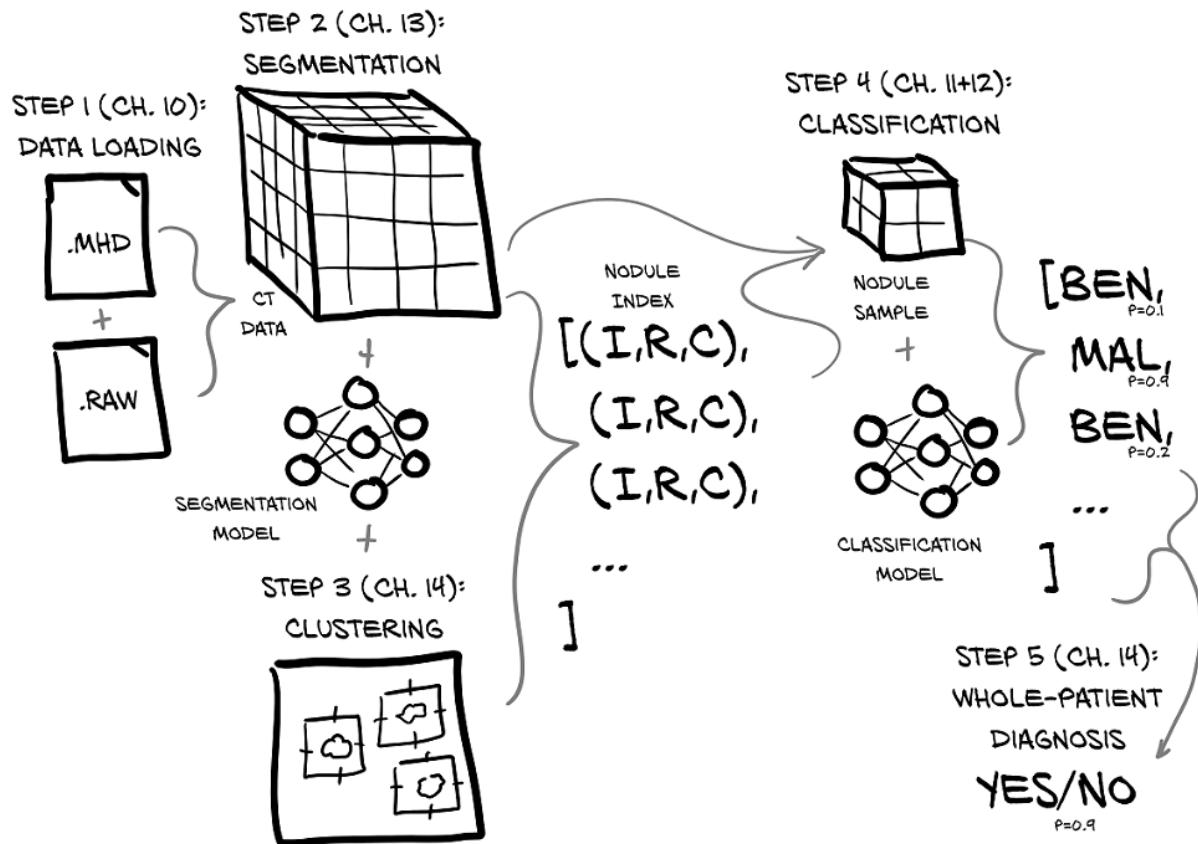


Figure 9.7 The end-to-end process of taking a full-chest CT scan and determining if the patient has a malignant tumor.

Our code for the next chapter will provide routines to produce zoomed-in nodule images like 9.6 yourself. We will be doing data loading work of step 1 in chapter 10; we'll start the classification work in chapter 11.

NOTE

Standard rendering of CTs will place the superior at the top of the image (basically, the head goes up), but CTs order their slices such that the first slice is the inferior (towards the feet), and so matplotlib will render the images upside down unless we take care to flip them. Since that flip doesn't really matter to our model, we won't complicate the code paths between our raw data and our model, but we will add a flip to our rendering code to get the images right-side-up. For more information about the CT coordinate systems, see 10.3.

9.2.2 What is a nodule?

As we've said, in order to understand our data well enough to use it effectively, we are going to need to learn some specifics about cancer and radiation oncology. One last key thing we need to understand is what a *nodule* is. Simply put, a nodule is any of the myriad lumps and bumps that might appear inside of someone's lungs. Some are problematic from a health-of-the-patient perspective, some are not. The precise definition⁸⁸ limits the size of a nodule to 3 cm or less, with a larger lump being a "lung mass," but we're going to use "nodule" interchangeably for all such anatomical structures, since it's a somewhat arbitrary cutoff, and we're going to deal with lumps on both sides of 3 cm using the same code paths. Nodules have a wide variety of causes — infection, inflammation, blood supply issues, malformed blood vessels, disease, benign tumors, or cancer.

The key part is this: the cancers that we are trying to detect will *always* be nodules; ones either suspended in the very non-dense tissue of the lung, or attached to the lung wall. That means that we can limit our classifier to only nodules, rather than having it examine all tissue. Being able to restrict the scope of expected inputs will help our classifier learn the task at hand.

This is another example of how the underlying deep learning techniques we'll use are universal, but they can't be applied blindly⁸⁹. We'll need to understand the field we're working in to make choices that will serve us well.

In 9.8 we can see a stereotypical example of a malignant nodule.

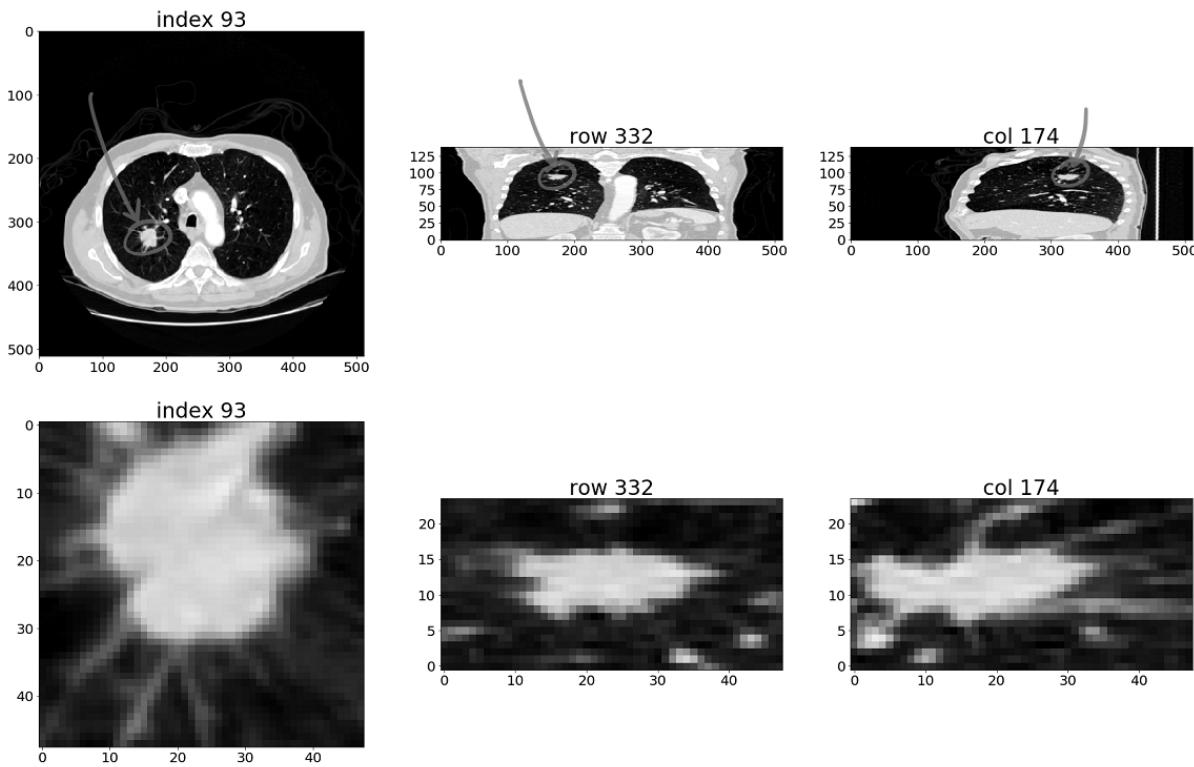


Figure 9.8 A CT scan with malignant nodule displaying visual discrepancy from other nodules.

The smallest nodules that we'll be concerned with are only a few millimeters across. As we discussed earlier in the chapter, this puts them at approximately a million times smaller than the CT scan as a whole. The vast majority of the nodules in a patient are not malignant;⁹⁰ the data set that we will be using has an approximate 400:1 ratio of benign to malignant nodules.

9.2.3 Our data source: the LUNA Grand Challenge

The CT scans we were just looking at come from the LUNA (LUNG Nodule Analysis) Grand Challenge. The LUNA grand challenge is the combination of an open data set with high-quality labels of patient CT scans (many with lung cancer) and a public ranking of classifiers against the data. The goal is to encourage improvements in cancer detection by making it easy for teams to compete for high positions on the leader board. A project team can test the efficacy of their detection methods against a standardized criteria (namely, the data set provided). To be included in the public ranking, a team must provide a scientific paper describing the project architecture, training methods, etc. This makes for a great resource to use to provide further ideas and inspiration for project improvements.

There is something of a culture of publicly sharing medical datasets for research and analysis. Open access to such data allows researchers to use, combine, and perform novel work on said data without having to enter into formal research agreements between institutions (obviously, some data is kept private as well).

NOTE

Many CT scans "in the wild" can be incredibly messy, in terms of various idiosyncrasies between various scanners and processing programs. For example, some scanners will indicate areas of the CT scan that are outside of the scanner's field-of-view by setting the density of those voxels to something negative. CT scans can also be acquired with a variety of settings on the CT scanner, which can change the resulting image in ways ranging from subtly to wildly different. While the LUNA data is generally clean, make sure to check your assumptions if you incorporate other data sources.

We will be using the LUNA 2016 data set. The LUNA site⁹¹ describes two "tracks" for the challenge. The first track, "Nodule detection (NDET)" roughly corresponds to our step 1 (segmentation), while the second track, "False positive reduction (FPRED)" is similar to our step 3 (classification). When the site discusses "locations of possible nodules" it is talking about a process similar to what we'll cover in chapter 13, 13.

9.2.4 How to download the LUNA data

Before we get further into the nuts-and-bolts of our project, we're going to cover how to get the data we're going to use. It's about 60GB of data compressed, and so depending on your connection to the internet it might take a while to download. Once uncompressed, it takes up about 120GB of space, and we'll need another 80GB or so of cache space, which we will use to store smaller chunks of data so that we can access it more quickly than reading in the whole CT.

Navigate to luna16.grand-challenge.org/download/⁹² and either register using email, or use the Google OAuth login. Once logged in, you should see three download options:

- Academic Torrents
- Dropbox
- Google Drive

We ended up using the Google Drive option, but the data is the same via all three.

NOTE

The `luna.grand-challenge.org` domain does not have links to the data download page as of this writing. If you are having issues finding the download page, double-check the domain for `luna16.` not `luna.,` and re-enter the URL if needed.

The data we will be using comes in ten subsets, aptly named `subset0` through `subset9`. Unzip each of them such that you have separate subdirectories like `code/data-unversioned/part2/luna/subset0`, etc. You'll need about 110GB of free space

to store the uncompressed data, and another 40GB or so to use as cache space to make loading faster. On Linux, you'll need the `7z` decompression utility.⁹³ Windows users can get an extractor from the 7-zip website.⁹⁴

In addition, we need the `candidates.csv` and `annotations.csv` files from `CSVFILES.zip`. We've included these files in the GitHub repository for convenience, and so they should already be present in `code/data/part2/luna/*.csv`. These can also be downloaded from same location as the data subsets.

NOTE

If you do not have easy access to 150GB of free disk space, then it's possible to run the examples using only one or two of the ten subsets of data. The smaller training set will result in the model performing much more poorly, but that's better than not being able to run the examples at all.

Once you have the candidates file and at least one subset downloaded, uncompressed, and put into location, you should be able to start running the examples in this chapter. If you want to jump ahead, you can use the `code/p2ch09_explore_data.ipynb` Jupyter Notebook to get started. Otherwise, we'll return to the notebook in more depth later in the chapter.

Hopefully your downloads will finish before you start reading the next chapter.

9.3 Conclusion

Congratulations, we've now made major strides towards finishing our project! Now, you might have the feeling that we haven't accomplished much. After all, we haven't implemented a single line of code yet! But keep in mind that research and preparation like we've done here will be needed when tackling projects on your own.

In this chapter, we set out to do two things:

1. Understand the larger context around our lung cancer detection project.
2. Sketch out the direction and structure of our project for part 2.

If you are still feeling that we haven't made real progress yet, please recognize that mindset as a trap — understanding the space your project is working in is crucial, and the design work we've done here will pay off handsomely as we move forward. We'll see those dividends shortly, once we start implementing our data loading routines in the next chapter.

Since this chapter has been informational only, without any code, we'll skip the exercises for now.

9.4 Summary

- Part 2 focuses on a single project. This will allow us to explore the project in-depth and understand how to work with messy real-world data on messy, real-world problems.
- Our approach to detecting cancerous nodules has five steps: data loading, segmentation, clustering, classification, and diagnosis. We'll start the next chapter with data loading. Classification will have several chapters dedicated to it, segmentation will get a chapter, and the two steps connecting those will be covered in the final chapter.
- Breaking down our project into smaller, semi-independent sub-projects makes teaching each sub-project easier. Other approaches might make more sense for future projects with different goals than the ones for this book.
- We use the LUNA Grand Challenge data for training our model. The LUNA data contains CT scans, as well as human-annotated outputs for classification and clustering. Having high-quality data makes a big impact on a project being successful.
- Nodules are small masses in the lungs. Most are benign, but a rare few are malignant. Most aspects of our project are going to revolve around nodules. Identifying the key concepts of a project and making sure they are well represented in your design can be crucial.
- A CT scan results in a three-dimensional array of intensity data with at least 32 million voxels. This is around a million times larger than the nodules we want to recognize. Being able to allow our network to focus on the data relevant to the task at hand will make it easier to get reasonable results from training.
- The array of CT scan data will typically not have cubic voxels; converting real-world units to array indexes requires conversion. The intensity of a CT scan corresponds roughly to mass density. Understanding aspects of our data like these will make it easier to write processing routines for our data that don't distort or destroy the important aspects of the data.

10

Ready, Dataset, Go!

This chapter covers:

- Loading and processing our raw data files; these are the annotations that describe the location of potentially malignant parts of a CT scan
- Implementing a Python class to represent our data to the rest of our project; for us, this will be the `CT` class
- Converting our data into a format usable by PyTorch by implementing a `Dataset` subclass; the `LunaDataset` class will combine the CT and annotation data and convert it into tensors
- Visualizing the data we will be using as training and validation data for the project

Now that we've covered the larger project for part 2, let's get into specifics about what we're going to do here in chapter 10. It's time to implement basic data loading and processing routines for our raw data. Basically every significant project you work on will need something analogous to what we cover here.⁹⁵ Here is our high-level map of our project from chapter 9, shown here in 10.1. We're going to be focusing on step 1, data loading for the rest of this chapter.

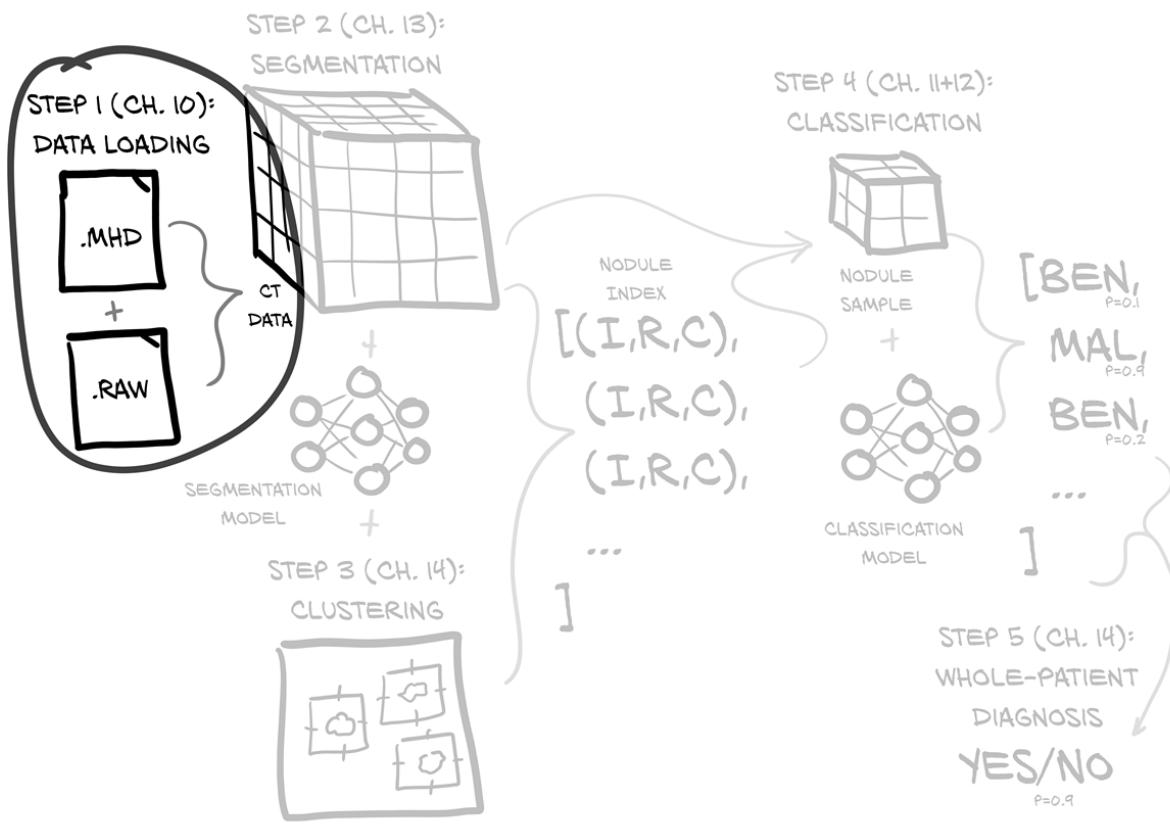


Figure 10.1 Our end-to-end lung cancer detection project, with a focus on this chapter's topic; step 1, data loading.

Our goal is to be able to produce a training sample given our inputs of raw CT scan data and a list of annotations for those CTs. This might sound simple, but there's actually quite a bit that needs to happen before we can load, process, and extract the data we're interested. 10.2 shows us what we'll need to do to go from our raw data and turn it into a training sample. Luckily, we've got a head start on *understanding* our data from last chapter, but there's going to be more work to do on that front as well.

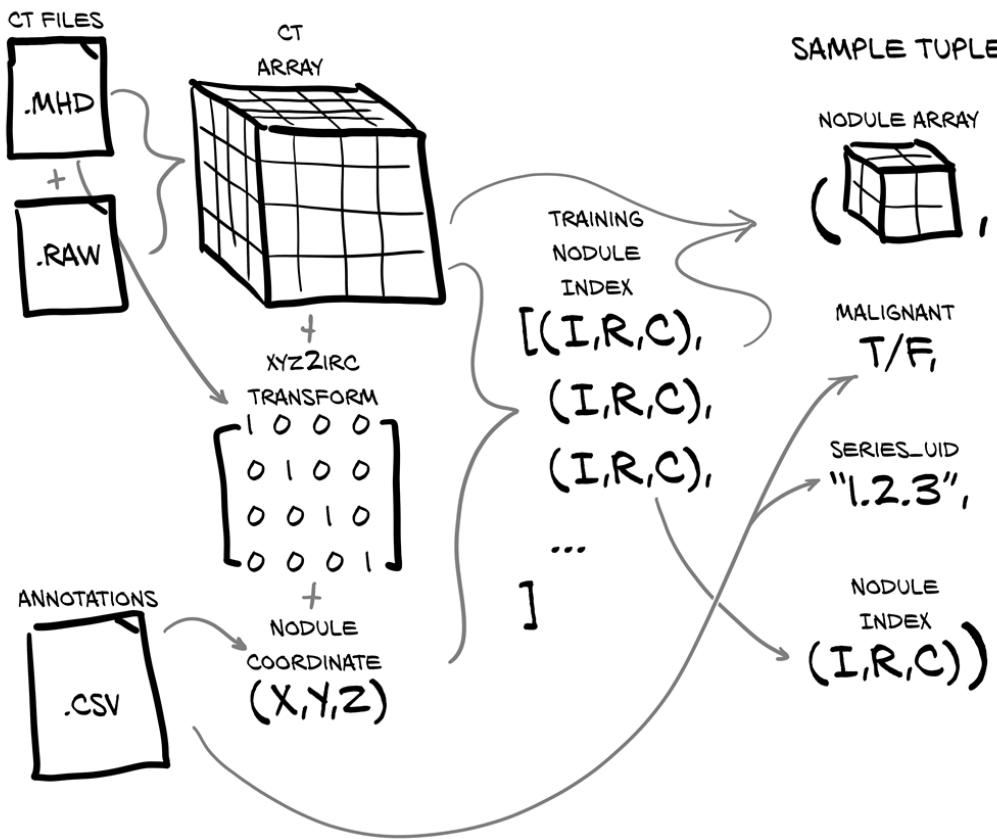


Figure 10.2 The data transforms required to make a sample tuple, as returned by `LunaDataset.{uu}getitem{uu}`. These sample tuples will be used as input to our model training routine.

This is a crucial moment, where we begin to transmute the leaden raw data, if not into gold, then at least into the stuff that our neural network will spin *into* gold. We discussed the mechanics of this transformation in chapter 4, 4, but there are some subtle decisions that we'll need to make as well. Limiting or cropping our data so as to not drown our model in noise is important, as is making sure we're not so aggressive that our signal gets cropped out of our input. We want to make sure that the range of our data is well-behaved, especially after normalization. Clamping our data to remove outliers can be useful, especially if our data is prone to extreme outliers. We can also create hand-crafted, algorithmic transformations of our input; this is known as *feature engineering*, and we discussed it briefly in chapter 1, 1.3.1. We'll usually want to let the model do most of the heavy lifting, but feature engineering has its uses, and we'll see a practical example of it in chapter 13, 13.3.3.

Let's get back to the problem at hand: producing a Dataset containing our CT scan data. Our CT data comes in two files, a `.mhd` file containing metadata header information, and a `.raw` file that has the raw bytes that make up the 3D array. Each file is named starting with the unique identifier called the *Series UID*⁹⁶ for the CT scan in question, so for series UID "1.2.3" there would be two files, `1.2.3.mhd` and `1.2.3.raw`.

Our `ct` class will consume those two files, and produce the 3D array, as well as the transformation matrix to convert from the patient coordinate system (which we will discuss in more detail in 10.3) to the index, row, column coordinates needed by the array (these coordinates are shown as "(I,R,C)" in the figures, and are denoted with `_irc` variable suffixes in the code). Don't sweat the details of all this right now; just remember that we've got some coordinate system conversion to do before we can apply these coordinates to our CT data. The details will get explored as we need them.

We will also load the annotation data provided by LUNA, which will give us a list of nodule coordinates, each with malignancy flag, along with the series UID of the relevant CT scan. By combining the nodule coordinate with coordinate system transformation information, we get the index, row, and column of the voxel at the center of our nodule.

Using the "(I,R,C)" coordinates we can take a small 3D slice of our CT data and using that as the input to our model. In order to do so, we must construct our training sample tuple, which will have the sample array, malignancy flag, the series UID, and the nodule index in the CT data array. This sample tuple is exactly what PyTorch expects from our `Dataset` subclass, and represents the last section of our bridge from our original raw data to the standard structure of PyTorch tensors.

10.1 Parsing LUNA's annotation data

The first thing that we're going to want to do is to start loading our data. When working with a new project, that's often a good place to start. Making sure we know how to work with the raw input is required no matter what, and knowing how your data will look post-loading can help inform the structure of your early experiments. We could try 10.2 first, but we think it makes sense to start with parsing the CSV files that LUNA provides containing information about the points of interest inside of each CT scan. Since there are fewer types of information in the CSV files and they're easier to parse, we're hoping that they will give us some clues about what to look for once we start loading CTs themselves.

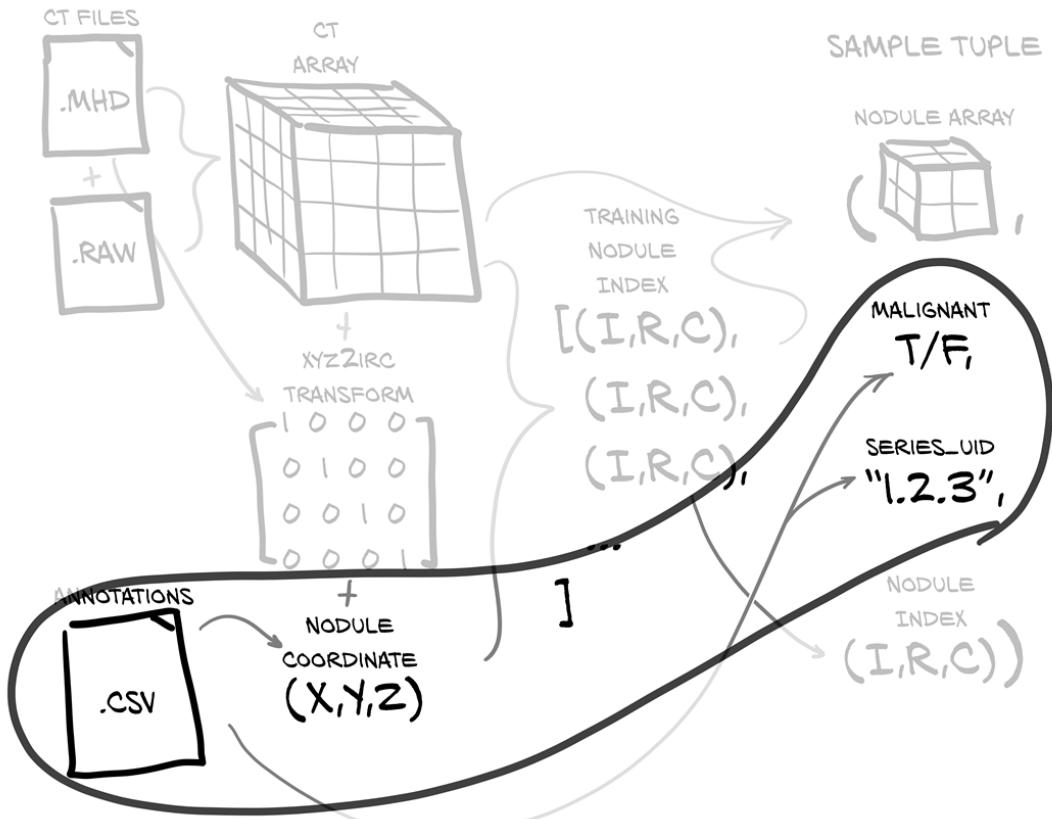


Figure 10.3 The LUNA annotations in candidates.csv contain the CT series, nodule position, and a malignancy flag.

For any standard supervised learning task (classification the prototypical example), we're going to want to split our data into training and validation sets. The `candidates.csv` file contains information about all nodules, both malignant and benign. We'll use this as the basis building a complete list of nodules, and that can then be split into our training and validation data sets. Let's see what the file contains:

Listing 10.1 Bash shell session

```
$ wc -l candidates.csv      ①
551066 candidates.csv

$ head data/part2/luna/candidates.csv      ②
seriesuid,coordX,coordY,coordZ,class
1.3...6860,-56.08,-67.85,-311.92,0
1.3...6860,53.21,-244.41,-245.17,0
1.3...6860,103.66,-121.8,-286.62,0
1.3...6860,-33.66,-72.75,-308.41,0
...
$ grep ',1$' candidates.csv | wc -l      ④
1351
```

- ① This counts the number of lines in the file.
- ② This prints the first few lines of the file.

- ③ The first line of the `.csv` file defines the column headers.
- ④ This counts the number of lines that end with a 1, which indicates malignancy.

NOTE

The values of the `seriesuid` column have been elided to better fit the printed page.

So 551k lines, each with a `seriesuid` (that we'll call `series_uid` in the code), some X,Y,Z coordinates, and a `class` column that corresponds to malignancy (it's a boolean value; 0 for benign, and 1 for malignant). We have 1351 nodules flagged as malignant.

The `annotations.csv` file contains information about some of the nodules that have been flagged as malignant. We are interested in the `diameter_mm` information in particular.

Listing 10.2 Bash shell session

```
$ wc -l annotations.csv
1187 annotations.csv ①

$ head data/part2/luna/annotations.csv
seriesuid,coordX,coordY,coordZ,diameter_mm ②
1.3.6...6860,-128.6994211,-175.3192718,-298.3875064,5.651470635
1.3.6...6860,103.7836509,-211.9251487,-227.12125,4.224708481
1.3.6...5208,69.63901724,-140.9445859,876.3744957,5.786347814
1.3.6...0405,-24.0138242,192.1024053,-391.0812764,8.143261683
...
```

- ① This is a different number from the `candidates.csv` file.
- ② The last column is different also.

This means we've got size information for about about 1200 nodules. This is useful, since we can use it to make sure that our training and validation data has a representative spread of nodule sizes. Without this, it's possible that our validation set could end up with only extreme values, making it seem like our model is under-performing.

We want to make sure that our training set and validation set are both *representative* of the range of input data we're interested in being able to handle properly. If either of those sets are meaningfully different from our real-world use cases, it's pretty likely that our model is going to behave differently than we expect. All of the training and statistics we collected won't be predictive once we transfer over to production use! We're not trying to make this an exact science, but you should keep an eye out in future projects for hints that your training and testing on data that doesn't actually make sense for your operating environment.

Let's get back to our nodules. We're going to sort our nodules by size, and take every Nth one for our validation set. That should give us that representative spread we're looking for. Unfortunately, the location information provided in `annotations.csv` doesn't always precisely

line up with the coordinates in `candidates.csv`:

```
$ grep 1.3.6.1.4.1.14519.5.2.1.6279.6001.100225287222365663678666836860 annotations.csv
1.3.6...6860,-128.6994211,-175.3192718,-298.3875064,5.651470635
1.3.6...6860,103.7836509,-211.9251487,-227.12125,4.224708481

$ grep '1.3.6.1.4.1.14519.5.2.1.6279.6001.100225287222365663678666836860.*,1$' candidates.csv
1.3.6...6860,104.16480444,-211.685591018,-227.011363746,1
1.3.6...6860,-128.94,-175.04,-297.87,1
```

If we truncate the corresponding coordinates from each file, we end up with `-128.70, -175.32, -298.39` vs `-128.94, -175.04, -297.87`. Since the nodule in question has a diameter of 5mm, both of these points are clearly meant to be the "center" of the nodule, but they don't line up exactly. Now, it'd be a perfectly valid response to decide that dealing with this data mismatch isn't worth it, and to ignore the file. We are going to do the legwork to make things line up, though, since real-world data sets are often imperfect this way, and this is a good example of the kind of work that will need to get done to assemble data from disparate data sources.

Now that we know what our raw data files look like, let's build a `getNoduleInfoList` function that will stitch it all together. We'll be using a named tuple that defined at the top of the file to hold the information for each nodule:

Listing 10.3 p2ch10/dsets.py, line 7

```
from collections import namedtuple
# ... line 27
NoduleInfoTuple = namedtuple('NoduleInfoTuple', 'isMalignant_bool, diameter_mm,
                                series_uid, center_xyz')
```

These tuples are *not* our training samples, as they're missing the chunks of CT data we need. Instead, these represent a sanitized, cleaned, unified interface to the human-annotated data we're using. It's very important to isolate having to deal with messy data from the model training. Without that, your training loop can get cluttered quickly, as you have to keep dealing with special cases and other distractions in the middle of code that should be focused on training.

Our list of nodule information will have malignancy (what we're going to be training the model to classify), diameter (useful for getting a good spread in training, since large and small nodules will not have the same features), series (to locate the correct CT scan), and the nodule center (to find the nodule in the larger CT). The function that will build a list of these `NoduleInfoTuple` instances starts by using an in-memory caching decorator, followed by getting the list of files present on disk.

Listing 10.4 p2ch10/dsets.py, line 29

```
@functools.lru_cache(1) ①
def getNoduleInfoList(requireDataOnDisk_bool=True): ②
    mhd_list = glob.glob('data-unversioned/part2/luna/subset*/*.mhd')
    dataPresentOnDisk_set = {os.path.split(p)[-1][:4] for p in mhd_list}
```

- ① Standard library in-memory caching.
- ② `requireDataOnDisk_bool` defaults to screening out series from data subsets that aren't in place yet.

Since parsing some of the data files can be slow, we'll cache the results of this function call in memory. This will come in handy later, since we'll be calling this function more often in future chapters. Speeding up your data pipeline by carefully applying in-memory or on-disk caching can result in some pretty impressive training speed gains. Keep an eye out for these opportunities as you work on your projects.

Earlier we said that we'll support running our training program with less than the full set of training data, due to the long download times and high disk space requirements. The `requireDataOnDisk_bool` parameter is what makes good on that promise; we're detecting which LUNA series UIDs are actually present and ready to be loaded from disk, and will use that to limit which entries we use from the CSV files we're about to parse. Being able to run a subset of your data through the training loop can be useful to verify that your code is working as intended. Often your model's training results will be bad-to-useless when doing so, but exercising your logging, metrics, model checkpointing, and similar functionality is useful.

After we get our candidate nodule information, we want to merge in the diameter information from `annotations.csv`. First, we need to take our annotations and group them by `series_uid`, as that's the first key we'll use to cross-reference data by the two files.

Listing 10.5 p2ch10/dsets.py, line 37: def getNoduleInfoList

```
diameter_dict = {}
with open('data/part2/luna/annotations.csv', "r") as f:
    for row in list(csv.reader(f))[1:]:
        series_uid = row[0]
        annotationCenter_xyz = tuple([float(x) for x in row[1:4]])
        annotationDiameter_mm = float(row[4])

        diameter_dict.setdefault(series_uid, []).append((annotationCenter_xyz, annotationDiameter_mm))
```

Now we'll build our full list of nodules using the information in the `candidates.csv` file. For each of the candidate entries for a given `series_uid`, we will loop through the annotations we collected earlier for the same `series_uid` and see if the two coordinates are close enough to consider them the same nodule. If they are, great! Now we have diameter information for that nodule. If we don't find a match, that's fine, we'll just treat the nodule as having a `0.0` diameter. Since we're only using this information to get a good spread of nodule sizes in our training and validation sets, having incorrect diameter sizes for some nodules shouldn't be a problem, but it's something we should remember we're doing, in case our assumption here is wrong.

Listing 10.6 p2ch10/dsets.py, line 46: def getNoduleInfoList

```
noduleInfo_list = []
with open('data/part2/luna/candidates.csv', "r") as f:
    for row in list(csv.reader(f))[1:]:
        series_uid = row[0]

        if series_uid not in dataPresentOnDisk_set and requireDataOnDisk_bool: ❶
            continue

        isMalignant_bool = bool(int(row[4]))
        candidateCenter_xyz = tuple([float(x) for x in row[1:4]])

        candidateDiameter_mm = 0.0
        for annotationCenter_xyz, annotationDiameter_mm in diameter_dict.get(series_uid, []):
            for i in range(3):
                delta_mm = abs(candidateCenter_xyz[i] - annotationCenter_xyz[i])
                if delta_mm > annotationDiameter_mm / 4: ❷
                    break
            else:
                candidateDiameter_mm = annotationDiameter_mm
                break

        noduleInfo_list.append(NoduleInfoTuple(isMalignant_bool, candidateDiameter_mm, series_uid,
                                                candidateCenter_xyz))
```

- ❶ If a series_uid isn't present, that means it's in a subset that we don't have on disk, so we should skip it.
- ❷ We divide the diameter by 2 to get the radius, and the radius by 2 to require that the two nodule center points not be too far apart, relative to the size of the nodule.
Note that this results in a bounding-box check, not a true distance check.

That's a lot of somewhat fiddly code, just to merge in our nodule diameter. Unfortunately, having to do this kind of manipulation and fuzzy matching can be fairly common, depending on your raw data. Once we get to this point, however, we just need to sort the data and return it:

Listing 10.7 p2ch10/dsets.py, line 69: def getNoduleInfoList

```
noduleInfo_list.sort(reverse=True) ❶
return noduleInfo_list
```

- ❶ This means that we have all of the malignant samples starting with the largest first, followed by all of the benign samples (which don't have nodule size information).

The ordering of the tuple members in `noduleInfo_list` is being driven by this sort. We're sorting this like we are to help make sure that when we take a slice of the data, that slice gets a representative chunk of the malignant nodules, with a good spread of nodule diameters. We'll discuss this more in 10.4.3.

10.2 Loading individual CT scans

Next up, we need to be able to take our CT data from a pile of bits on disk and turn it into a python object that we can use to extract 3D nodule density data from. This is also pretty common; our nodule information is acting like a map to the interesting parts of our raw data. Before we can follow that map to our data of interest, we need to get the data into an addressable form.

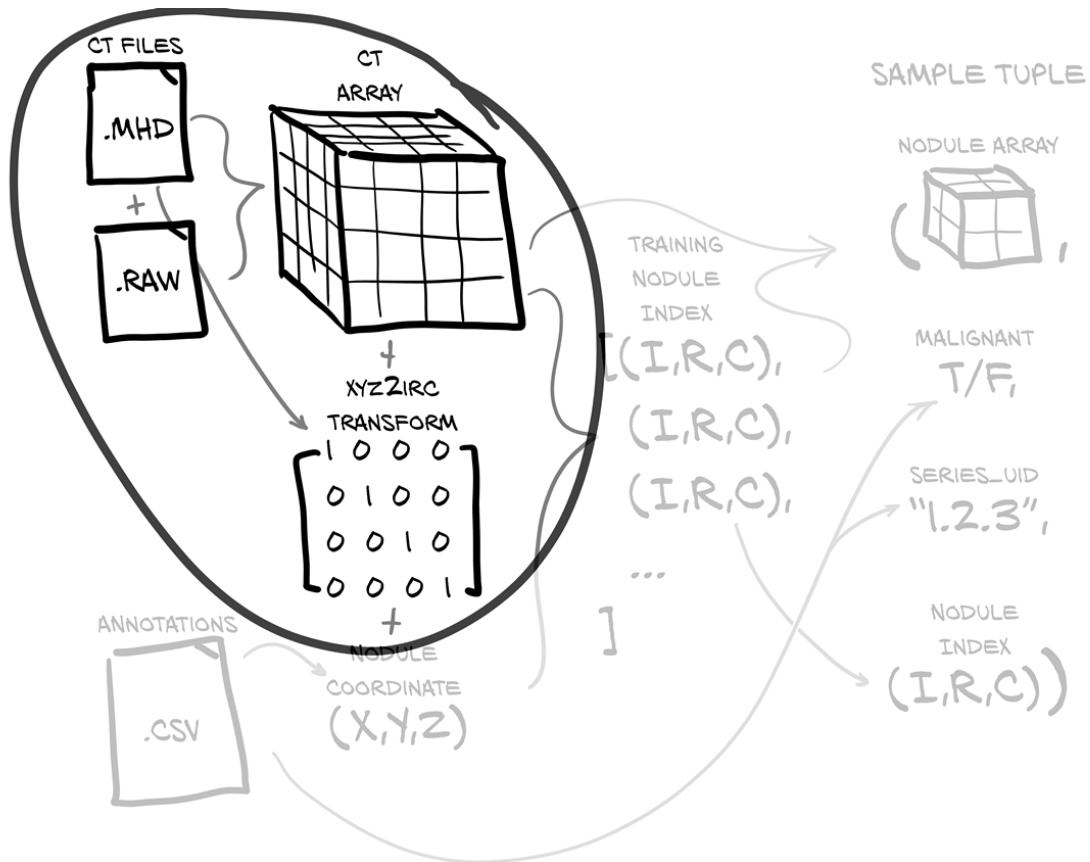


Figure 10.4 Loading a CT scan produces a voxel array and a transformation from patient coordinates to array indices.

The native file format for CT scans is DICOM⁹⁷, which is an acronym for Digital Imaging and COmmunication in Medicine. The first version of the DICOM standard was authored in 1984, and as one might expect from anything computing-related from then, it's a bit of a mess (for example, there are whole sections that are now retired that used to be devoted to the data link layer protocol to use, since ethernet hadn't won yet).

NOTE

We've done the legwork of finding the right library to parse these raw data files, but for other formats you've never heard of, you'll have to find a parser yourself. We recommend spending the time to do so! The Python ecosystem has parsers for just about every file format under the sun, and your time is almost certainly better spent working on the novel parts of your project, rather than writing parsers for esoteric data formats.

Happily, LUNA has converted the data that we're going to be using for this chapter into the MetaIO format, which is quite a bit easier to use.⁹⁸ Don't worry if you've never heard of the format before! We can treat the format of those data files as a black box, and use `SimpleITK` to load them into more familiar numpy arrays.

Listing 10.8 p2ch10/dsets.py, line 9

```
import SimpleITK as sitk
# ... line 72
class Ct:
    def __init__(self, series_uid):
        mhd_path = glob.glob('data-unversioned/part2/luna/subset*/{}.{mhd}'.format(series_uid))[0] ❶

        ct_mhd = sitk.ReadImage(mhd_path) ❷
        ct_a = np.array(sitk.GetArrayFromImage(ct_mhd), dtype=np.float32) ❸
```

- ❶ We don't care to track which subset a given `series_uid` is in, so we wildcard the subset here.
- ❷ `sitk.ReadImage` implicitly consumes the `.raw` file in addition to the passed-in `.mhd` file.
- ❸ We re-create an `np.array` here, since we want to convert the value type to `np.float32`

For real projects you'll want to understand what types of information are contained in your raw data, but it's perfectly fine to rely on third-party code like `SimpleITK` to actually parse the bits on disk. Finding the right balance of knowing everything about your inputs versus blindly accepting whatever your data loading library hands you will probably take some experience. Just remember that you're mostly concerned about *data* not *bits*. It's the information that matters, not how it's represented.

Being able to uniquely identify a given sample of your data can be useful. For example, being able to clearly communicate which sample is causing a problem, or is getting poor classification results can drastically improve your ability to isolate and debug the issue. Depending on the nature of your samples, sometimes that unique identifier will be an atom, like a number or a string, and sometimes it might be more complicated, like a tuple.

We identify specific CT scans using the *Series Instance UID*, or just `series_uid`, assigned when the CT scan was created. DICOM makes heavy use of unique identifiers ("UIDs") for

individual DICOM files, groups of files, courses of treatment, and similar. These identifiers are similar in concept to UUIDs⁹⁹, but have a different creation process, and are formatted differently. For our purposes, we can treat them as opaque ASCII strings that serve as unique keys to reference the various CT scans by. Officially, only the characters 0 through 9 and . (period) are valid characters in a DICOM UID, but some DICOM files in the wild have been anonymized with routines that replace the UIDs with hexadecimal (0-9, and a-f) or other technically out-of-spec values (these out-of-spec values aren't typically flagged or cleaned by DICOM parsers; as we said before, it's a bit of a mess).

The ten subsets we discussed earlier have about ninety CT scans each (888 in total), with every CT scan represented as two files; one with a .mhd and one a .raw extension. This is hidden behind the `sitk` routines, however, and is not something we need to be directly concerned with.

At this point, `ct_a` is a three-dimensional array. All three dimensions are spatial, and the single intensity channel is implicit. As we saw in chapter 4, 4.2, in a PyTorch tensor, the channel information would be represented as a fourth dimension with size 1.

10.2.1 Hounsfield Units

If you recall, we wrote earlier that we need to understand our *data*, not the *bits* that store it. Here, we have a perfect example of that in action. Without understanding the nuances of our data's values and range, we'll end up feeding values into our model that will hinder it's ability to learn what we want it to.

Continuing the `{uu}init{uu}` method, we need to do a bit of cleanup on the `ct_a` values. CT scan voxels are expressed in Hounsfield Units¹⁰⁰ or "HU", which are odd units; air is -1000 HU (close enough to 0 g/cc for our purposes), water is 0 HU (1 g/cc), and bone is at least +1000 HU (2-3 g/cc).

NOTE The HU values are typically stored on disk as signed 12-bit integers (shoved into 16-bit integers), which fits well with the level of precision that CT scanners can provide. While perhaps interesting, it's not particularly relevant to the project.

Some CT scanners will use HU values that correspond to negative densities to indicate that those voxels are outside of the CT scanners field of view. For our purposes, everything outside of the patient should be air, so we discard that field of view information by setting a lower bound of the values to -1000 HU. Similarly, the exact density of bones, metal implants, etc. are not particularly relevant to our use case, so we cap density at roughly 2 g/cc (1000 HU), even though that's not biologically accurate in most cases.

Listing 10.9 p2ch10/dsets.py, line 83: class Ct.{uu}init{uu}

```
ct_a.clip(-1000, 1000, ct_a)
```

Values above 0 HU don't scale perfectly with density, but the tumors that we're interested in are typically around 1 g/cc (0 HU), and so we're going to ignore that HU doesn't map perfectly to common units like g/cc. That's fine, since our model is going to be trained to consume HU directly.

We want to remove all of these outlier values from our data, since they aren't directly relevant to our goal, and having those outliers can make the model's job harder. This can happen in many ways, but a common example is when batch normalization is fed these outlier values, the statistics about how to best normalize the data can be skewed. Always be on the lookout for ways to clean your data.

All of the values we've built then get assigned to `self`:

Listing 10.10 p2ch10/dsets.py, line 85: class Ct.{uu}init{uu}

```
self.series_uid = series_uid
self.hu_a = ct_a
```

It's important to know that our data uses the range of -1000 to +1000, however, since in chapter 13, 13.2.4, we end up adding additional channels of information to our samples. If we don't account for the disparity between HU and our additional data, those new channels can easily be overshadowed by the raw HU values. Since we won't be adding additional channels of data for the classification step of our project, we don't need to implement special handling right now.

10.3 Locating a nodule using the patient coordinate system

Deep learning models typically need fixed-size inputs,¹⁰¹ due to having a fixed number of input neurons. We need to be able to produce a fixed-size array containing the nodule so that we can use that as input to our classifier. We'd like to train our model using a crop of the CT scan that has a nodule nicely-centered, since then our model doesn't have to learn how to notice nodules tucked away in the corner of the input. By reducing the variation in expected inputs, we make the model's job easier.

Unfortunately, all of our nodule center data we loaded up previously in 10.1 is expressed in millimeters, not voxels! We can't just plug locations in millimeters into an array index and expect everything to work out the way we want. As we can see in 10.5 we'll need to transform our coordinates from the millimeter-based coordinate system (X, Y, Z) they're expressed in, into the voxel-address-based coordinate system (I, R, C) we need to be able to take array slices from our CT scan data. This is a classic example of how it's important to handle units consistently!

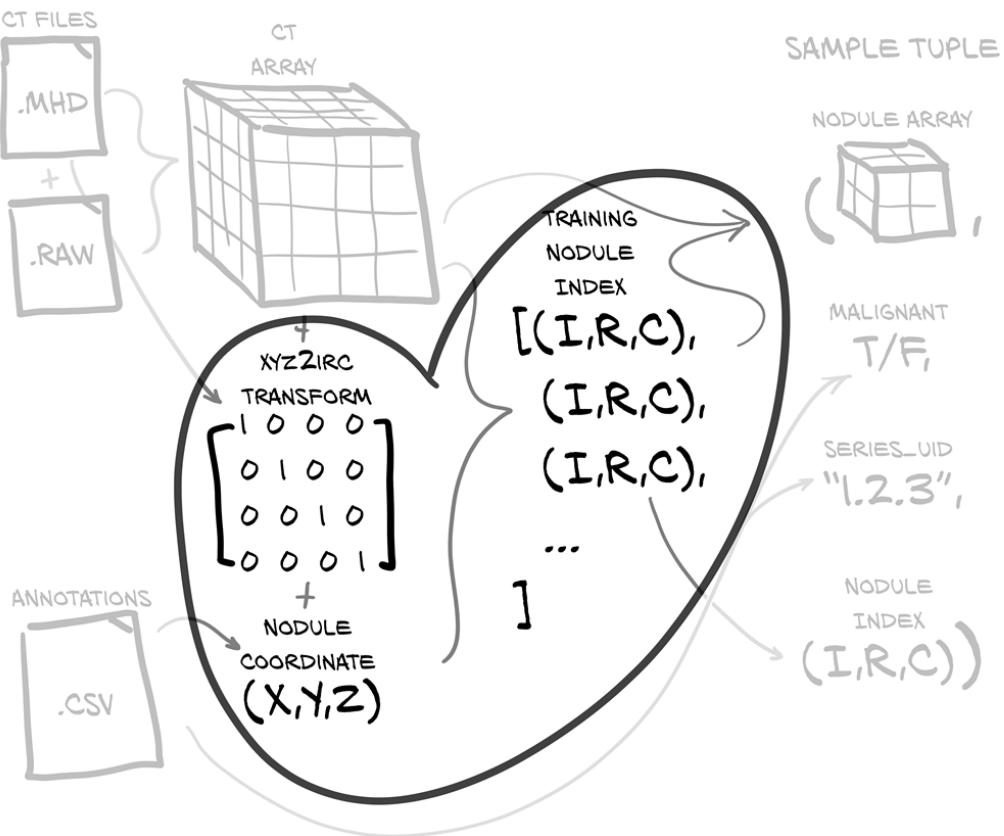


Figure 10.5 Using the transformation information to convert a nodule center coordinate in patient coordinates (X, Y, Z) to an array index (Index, Row, Column).

When dealing with CT scans, we will refer to the array dimensions as index, row, and column, as a separate meaning exists for X, Y, and Z, which is illustrated in 10.6. There is a "patient" coordinate system that defines positive X to be patient-left ("left"), positive Y to be patient-behind ("posterior"), and positive Z to be toward-patient-head ("superior"). Left-posterior-superior is sometimes abbreviated "LPS".

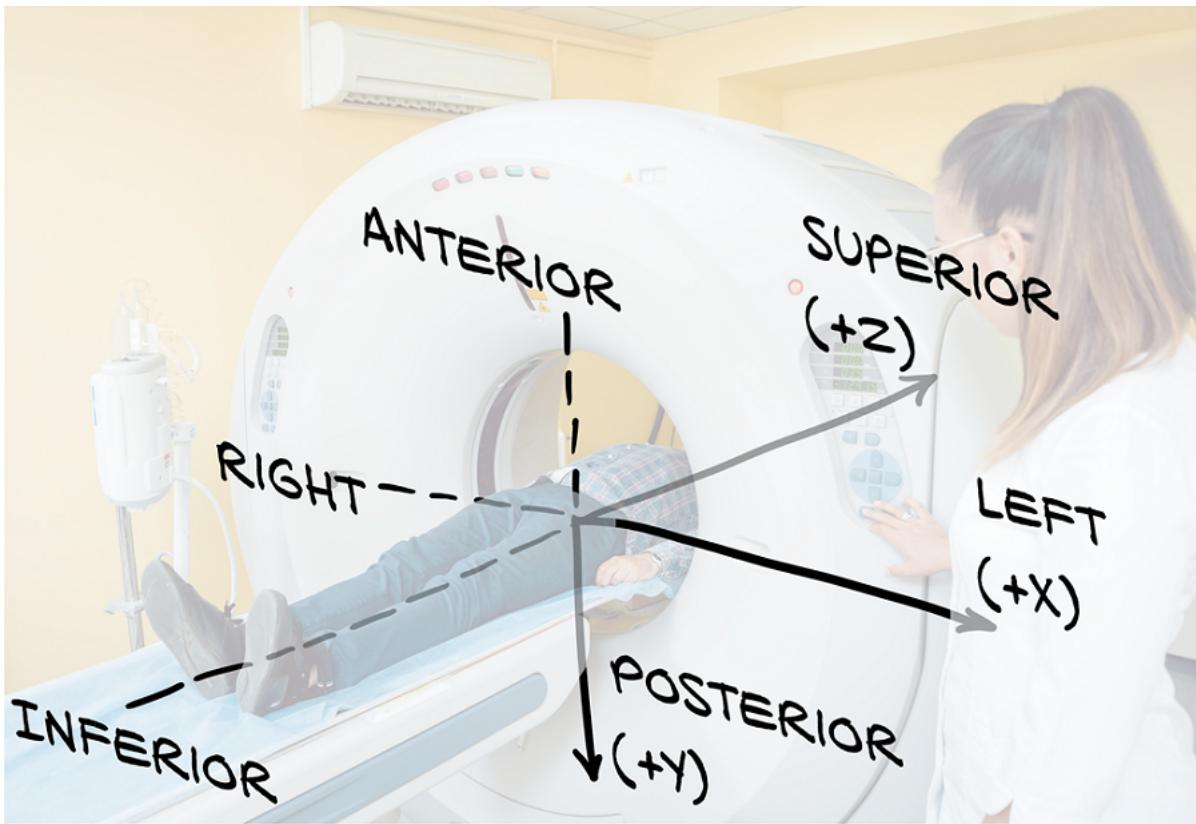


Figure 10.6 Our inappropriately-clothed patient demonstrating the axes of the patient coordinate system.

The patient coordinate system is measured in millimeters, and has an arbitrarily-positioned origin that does not correspond to the origin of the CT voxel array, as shown in 10.7.

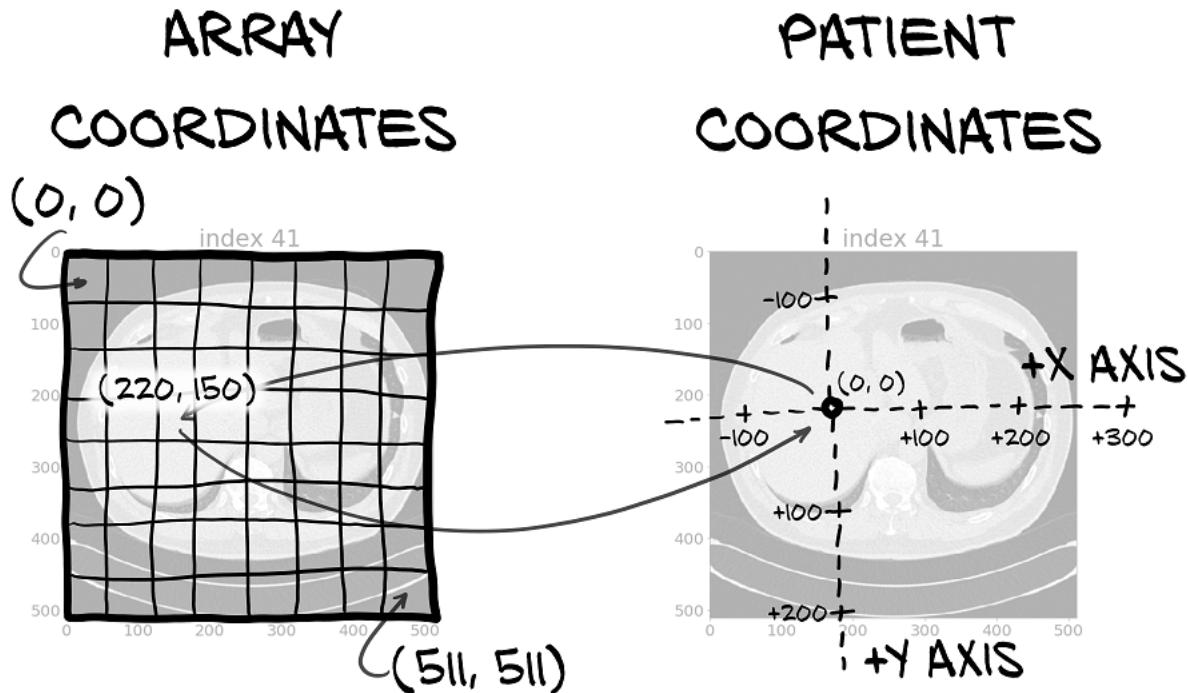


Figure 10.7 Array coordinates and patient coordinates have different origins and scaling.

The patient coordinate system is often used to specify the locations of interesting anatomy in a way that is independent of any particular scan. The metadata that defines the relationship between the CT array and the patient coordinate system is stored in the header of DICOM files, and that Meta Image format preserves the data in its header as well. This metadata allows us to construct the transformation from (X,Y,Z) to (I,R,C) we saw in 10.5. The raw data contains many other fields of similar metadata, but since we don't have a use for them right now, those unneeded fields will be ignored.

One of the most common variations between CT scans in the size of the voxels; they are typically not cubes. Instead, they can be 1.125mm x 1.125mm x 2.5mm or similar. Usually the row and column dimensions have voxel sizes that are the same, and the index dimension has a larger value, but other ratios can exist.

When plotted using square pixels, the non-cubic voxels can end up looking somewhat distorted, somewhat similar to the distortion near the north and south poles when using a mercator projection map. That's an imperfect analogy, since in this case the distortion is uniform and linear — the patient looks far more squat or barrel-chested in 10.8 than they would in reality. We will need to apply a scaling factor if we want the images to depict realistic proportions.

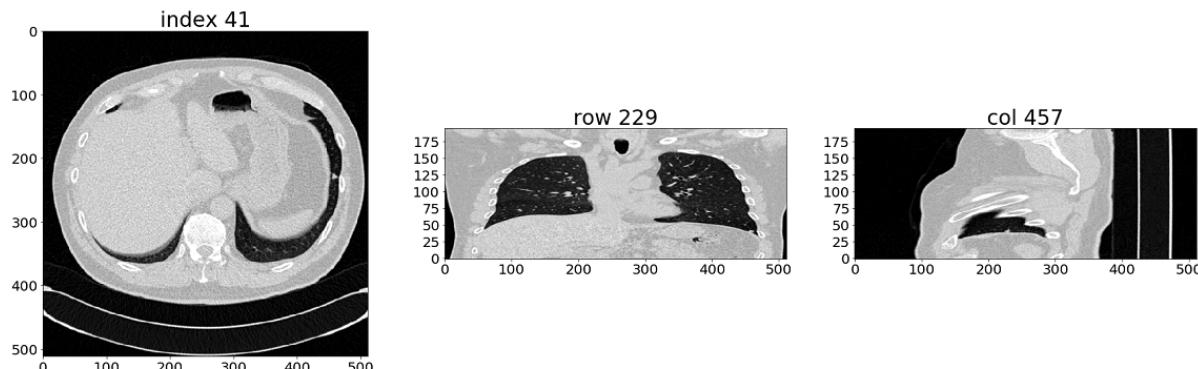


Figure 10.8 A CT scan with non-cubic voxels along the index-axis. Note how compressed the lungs are top-to-bottom.

Knowing these kinds of details can help when trying to interpret our results visually. Without this information, it would be easy to assume that something in our data loading was wrong! We might think that the data looked so squat because we were skipping half of the slices by accident, or something along those lines. It can be easy to waste a lot of time debugging something that's been working all along, and knowing your data can help prevent that.

Commonly, CTs are 512 rows by 512 columns, with the index dimension ranging from around 100 total slices up to perhaps 250 slices (250 slices times 2.5 millimeters is typically enough to contain the anatomical region of interest). This results in a lower bound of approximately 2^{25} voxels, or about 32 million data points. Each CT specifies the voxel size in millimeters as part of the file metadata (that was the `ct_mhd.GetSpacing()` we just saw).

We will define some utility code to assist with conversion between patient coordinates in millimeters (which we will denote in code by an `_xyz` suffix on variables and the like) and array coordinates in index, row, and columns (which we will denote in code by an `_irc` suffix).

Listing 10.11 p2ch10_first_dataset.adoc

```
IrcTuple = collections.namedtuple('IrcTuple', ['index', 'row', 'col'])
XyzTuple = collections.namedtuple('XyzTuple', ['x', 'y', 'z'])

def xyz2irc(coord_xyz, origin_xyz, vxSize_xyz, direction_tup):
    # Note: _cri means Col,Row,Index
    if direction_tup == (1, 0, 0, 0, 1, 0, 0, 0, 1):
        direction_ary = np.ones((3,))
    elif direction_tup == (-1, 0, 0, 0, -1, 0, 0, 0, 1):
        direction_ary = np.array((-1, -1, 1))
    else:
        raise Exception(
            "Unsupported direction_tup: {}".format(direction_tup),
        )

    coord_cri = (
        np.array(coord_xyz)
        - np.array(origin_xyz)
        ) / np.array(vxSize_xyz) ①
    coord_cri *= direction_ary
    return IrcTuple(*list(reversed(coord_cri.tolist())))

def irc2xyz(coord_irc, origin_xyz, vxSize_xyz, direction_tup):
    # Note: _cri means Col,Row,Index

    coord_cri = np.array(list(reversed(coord_irc))) ②
    if direction_tup == (1, 0, 0, 0, 1, 0, 0, 0, 1):
        direction_ary = np.ones((3,))
    elif direction_tup == (-1, 0, 0, 0, -1, 0, 0, 0, 1):
        direction_ary = np.array((-1, -1, 1))
    else:
        raise Exception(
            "Unsupported direction_tup: {}".format(direction_tup),
        )

    coord_xyz = coord_cri \
        * direction_ary \
        * np.array(vxSize_xyz) \
        + np.array(origin_xyz)
    return XyzTuple(*coord_xyz.tolist())
```

- ① Some CTs are "turned around" from the typical orientation. These blocks handle those cases.
- ② The suffix `_cri` here means Col, Row, Index; an ordering that is unused outside of these two functions.

NOTE

The transformations implemented in these two functions could easily be implemented as a standard 4x4 transformation matrix. We chose to use named tuples because we felt that resulted in code that is easier to read and understand for readers that might not be familiar with 4x4 transformation matrices.

The metadata we need to convert from patient coordinates (_xyz) to array coordinates (_irc) is contained in the MetaIO file alongside the CT data itself. We pull those out at the same time we get the `ct_a`:

Listing 10.12 p2ch10/dsets.py, line 72: class Ct

```
class Ct:  
    def __init__(self, series_uid):  
        mhd_path = glob.glob('data-unversioned/part2/luna/subset*/{}.{mhd}'.format(series_uid))[0]  
  
        ct_mhd = sitk.ReadImage(mhd_path)  
        # ... line 91  
        self.origin_xyz = XyzTuple(*ct_mhd.GetOrigin())  
        self.vxSize_xyz = XyzTuple(*ct_mhd.GetSpacing())  
        self.direction_tup = tuple(int(round(x)) for x in ct_mhd.GetDirection())
```

These are the inputs we need to pass into our `xyz2irc` conversion function, aside from the individual point to convert. With this attributes, our CT object implementation now has all of the data needed to convert a nodule center from patient coordinates to array coordinates.

10.3.1 Extracting a nodule from a CT scan

As we mentioned last chapter, up to 99.9999% of the voxels in a CT scan of a patient with a malignant nodule won't be cancer. Again, that ratio is equivalent to a two-pixel blob of incorrectly tinted color somewhere on a high-definition television, or a single misspelled word out of a large trilogy of novels. Forcing our model to examine such huge swathes of data looking for the hints of malignancy we want it to focus on is going to work about as well as asking you to find a single misspelled word¹⁰² from a set of novels written in a language you don't know!

Instead, as we can see in 10.9, we are going to extract an area around each nodule, and let the model focus on one nodule at a time. This is akin to letting you read individual paragraphs in that foreign language; still not an easy task, but far less daunting! Looking for ways to reduce the scope of the problem for your model can help, especially in the early stages of a project when you're trying to get your first working implementation up and running.

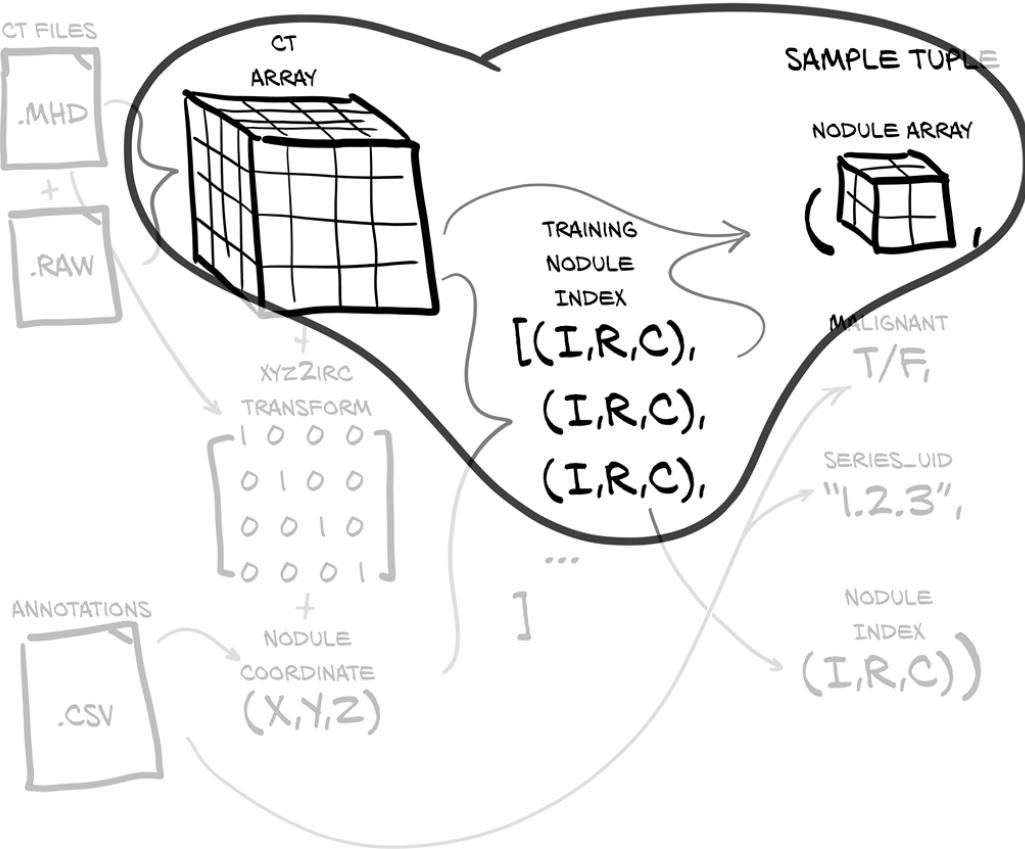


Figure 10.9 Cropping a nodule sample out of the larger CT voxel array using the nodule center's array coordinate information (Index, Row, Column).

The `getRawNodule` function takes the center expressed in the patient coordinate system (X,Y,Z), just as it's specified in the LUNA CSV data, as well as a width in voxels. It returns a cubic chunk of CT, as well as the center of the nodule converted to array coordinates. Keep in mind that these array coordinates are not integers!

Listing 10.13 p2ch10/dsets.py, line 92: class Ct.getRawNodule

```
def getRawNodule(self, center_xyz, width_irc):
    center_irc = xyz2irc(center_xyz, self.origin_xyz, self.vxSize_xyz, self.direction_tup)

    slice_list = []
    for axis, center_val in enumerate(center_irc):
        start_ndx = int(round(center_val - width_irc[axis]/2))
        end_ndx = int(start_ndx + width_irc[axis])
        slice_list.append(slice(start_ndx, end_ndx))

    ct_chunk = self.hu_a[tuple(slice_list)]

    return ct_chunk, center_irc
```

The actual implementation will need to deal with situations where the combination of center and width puts the edges of the cropped areas outside of the array, but as noted earlier, we will be skipping complications that obscure the larger intent of the function. The full implementation can be found in the GitHub repository.¹⁰³

10.4 A straightforward Dataset implementation

We first saw PyTorch `Dataset` instances in chapter 7, 7.1, but this will be the first time we've implemented one ourselves. By subclassing `Dataset` we will take our arbitrary data and plug it into the rest of the PyTorch ecosystem. Each `ct` instance represents hundreds of different samples that we can use for training our model or validating its effectiveness. Our `LunaDataset` class will normalize those samples, flattening each CT's nodules into a single collection from which samples can be retrieved without regard for which `ct` instance the sample originates from. This flattening is often how we want to process our data, though as we'll see in chapter 12, 12.3.1, there will be situations where a simple flattening of the data won't be enough to train our model well.

In terms of implementation, we are going to start with the requirements imposed from subclassing `Dataset` and work backwards. This is a difference from the Datasets that we've worked with earlier; there we were using classes that were provided by external libraries, while here we need to implement and instantiate the class ourselves. Once we have done so, we can use it similarly to those earlier examples. Luckily, the implementation of our custom subclass will not be too difficult, as the PyTorch API requires only two functions to be provided by any `Dataset` subclasses you might want to implement:

1. We provide an implementation of `{uu}len{uu}` that must return a single, constant value after initialization (the value ends up being cached in some use cases).
2. We provide the `{uu}getitem{uu}` method, which takes an index and returns a tuple with sample data to be used for training (or validation, as the case may be).

First, let's just see what the function signatures and return values of those functions look like:

Listing 10.14 p2ch10/dsets.py, line 156: class LunaDataset.{uu}len{uu}

```
def __len__(self):
    return len(self.noduleInfo_list)

def __getitem__(self, ndx):
    # ... line 180
    return nodule_t, malignant_t, nodule_tup.series_uid, torch.tensor(center_irc)
```

Our `{uu}len{uu}` implementation is straightforward: we have a list of nodules; each nodule is a sample; our data set is as large as the number of samples we have. Now, we don't have to make the implementation as simple as we've seen here. In fact, in later chapters we'll see this change!¹⁰⁴ The only rule is that if `{uu}len{uu}` returns a value of `N`, then the `{uu}getitem{uu}` needs to return something valid for all inputs `0` to `N-1`.

For `{uu}getitem{uu}` we take an `ndx` (typically an integer, given the rule about supporting inputs `0` to `N-1`) and return the four-item sample tuple as depicted in 10.2. Building this tuple is a bit more complicated than getting the length of our data set, however, so let's take a look.

The first part of this method implies we need to construct `self.noduleInfo_list` as well as provide the `getCtRawNodule` function:

Listing 10.15 p2ch10/dsets.py, line 159: class LunaDataset.{uu}getitem{uu}

```
def __getitem__(self, ndx):
    nodule_tup = self.noduleInfo_list[ndx]
    width_irc = (32, 48, 48)

    nodule_a, center_irc = getCtRawNodule( ①
        nodule_tup.series_uid,
        nodule_tup.center_xyz,
        width_irc,
    )
```

- ① The return value `nodule_a` has shape (32, 48, 48); the axes are depth, height, and width.

We will get to those in a moment in 10.4.1 and 10.4.2.

The next thing we'll need to do in the `{uu}getitem{uu}` method is manipulating the data into the proper data types and required array dimensions as will be expected by downstream code:

Listing 10.16 p2ch10/dsets.py, line 169: class LunaDataset.{uu}getitem{uu}

```
nodule_t = torch.from_numpy(nodule_a)
nodule_t = nodule_t.to(torch.float32)
nodule_t = nodule_t.unsqueeze(0) ①
```

- ① The `.unsqueeze(0)` adds the 'Channel' dimension.

Don't worry too much about exactly why we are manipulating dimensionality for now; the next chapter will contain the code that will end up consuming this output, and will also impose the constraints we're proactively meeting here. This *will* be something that you should expect to for every custom `Dataset` you implement. These conversions are a key part of transforming your wild-west data into nice, orderly tensors.

Finally, we need to build our classification tensor:

Listing 10.17 p2ch10/dsets.py, line 173: class LunaDataset.{uu}getitem{uu}

```
malignant_t = torch.tensor([
    not nodule_tup.isMalignant_bool,
    nodule_tup.isMalignant_bool
],
    dtype=torch.long,
)
```

This has two elements; one each for our possible nodule classes (benign, or malignant). We could have a single output for malignancy, but `nn.CrossEntropyLoss` expects one output value per class, so that's what we provide here. The exact details of the tensors you construct here are

going to change based on the type of project you’re working on.

Let’s take a look at our final sample tuple:¹⁰⁵

Listing 10.18 p2ch10_explore_data.ipynb

```
# In[10]:  
LunaDataset()[0]  
  
# Out[10]:  
(tensor([[-899., -903., -825., ..., -901., -898., -893.],  
       ...,  
       [-92., -63., 4., ..., 63., 70., 52.]]]), ❶  
tensor([0, 1]), ❷  
'1.3.6.1.4.1.14519.5.2.1.6279.6001.287966244644280690737019247886', ❸  
IrcTuple(index=91.28, row=359.65, col=341.11)) ❹
```

- ❶ nodule_t
- ❷ cls_t
- ❸ nodule_tup.series_uid
- ❹ center_irc

Here we can see the four items we saw from our `{uu}getitem{uu}` return statement.

10.4.1 Caching nodule arrays with the `getCtRawNodule` function

In order to get decent performance out of our `LunaDataset`, we’re going to need to invest in some on-disk caching. This will allow us to avoid having to read an entire CT scan off of disk for every sample. Doing so would be prohibitively slow! Make sure you’re paying attention to bottlenecks in your project, and doing what you can to optimize them once they start slowing you down. We’re kind of jumping the gun here since we haven’t demonstrated to you, the reader, that we need the caching here. Without caching, the `LunaDataset` is easily 50x slower! We’ll revisit this in the exercises, 1.6.

The function itself is easy; it’s a file-cache-backed¹⁰⁶ wrapper around the `ct.getRawNodule` method we saw earlier:

Listing 10.19 p2ch10/dsets.py, line 121

```
@functools.lru_cache(1, typed=True)  
def getCt(series_uid):  
    return Ct(series_uid)  
  
@raw_cache.memoize(typed=True)  
def getCtRawNodule(series_uid, center_xyz, width_irc):  
    ct = getCt(series_uid)  
    ct_chunk, center_irc = ct.getRawNodule(center_xyz, width_irc)  
    return ct_chunk, center_irc
```

We use a few different caching methods here. First, we’re caching the `getCT` return value in

memory, so that we can repeatedly ask for the same CT instance without having to reload all of the data from disk. That's a huge speed increase in the case of repeated requests, but we're only keeping one CT in memory, so cache misses will be frequent if we're not careful about access order.

The `getCTRawNodule` function that calls `getCT` also has its outputs cached, however, so after our cache is populated `getCT` won't ever get called. These values get cached to disk using the python library `diskcache`. We'll discuss why we have this specific caching setup in the next chapter. For now, it's enough to know that it's much, much faster to read in 2^{15} `float32` values from disk than it is to read in 2^{25} `int16` values, convert to `float32`, and then select a 2^{15} subset. From the second pass through the data forward, I/O times for input should drop to insignificance.

NOTE

If the definition of these functions ever materially change, we will need to remove the cached values from disk. If we don't, the cache will continue to return them, even if now the function will not map the given inputs to the old output. The data is stored in the `data-unversioned/cache` directory.

10.4.2 Constructing our dataset in `LunaDataset.{uu}init{uu}`

Just about every project will need to separate samples into a training set and a validation set. We are going to do that here by taking every 10th sample, specified by the `val_stride` parameter, and designating that sample as a member of the validation set. We are also going to accept a `isValSet_bool` parameter, and use that to determine if we should be keeping only the training data, the validation data, or everything.

Listing 10.20 p2ch10/dsets.py, line 131: class `LunaDataset`

```
class LunaDataset(Dataset):
    def __init__(self,
                 val_stride=0,
                 isValSet_bool=None,
                 series_uid=None,
                 ):
        self.noduleInfo_list = copy.copy(getNoduleInfoList()) ❶

        if series_uid:
            self.noduleInfo_list = [x for x in self.noduleInfo_list if x.series_uid == series_uid]
```

- ❶ We copy the return value so that the cached copy won't be impacted by altering `self.noduleInfo_list`.

If we pass in a truthy `series_uid` then the instance will only have nodules from that series. This can be useful for visualization or debugging, by making it easier to look at e.g. a single problematic CT scan.

10.4.3 A Training / Validation Split

We allow for the `Dataset` to partition out $1/N^{\text{th}}$ of the data into a subset used for validating the model. How we will handle that subset is based on the value of the `isValSet_bool` argument.

Listing 10.21 p2ch10/dsets.py, line 142: class LunaDataset.{uu}init{uu}

```
if isValSet_bool:
    assert val_stride > 0, val_stride
    self.noduleInfo_list = self.noduleInfo_list[::-val_stride]
    assert self.noduleInfo_list
elif val_stride > 0:
    del self.noduleInfo_list[::-val_stride]
    assert self.noduleInfo_list
```

This means that we can create two `Dataset` instances and be confident that there is a strict segregation between our training data and our validation data. Of course, this depends on there being a consistent sorted order to `self.noduleInfo_list`, which we ensure by having there be a stable sorted order to the nodule info tuples, and by the `getNoduleInfoList` function sorting the list before returning it.

Let's take a look at the data using `p2ch10_explore_data.ipynb`:

```
# In[2]:
from p2ch10.dsets import getNoduleInfoList, getCt, LunaDataset
noduleInfo_list = getNoduleInfoList(requireDataOnDisk_bool=False)
malignantInfo_list = [x for x in noduleInfo_list if x[0]]
diameter_list = [x[1] for x in malignantInfo_list]

# In[4]:
for i in range(0, len(diameter_list), 100):
    print('{:4} {:4.1f} mm'.format(i, diameter_list[i]))

# Out[4]:
 0  32.3 mm
100 17.7 mm
200 13.0 mm
300 10.0 mm
400  8.2 mm
500  7.0 mm
600  6.3 mm
700  5.7 mm
800  5.1 mm
900  4.7 mm
1000 4.0 mm
1100 0.0 mm
1200 0.0 mm
1300 0.0 mm
```

We have a few very large nodules, starting at 32 mm, but rapidly dropping off to half that size. The bulk of the nodules are in the 4 to 10 mm range, and several hundred don't have size information at all. This looks as expected; you might recall we had more malignant nodules than we had diameter annotations. Quick sanity checks on your data can be very helpful. Catching a problem or mistaken assumption early will likely save hours of effort!

The larger takeaway is that our training and validation splits should have a few properties in

order to work well:

- Both sets should include examples of all variations of expected inputs.
- Neither set should have samples that aren't representative of expected inputs *unless* they have a specific purpose like training the model to be robust to outliers.
- The training set shouldn't offer unfair hints about the validation set that wouldn't be true for real-world data (for example, including the same sample in both sets; this is known as a *leak* in your training set).

10.4.4 Rendering the data

Again, either use `p2ch10_explore_data.ipynb` directly, or start Jupyter Notebook and enter:

```
# In[7]:  
%matplotlib inline ❶  
from p2ch10.vis import findMalignantSamples, showNodule  
malignantSample_list = findMalignantSamples()
```

- ❶ This magic line sets up the ability for images to be displayed inline via the Notebook, see: ipython.readthedocs.io/en/stable/interactive/plotting.html#id1

```
# In[8]:  
series_uid = malignantSample_list[11][2]  
showNodule(series_uid)
```

This will produce images akin to those showing CT and nodule slices earlier in this chapter.

Interested readers are invited to edit the implementation of the rendering code in `p2ch10/vis.py` to match their needs and tastes. The rendering code makes heavy use of `matplotlib`¹⁰⁷ which is too complex of a library to attempt to cover here.

Remember that rendering your data is not just about getting nifty looking pictures. The point is to be able to get an intuitive sense of what your inputs look like. Being able to tell at a glance "oh, this problematic sample is very noisy compared to the rest of my data" or "that's odd, this looks pretty normal" can be useful when investigating issues. It also helps foster insights like "perhaps if I modified things like so, I can solve the issue I'm having." That level of familiarity will be needed as you start tackling harder and harder projects.

NOTE

Due to the way that each subset has been partitioned, combined with the sorting used when constructing `LunaDataset.noduleInfo_list`, the ordering of the entries in the `malignantSample_list` is highly dependant on which subsets are present at the time the code is executed. Please remember this when trying to find a particular sample a second time, especially after decompressing more subsets.

10.5 Conclusion

In the last chapter we go our heads wrapped around our data. In this chapter we got *PyTorch*'s head wrapped around our data! By transforming our DICOM-via-Meta-Image raw data into tensors, we've set the stage to start implementing a model and training loop, which we'll see in the next chapter.

It's important to not underestimate the impact of the design decisions we've already made. The size of our inputs, the structure of our caching, and how we're partitioning our training and validation sets will all make a difference to the success or failure of our overall project.

Don't hesitate to revisit these decisions later on, especially once you're working on your own projects.

10.6 Exercises

1. Implement a program that iterates through a `LunaDataset` instance and time how long it takes to do so. In the interests of time, it might make sense to have an option to limit the iteration to the first $N=1000$ samples.
 - A. How long does it take to run the first time?
 - B. How long does it take to run the second time?
 - C. What does clearing the cache do to the run time?
 - D. What does using the *last* $N=1000$ samples do to the first/second runtime?
2. Change the `LunaDataset` implementation to randomize the sample list during `{uu}init{uu}`. Clear the cache and run the modified version.
 - A. What does that do to the run time of the first and second runs?
3. Revert the randomization and comment out the `@functools.lru_cache(1, typed=True)` decorator to `getct`. Clear the cache and run the modified version.
 - A. How does the runtime change now?

10.7 Summary

- Often the code required to parse and load raw data is non-trivial. For this project, we implement a `ct` class that will load data from disk and provide access to cropped regions around points of interest.
- Caching can be useful if the parsing and loading routines are expensive. Keep in mind that some caching can be done in-memory, and some is best on-disk. Each can have their place in a data loading pipeline.
- PyTorch `Dataset` subclasses are used to convert data from its native form into tensors suitable to pass into the model. We can use this to integrate our real-world data with PyTorch APIs.
- Subclasses of `Dataset` need to provide an implementation for two methods, `{uu}len{uu}` and `{uu}getitem{uu}`. Other helper methods are allowed, but not required.
- Splitting our data into a sensible training and validation set requires that we make sure no sample is in both sets. We accomplish this here by using a consistent sort order, and taking every 10th sample for our validation set.
- Data visualization is important; being able to investigate data visually can provide important clues about errors or problems. We are using Jupyter Notebooks and Matplotlib to render our data.

Training A Classification Model To Detect Suspected Tumors



This chapter covers:

- Using `DataLoader`s to load data
- Implementing a model that performs classification on our CT data from chapter 10
- Setting up the basic skeleton for our training and validation application
- Logging and displaying metrics to evaluate the model's performance

In the previous chapters we set the stage for our cancer detection project. We covered medical details of lung cancer, took a look at the main data sources we will use for our project, and transformed our raw CT scans into a PyTorch `Dataset`. Now that we have a `Dataset`, we can easily consume our training data.

So let's do that!

We're going to do two main things in this chapter. We're going to start with building the classification model and training loop that will be the foundation that the rest of Part 2 uses to explore the larger project. To do that, we'll take the `ct` and `LunaDataset` classes we implemented last chapter, and use them to feed `DataLoader` instances. Those instances, in turn, feed our classification model with data via training and validation loops.

We'll finish out the chapter by using the results from running that training loop to introduce one of the hardest challenges of Part 2, which is how to get high-quality results from messy, limited data. In later chapters we'll be exploring the specific ways that our data is limited, as well as mitigating those limitations.

Let's recall our high-level roadmap from chapter 9, shown here in 11.1. Right now, we are going to be working on producing a model capable of step 4, classification. As a reminder, that means we're going to be assigning a single, specific label to each sample that we present to the model.

In this case, those labels are "benign" and "malignant," since each sample represents a single nodule.

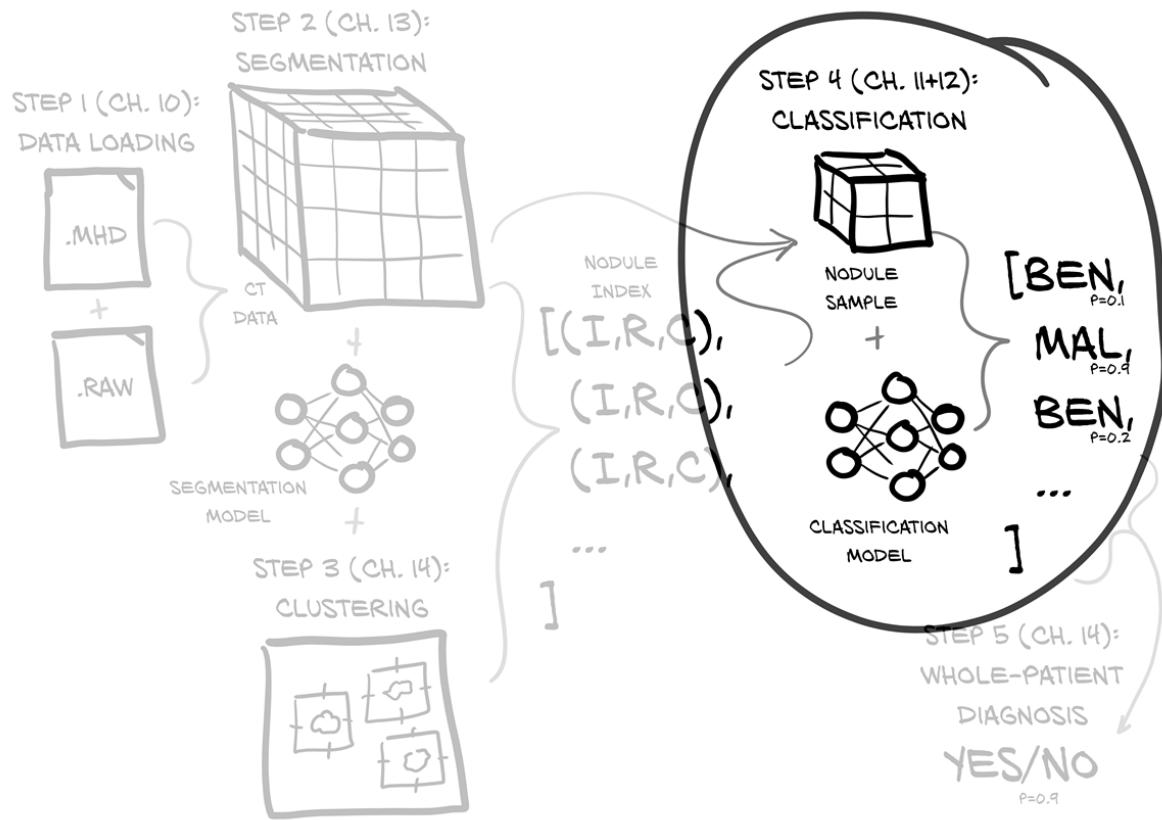


Figure 11.1 Our end-to-end lung cancer detection project, with a focus on this chapter's topic; step 4, classification.

Getting an early end-to-end version of a meaningful part of your project is a great milestone to reach. Having something that works well enough for the results to be evaluated analytically lets you move forward with future changes, confident that you are improving your results with each change — or at least that you're able to set aside those changes and experiments that don't work out! Expect to have to do a lot of experimentation when working on your own projects. Getting the best results will usually require a lot of tinkering and tweaking.

But before we can get to the experimental phase, we must lay our foundation. Let's see what our Part 2 training loop looks like in 11.2, which should seem generally familiar, given that we've seen similar in chapter 5, 5.1.7. Here we will also be using a validation set, as discussed in 5.2.2.

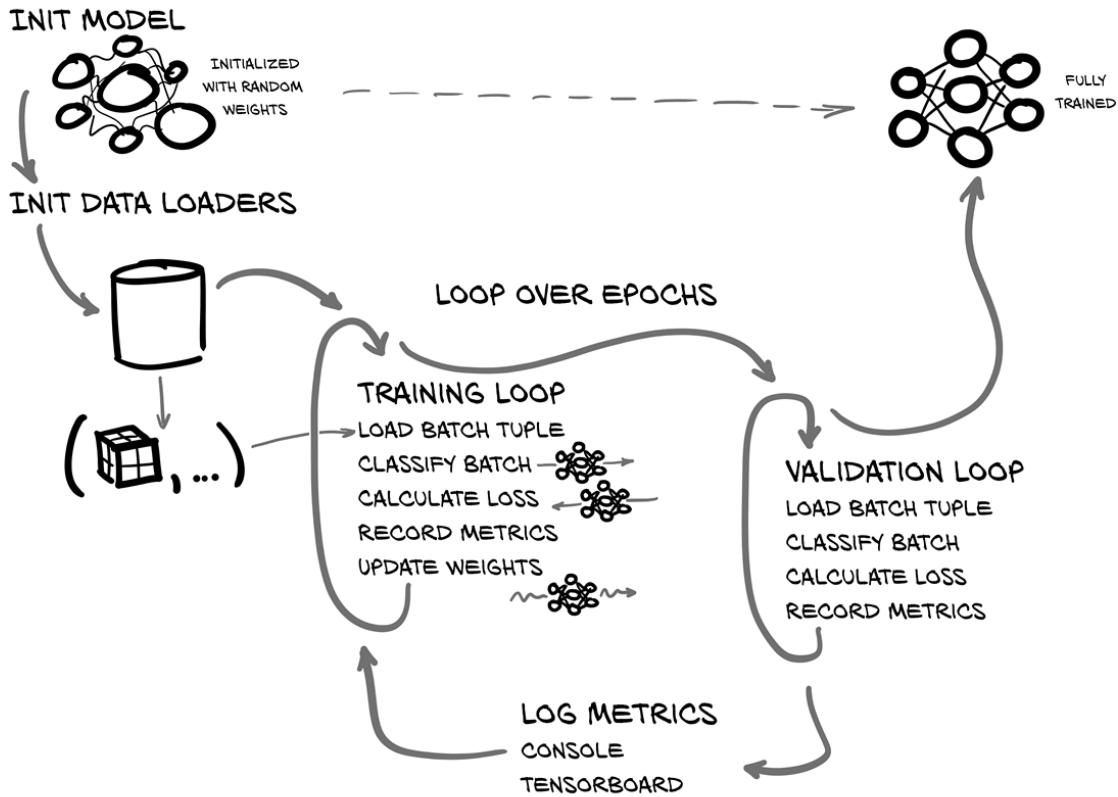


Figure 11.2 The training and validation script we will implement in this chapter.

The basic structure of what we're going to implement will be:

- Initialize our model and data loading
- Loop over a semi-arbitrarily chosen number of epochs
 - Loop over each batch of training data returned by our `LunaDataset`
 - The data loader worker process will have loaded the relevant batch of data in the background
 - Pass the batch into our classification model to get results
 - Calculate our loss based on the difference between our predicted results and our ground truth data
 - Record metrics about our model's performance into a temporary data structure
 - Update the model weights via backpropagation of the error
 - Loop over each batch of validation data (in a very similar manner to the training loop)
 - Load the relevant batch of validation data (again, in the background worker process)
 - Classify the batch and compute the loss
 - Record information about how well the model performed on the validation data
 - Print out progress and performance information for this epoch

As we go through the code for the chapter, keep an eye out for two main differences between the code we're producing here and what we used for a training loop in Part 1. We're going to want to put some more structure around our program, since the project as a whole is going to be quite a bit more complicated than what we did in earlier chapters. Without that extra structure, the code

can get messy quickly. For this project, we are going to have our main training application use a number of well-contained functions, and we will further separate code for things like our dataset into self-contained python modules. Make sure that for your own projects that you're matching the level of structure and design to the complexity level of your project. Too little structure and it will become difficult to perform experiments cleanly, troubleshoot problems, or even describe what you're doing! Conversely, too *much* structure means that you're wasting time writing infrastructure that you don't need, and most likely slowing yourself down by having to conform to it even after all that plumbing is in place. Plus it can be tempting to spend time on infrastructure as a procrastination tactic, rather than digging into the hard work of making actual progress on your project. Don't fall into that trap!

The other big difference will be a focus on collecting a variety of metrics about how training is progressing. Being able to accurately tell what the impact of making changes to training is impossible without having good metrics logging. Without spoiling the next chapter, we'll also see how important it is to be collecting not just metrics, but the *right metrics for the job*. We'll lay the infrastructure for tracking those metrics in this chapter, and exercise that infrastructure by collecting and displaying the loss and percent of samples correctly classified, both overall, and per-class. That's enough to get us started for now, but we'll cover a more realistic set of metrics in chapter 12.

11.1 The main entrypoint for our application

One of the big structural differences from earlier training work we've done in this book is that part 2 is going to wrap up our work in a fully-fledged command-line application. It will parse command-line arguments, have a fully-featured `--help` command, and be easily runnable in a wide variety of environments. All this will allow us to easily invoke the training routines from both Jupyter and a Bash shell.¹⁰⁸

Our application's functionality will be implemented via a class, so that we can instantiate the application and pass it around should we feel the need. This can make testing, debugging, or invocation from other python programs easier. We can invoke the application without needing to spin up a second os-level process (we won't do explicit unit testing in this book, but the structure we create by doing this can be helpful for real projects where that kind of testing is appropriate).

One way to take advantage of being able to invoke our training by either function call or OS-level process is by wrapping the function invocations up into a Jupyter Notebook, so that the code can easily be called from either the native CLI or the browser:

Listing 11.1 code/p2_run_everything.ipynb

```
# In[2]:  
def run(app, *argv):  
    argv.insert(0, '--num-workers=6')      ①  
    log.info("Running: {}({!r}).main()".format(app, argv))  
  
    app_cls = importstr(*app.rsplit('.', 1))  ②  
    app_cls(argv).main()  
  
    log.info("Finished: {}({!r}).main()".format(app, arg_list))  
  
# In[6]:  
run('p2ch11.training.LunaTrainingApp', '--epochs=1')
```

- ① We assume you have a 4-core, 8-thread CPU. Change the 6 if needed.
- ② This is a slightly cleaner call to `{uu}import{uu}`.

Let's get some semi-standard boilerplate code out of the way. We'll start at the end of the file with a pretty standard "if main" stanza that instantiates the application object and invokes the `main` method:

Listing 11.2 p2ch11/training.py, line 370

```
if __name__ == '__main__':  
    LunaTrainingApp().main()
```

From there, we can jump back to the top of the file and have a look at the application class and the two functions we just called, `{uu}init{uu}` and `main`. We're going to want to be able to accept command line arguments, so we'll be using the standard `argparse` library¹⁰⁹ in the application's `{uu}init{uu}` function. Note that we can pass in custom arguments to the initializer, should we wish to do so. The `main` method will be the primary entry point for the core logic of the application.

Listing 11.3 p2ch11/training.py, line 31: class LunaTrainingApp

```
class LunaTrainingApp:  
    def __init__(self, sys_argv=None):  
        if sys_argv is None: ①  
            sys_argv = sys.argv[1:]  
  
        parser = argparse.ArgumentParser()  
        parser.add_argument('--num-workers',  
                           help='Number of worker processes for background data loading',  
                           default=8,  
                           type=int,  
                           )  
        # ... line 63  
        self.cli_args = parser.parse_args(sys_argv)  
        self.time_str = datetime.datetime.now().strftime('%Y-%m-%d_%H.%M.%S') ②  
  
        # ... line 127  
    def main(self):  
        log.info("Starting {}, {}".format(type(self).__name__, self.cli_args))
```

- ① If the caller doesn't provide arguments, we get them from the command line.
- ② We'll use the timestamp to help identify training runs.

This structure is pretty general, and could be reused for future projects. In particular, parsing arguments in `{uu}init{uu}` allows for the application to be configured separately from it being invoked.

If you check the code for this chapter on GitHub, you might notice some extra lines mentioning "TensorBoard." Ignore those for now; we'll discuss those in detail later in the chapter, in 11.8.

11.2 Pre-training setup and initialization

Before we can begin iterating over each batch in our epoch, there's some initialization work that needs to happen. After all, we can't train a model if we haven't even instantiated one yet! There are two main things we need to do, as we can see in 11.3. The first, as we just mentioned, is initialize our model and optimizer, and the second is to initialize our `Dataset` and `DataLoader` instances.

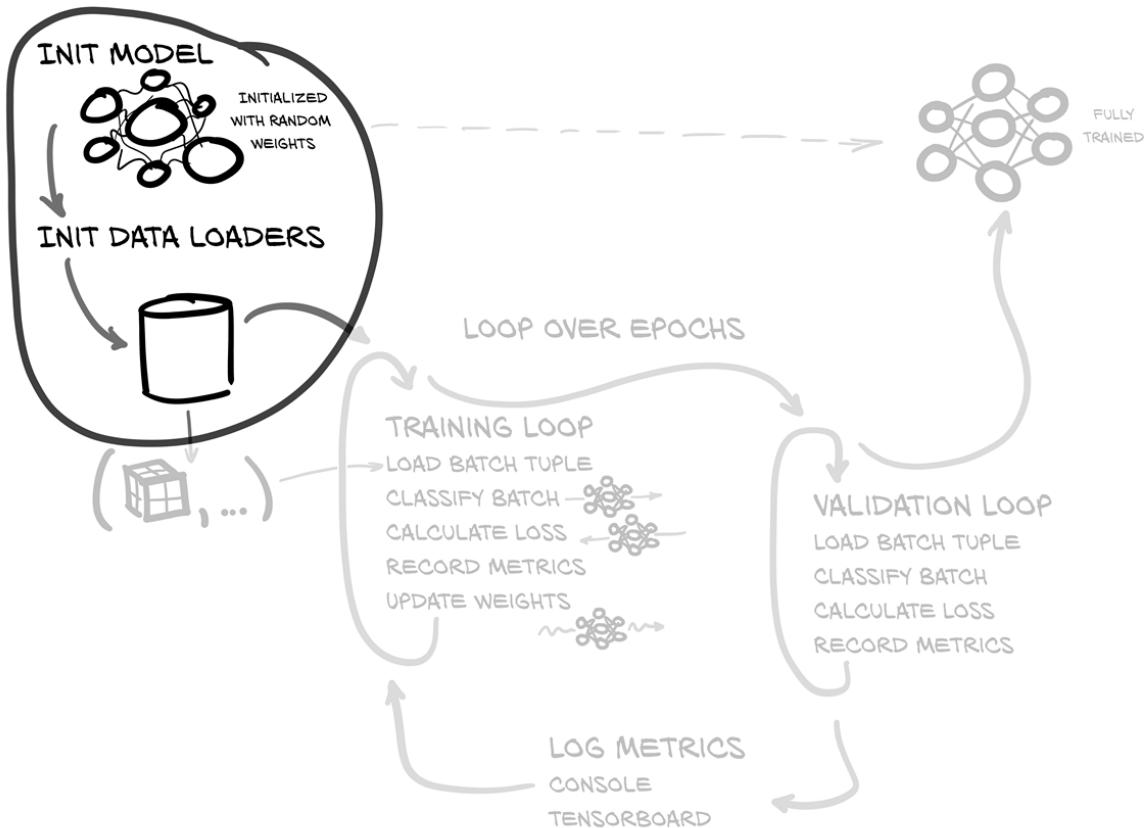


Figure 11.3 The training and validation script we will implement in this chapter, with a focus on the pre-loop variable initialization.

Our `LunaDataset` will define the randomized set of samples that will make up our training epoch, and our `DataLoader` instance will actually perform the work of loading the data out of

our data set and providing it to our application.

11.2.1 Initializing the model and optimizer

For this section, we are treating the details of the `LunaModel` as a black box. In the next section (11.3) we will detail the internal workings. Readers are welcome to explore changes to the implementation to better meet our goals for the model, though that's probably best done after finishing at least chapter 12, 12.

Let's see what our starting point is going to look like:

Listing 11.4 p2ch11/training.py, line 31: class LunaTrainingApp

```
class LunaTrainingApp:  
    def __init__(self, sys_argv=None):  
        # ... line 70  
        self.use_cuda = torch.cuda.is_available()  
        self.device = torch.device("cuda" if self.use_cuda else "cpu")  
  
        self.model = self.initModel()  
        self.optimizer = self.initOptimizer()  
  
    def initModel(self):  
        model = LunaModel()  
        if self.use_cuda:  
            log.info("Using CUDA with {} devices.".format(torch.cuda.device_count()))  
            if torch.cuda.device_count() > 1: ①  
                model = nn.DataParallel(model) ②  
            model = model.to(self.device) ③  
        return model  
  
    def initOptimizer(self):  
        return SGD(self.model.parameters(), lr=0.001, momentum=0.99)
```

- ① Detect multiple GPUs.
- ② Wrap the model.
- ③ Send model parameters to the GPU.

If the system used for training has more than one GPU, we will use the `nn.DataParallel` class to distribute the work between all of the GPUs in the system, and then collect and resync parameter updates, etc. This is almost entirely transparent in terms of both the model implementation and the code that uses that model¹¹⁰.

SIDE BAR **DataParallel vs. DistributedDataParallel**

For this book, we will be using `DataParallel` to handle utilizing multiple GPUs. We chose `DataParallel` because it's a simple drop-in wrapper around our existing models. It is not the best performing solution to using multiple GPUs, however, and it is limited to working with the hardware available in a single machine.

PyTorch also provides `DistributedDataParallel`, which is the recommended wrapper class to use when you need to spread work between more than a single GPU or single machine. Since proper setup and configuration is non-trivial, and we suspect that the vast majority of our readers won't see any benefit from the complication, we won't be covering `DistributedDataParallel` in this text. For readers wishing to learn more, we suggest reading the official documentation.¹¹¹

Assuming that `self.use_cuda` is true, the call `self.model.to(device)` moves the model parameters to the GPU, setting up the various convolutions and other calculations to use the GPU for the heavy numerical lifting. It's important to do so before constructing the optimizer, since otherwise the optimizer would be left looking at the CPU-based parameter objects, not the copied-to-the-GPU ones.

For our optimizer, we are going to use basic stochastic gradient descent¹¹² with momentum. We first saw this optimizer in chapter 5, 5.2.1. As we recall from part 1, there are many different optimizers available in PyTorch; while we won't cover most of them in any detail, the official documentation¹¹³ does a good job of linking to the relevant papers describing each of them.

Using stochastic gradient descent is generally considered a safe place to start when it comes to picking an optimizer; there are some problems that might not work well with SGD, but they're relatively rare. Similarly, picking a learning rate of `0.01` and a momentum of `0.9` are pretty safe choices. Empirically, SGD with those values has worked reasonably well for a wide range of projects, and it's easy to try a learning rate of `0.1` or `0.001` if things aren't working well right out of the box.

That's not to say any of those values are the best for our use case, but trying to find better ones is getting ahead of ourselves somewhat. Systematically trying different values for learning rate, momentum, network size, and other similar configuration settings is called a *hyperparameter search*,

There are other, more glaring issues we'll need to address first in the coming chapters. Once those get addressed, we can begin to do the work to fine-tune these values. There are also other,

more exotic optimizers that we might choose as well, but aside from perhaps swapping `torch.optim.SGD` for `torch.optim.Adam`, understanding the tradeoffs that you're making with those other optimizers is a topic too advanced for this book.

11.2.2 Care and feeding of DataLoaders

The `LunaDataset` class that we built last chapter in the 10.4 section acts as the bridge between whatever wild west data we have, and the somewhat more structured world of tensors that the PyTorch building blocks expect. For example, `torch.nn.Conv3d`¹¹⁴ expects 5-dimensional input: (N, C, D, H, W) ¹¹⁵ — quite different from the native 3D our CT provides!

You may recall the `ct_t.unsqueeze(0)` call in `LunaDataset.{uu}getitem{uu}` from last chapter; that provided the 4th dimension, a "channel" for our data. As we recall from chapter 4, 4, an RGB image would have three channels, one for each of red, green, and blue. Astronomical data could have dozens, each for various slices of the electromagnetic spectrum — gamma rays, x-rays, ultraviolet light, visible light, infrared, microwaves, and/or radio waves. Since CT scans are single-intensity, our channel dimension is only size one (for now!).

We also recall from Part 1 that training on single samples at a time is typically an inefficient use of computing resources, due to most processing platforms being capable of more parallel calculations than are required by a model to process a single training or validation sample. The solution is to group sample tuples together into a batch tuple, like in 11.4, allowing multiple samples to be processed at the same time. The 5th dimension, `N` above, is there to differentiate multiple samples in the same batch.

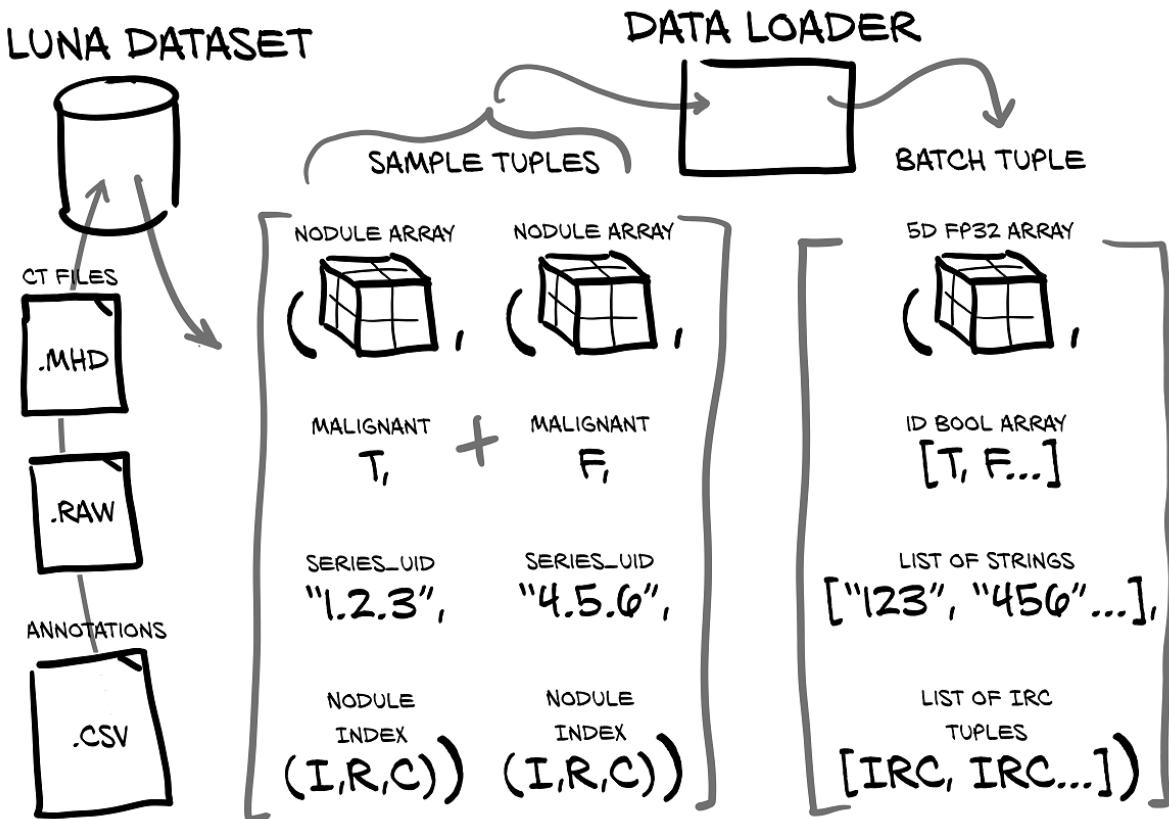


Figure 11.4 Sample tuples being collated into a single batch tuple inside of a DataLoader.

Conveniently, we don't have to implement any of this batching; The PyTorch `DataLoader` class will handle all of the collation work for us. We've already built the bridge from the wild west CT scans to PyTorch tensors with our `LunaDataset` class, so all that remains is to plug our `Dataset` into a `DataLoader`:

Listing 11.5 p2ch11/training.py, line 89: class LunaTrainingApp.initTrainDL

```
def initTrainDL(self):
    train_ds = LunaDataset(
        val_stride=10,
        isValSet_bool=False,
    )

    train_dl = DataLoader(
        train_ds,
        batch_size=self.cli_args.batch_size * (torch.cuda.device_count() if self.use_cuda else 1),
        num_workers=self.cli_args.num_workers,
        pin_memory=self.use_cuda,
    )

    return train_dl

# ... line 127
def main(self):
    train_dl = self.initTrainDL()
    val_dl = self.initValDL()
```

In addition to taking individual samples and batching them, `DataLoaders` can also provide

parallel loading of data by using separate processes and shared memory. All we need to do is specify `num_workers=...` when instantiating the `DataLoader` and the rest is taken care of behind the scenes. Each worker process produces complete batches like in 11.4. This helps make sure that hungry GPUs are kept well-fed with data. Our `validation_ds` and `validation_dl` instances look similar, except for the obvious `isValSet_bool=True`.

When we iterate, like `for batch_tup in self.train_dl:` we won't have to wait for each `ct` to be loaded, samples taken, the samples batched, etc. Instead, we'll get the already-loaded `batch_tup` immediately, and in the background a worker process will get freed up to begin loading another batch to use on a later iteration. Using the data loading features of PyTorch can help speed up most projects, due to being able to overlap data loading and processing with GPU calculation.

11.3 Our first-pass neural network design

The possible design space of a convolutional neural network capable of detecting tumors is effectively infinite. Luckily, considerable effort has been spent over the past decade or so investigating effective models for image recognition. While these largely focused on 2-dimensional images, the general architecture ideas transfer well to 3D, so there are many tested designs that we can use as a starting point. This helps because, while our first network architecture is unlikely to be our best option, right now we are only aiming for "good enough to get us going."

We are going to be basing the network design off of what we used in chapter 8, 6.3. We will have to update the model somewhat due to our input data being three dimensional, and we will add some complicating details, but the overall structure we see in 11.5 should feel familiar. Similarly, the work we do for this project will be a good base for your future projects, though the further you get from classification or segmentation projects, the more you'll have to adapt this base to fit.

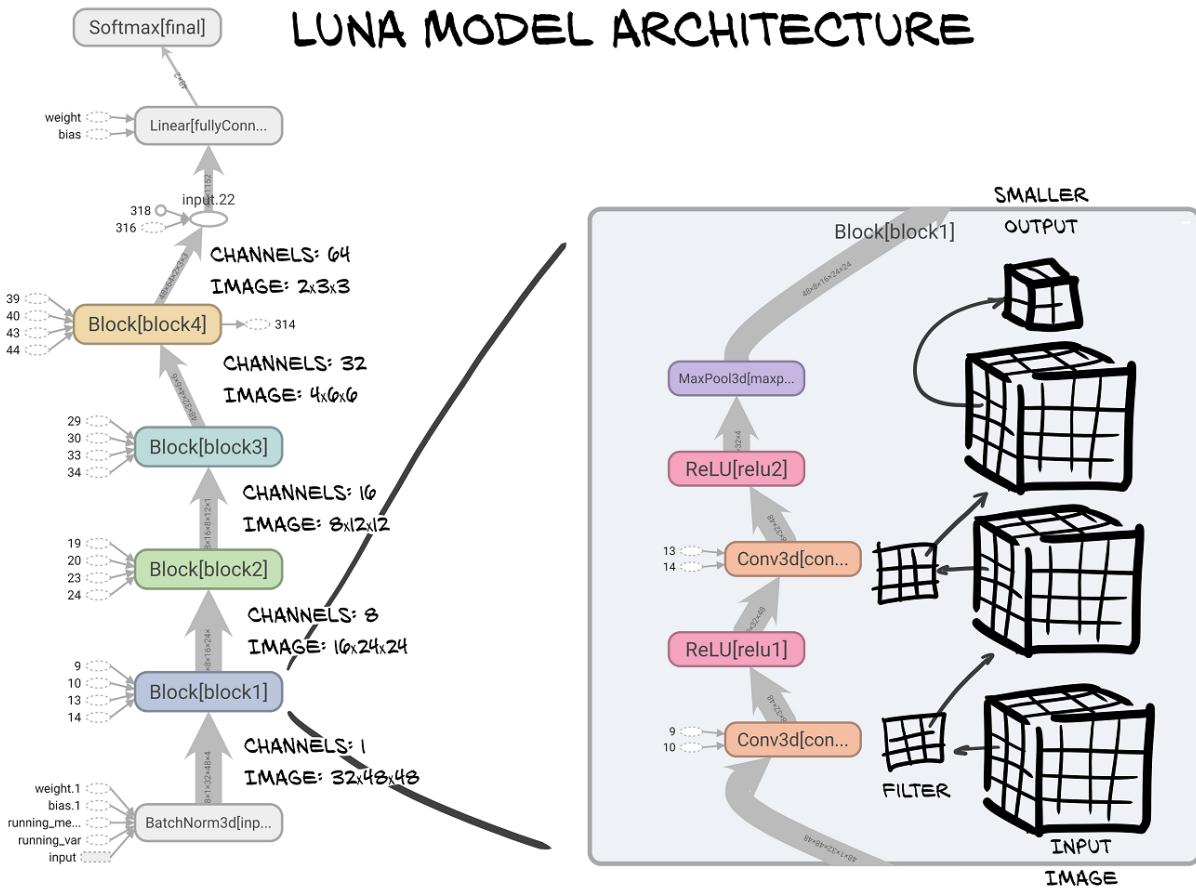


Figure 11.5 The architecture of the `LunaModel` class, consisting of a tail of batch normalization, a 4-block body, and a head of a linear layer followed by softmax.

Let's dissect this architecture, starting with the four repeated blocks that make up the bulk of the network.

11.3.1 The Core Convolutions

Classification models often have a structure that consists of a tail, backbone (or body), and head. The *tail* is the first few layers that process the input to the network. These early layers often have a different structure or organization from the rest of the network, as they must adapt the input to the form expected by the backbone. The *backbone* of the network typically contains the bulk of the layers, which are usually arranged in series of *blocks*. Each block will have the same (or at least similar) set of layers, though often the size of the expected input and the number of filters will change from block to block. Finally, the *head* of the network takes the output from the backbone and converts it into the desired output form. For convolutional networks, this often involves flattening the intermediate output, and passing it to a fully connected layer.

We will be using a block that consists of two 3x3 convolutions, each followed by an activation, with a max pooling operation at the end of the block. Using this structure can be a good first building block for a convolutional network. There are more complicated designs out there, but for many projects they're overkill, both in terms of implementation complexity, and in terms of

computational demands. It's a good idea to start simple and only add complexity when there's a demonstrable need for it.

Listing 11.6 p2ch11/model.py, line 63: class LunaBlock

```
class LunaBlock(nn.Module):
    def __init__(self, in_channels, conv_channels):
        super().__init__()

        self.conv1 = nn.Conv3d(in_channels, conv_channels, kernel_size=3, padding=1, bias=True)
        self.relu1 = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv3d(conv_channels, conv_channels, kernel_size=3, padding=1, bias=True)
        self.relu2 = nn.ReLU(inplace=True)

        self.maxpool = nn.MaxPool3d(2, 2)

    def forward(self, input_batch):
        block_out = self.conv1(input_batch)
        block_out = self.relu1(block_out)
        block_out = self.conv2(block_out)
        block_out = self.relu2(block_out)

        return self.maxpool(block_out)
```

Since we're using two 3x3x3 convolutions stacked back-to-back, the final output of the block has an effective field of 5x5x5, but uses fewer parameters than a true 5x5x5 would. This gets fed into a 2x2x2 max pool, which means that we're taking a 6x6x6 effective field and throwing away 7/8ths of the data and going with the one 5x5x5 field that produced the largest value. Now, those "discarded" input voxels still have a chance to contribute, since the next max pool over has an overlapping input field, and so it's possible that they'll be included that way.

The `nn.ReLU` layers are the same as the ones we looked at in chapter 6, 6.1.1.

This block will be repeated multiple times to form our model's backbone.

11.3.2 The Full Model

Let's take a look at the full model implementation. We'll skip the block definition, since we just saw that earlier.

Listing 11.7 p2ch11/model.py, line 13: class LunaModel

```
class LunaModel(nn.Module):
    def __init__(self, in_channels=1, conv_channels=8):
        super().__init__()

        self.tail_batchnorm = nn.BatchNorm3d(1)      ①

        self.block1 = LunaBlock(in_channels, conv_channels)  ②
        self.block2 = LunaBlock(conv_channels, conv_channels * 2)
        self.block3 = LunaBlock(conv_channels * 2, conv_channels * 4)
        self.block4 = LunaBlock(conv_channels * 4, conv_channels * 8)  ②

        self.head_linear = nn.Linear(1152, 2)      ③
        self.head_softmax = nn.Softmax(dim=1)      ③
```

- ① Tail.
- ② Backbone (including all code between).
- ③ Head.

Here, our tail is relatively simple. We are going to normalize our input using `nn.BatchNorm3d`, which, as we saw in chapter 8, 8.1.3, will shift and scale our input so that it has a mean of 0 and standard deviation of 1. This means that our somewhat odd HU scale that our input is in won't really be visible to the rest of the network. Doing so is a somewhat arbitrary choice; we know what the units of our input are, and know what the expected values of the relevant tissues are, so we could probably implement a fixed normalization scheme pretty easily. It's not clear which would be better¹¹⁶.

Our backbone is four repeated blocks, with the block implementation pulled out into the separate `nn.Module` subclass we saw earlier in 11.5. Since each block ends with a 2x2x2 max-pool operation, after 4 layers we will have decreased the resolution of the image 16x in each dimension. If you recall from last chapter, our data is being returned in chunks that are 32x48x48, which will become 2x3x3 by the end of the backbone.

Finally, our tail is just a fully connected layer followed by a call to `nn.Softmax`. Softmax is a useful function for single-label classification tasks that has a few nice properties. First, it bounds the output between zero and one. Second, it's relatively insensitive to the absolute range of the inputs; only the *relative* values of the inputs matter. Finally, it allows our model to express the degree of certainty it has in an answer.

The function itself is relatively simple. Every value from the input is used to exponentiate e , and the resulting series of values is then divided by the sum of all the results of exponentiation. Here's what it looks like implemented in a simple fashion:

Listing 11.8 Unoptimized softmax implementation in pure python.

```
>>> logits = [1, -2, 3]
>>> exp = [e ** x for x in logits]
>>> exp
[2.718, 0.135, 20.086]

>>> softmax = [x / sum(exp) for x in exp]
>>> softmax
[0.118, 0.006, 0.876]
```

Of course, we use the PyTorch version `nn.Softmax` for our model, as it natively understands batches, tensors, and will perform autograd quickly and as expected.

COMPLICATION: CONVERTING FROM CONVOLUTION TO LINEAR

Continuing on with our model definition, we come to a complication. We can't just feed the output of `self.block4` into a fully connected layer, since that output is a per-sample 2x3x3 image with 64 channels, and fully connected layers expect a 1D vector as input (well, technically it expects a *batch* of 1D vectors, which is a 2D array, but the mismatch remains either way). Let's take a look at the `forward` method:

Listing 11.9 p2ch11/model.py, line 46: class LunaModel.forward

```
def forward(self, input_batch):
    bn_output = self.tail_batchnorm(input_batch)

    block_out = self.block1(bn_output)
    block_out = self.block2(block_out)
    block_out = self.block3(block_out)
    block_out = self.block4(block_out)

    conv_flat = block_out.view(
        block_out.size(0),      ①
        -1,
    )
    linear_output = self.head_linear(conv_flat)

    return linear_output, self.head_softmax(linear_output)
```

- ① This is the batch size.

Note that before we pass data into fully connected layer, we first must flatten it using the `view` function. Since that operation is stateless (it has no parameters that govern its behavior), we can simply perform the operation in the `forward` function. This is somewhat similar to the functional interfaces we discussed in chapter 8, 6.3.1. Almost every model that uses convolution, and produces classifications, regressions, or other non-image outputs will have a similar component in the head of the network.

For the return value of the `forward` method, we return both the raw *logits*, as well as the softmax-produced probabilities. The logits are the numerical values produced by the network prior to being normalized into probabilities by the softmax layer. We'll use the logits when we calculate the `nn.CrossEntropyLoss` during training¹¹⁷, and the probabilities for when we want to actually classify the samples. This kind of slight difference between what's used for training and what's used in production is somewhat common, especially when the difference between the two outputs is a simple, stateless function like softmax.

INITIALIZATION

Finally, let's talk about the initialization of our network's parameters. In order to get well-behaved performance out of our model, the weights, biases, and other parameters of the network need to exhibit certain properties. Let's imagine a degenerate case, where all of the weights of the network are greater than 1. In that case, the repeated multiplication by those weights would result in output that tends toward infinity. Similarly, weights less than 1 would all have layer outputs tend toward zero.

There are many normalization techniques that can be used to keep layer outputs well-behaved, but one of the simplest is to just make sure that the weights of the network are initialized to reasonable values. Unfortunately, for historical reasons, PyTorch has default weight initializations that are not ideal. Work is in progress to fix the situation; progress can be tracked on GitHub.¹¹⁸

In the meantime, we need to fix weight initialization ourselves. We can just treat the following `_init_weights` function as boilerplate, as the exact details aren't particularly important:

Listing 11.10 p2ch11/model.py, line 30: class LunaModel._init_weights

```
def _init_weights(self):
    for m in self.modules():
        if type(m) in {
            nn.Linear,
            nn.Conv3d,
        }:
            nn.init.kaiming_normal_(m.weight.data, a=0, mode='fan_out', nonlinearity='relu')
            if m.bias is not None:
                fan_in, fan_out = nn.init._calculate_fan_in_and_fan_out(m.weight.data)
                bound = 1 / math.sqrt(fan_out)
                nn.init.normal_(m.bias, -bound, bound)
```

Once you've progressed to the point where the details about weight initialization are of specific interest to you, feel free to revisit this topic.¹¹⁹

11.4 Training and validating the model

Now it's time to take the various pieces we've been working with and assemble them into something we can actually execute. This training loop should be familiar; we've seen loops like 11.6 previously in chapter 5.

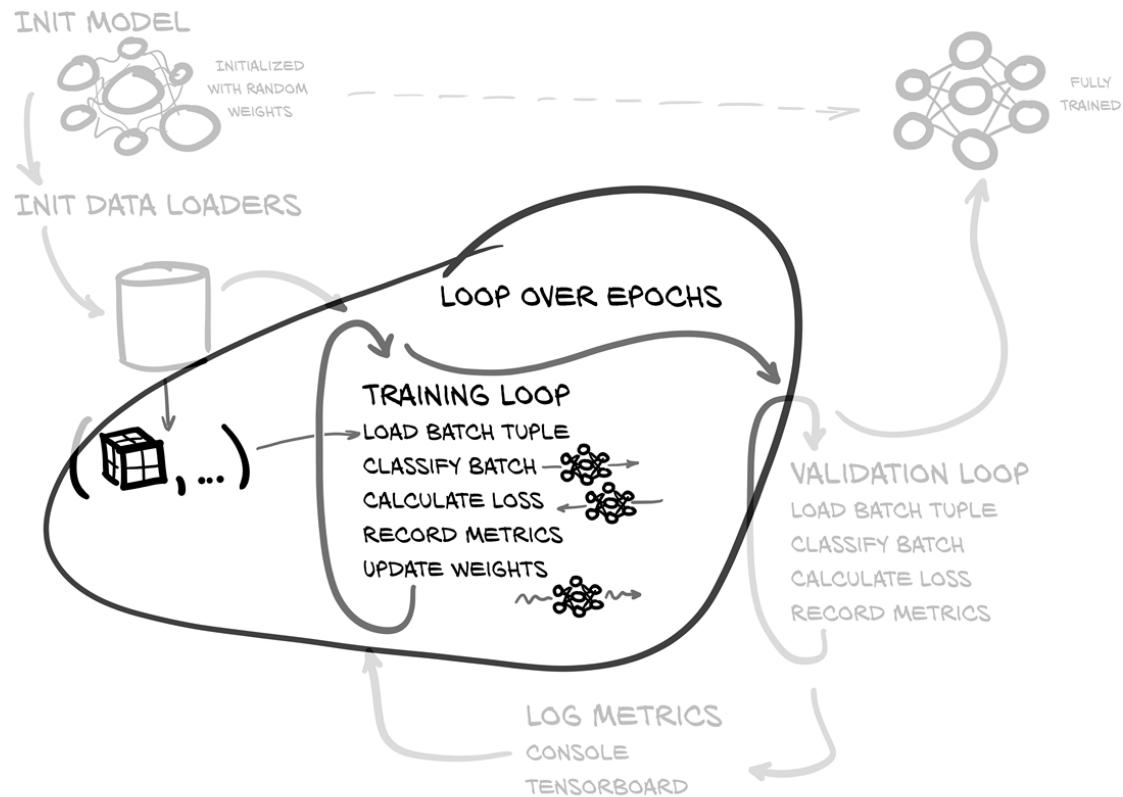


Figure 11.6 The training and validation script we will implement in this chapter, with a focus on the nested loops over each epoch and batches-in-the-epoch.

The code is relatively compact (the `doTraining` function is only 12 statements; it's longer here due to line length limitations):

Listing 11.11 p2ch11/training.py, line 127: class LunaTrainingApp.main

```
def main(self):
    # ... line 135
    for epoch_ndx in range(1, self.cli_args.epochs + 1):
        trnMetrics_t = self.doTraining(epoch_ndx, train_dl)
        self.logMetrics(epoch_ndx, 'trn', trnMetrics_t)

    # ... line 157
def doTraining(self, epoch_ndx, train_dl):
    self.model.train()
    trnMetrics_g = torch.zeros(METRICS_SIZE, len(train_dl.dataset)).to(self.device)
    batch_iter = enumerateWithEstimate(
        train_dl,
        "E{} Training".format(epoch_ndx),
        start_ndx=train_dl.num_workers,
    )
    for batch_ndx, batch_tup in batch_iter:
        self.optimizer.zero_grad()

        loss_var = self.computeBatchLoss(
            batch_ndx,
            batch_tup,
            train_dl.batch_size,
            trnMetrics_g
        )

        loss_var.backward()
        self.optimizer.step() ❶
        del loss_var ❷

    self.totalTrainingSamples_count += len(train_dl.dataset)

    return trnMetrics_g.to('cpu')
```

- ❶ This helps keep memory usage low; see 11.4.1

The main differences that we see from training loops in earlier chapters are:

1. The `trainingMetrics_t` tensor, which will be used to collect detailed per-class metrics during training. For larger projects like this, this kind of insight can be very nice to have.
2. We don't directly iterate over the `train_dl` Data Loader. This isn't crucial; it's just a stylistic choice.
3. The actual loss computation is pushed into a function. Again, this isn't strictly necessary, but code reuse is typically a plus.
4. We explicitly delete our loss variable at the end of the loop body.

We'll discuss why we've wrapped `enumerate` with additional functionality in 11.6.2; for now assume it's the same as `enumerate(train_dl)`.

The purpose of the `trainingMetrics_t` tensor is to transport information about how the model is behaving on a per-sample basis from the 11.4.2 function to the 11.5.1. Let's take a look at the first function, `computeBatchLoss` next. We'll cover `logMetrics` after we're done with the rest of the main training loop.

11.4.1 Deleting the loss variable

One interesting aspect of how PyTorch performs backpropagation is that a loss variable, shown here as `loss_var`, will contain references to the entire tree of gradients. If the relevant model parameters are stored on the GPU, then the gradients will be as well.

Let's take a simplified look at our `doTraining` loop:

Listing 11.12 p2ch11/training.py, line 165: class LunaTrainingApp.doTraining

```
for batch_ndx, batch_tup in batch_iter:  
    self.optimizer.zero_grad()  
  
    loss_var = self.computeBatchLoss(  
        batch_ndx,  
        batch_tup,  
        train_dl.batch_size,  
        trnMetrics_g  
    )  
  
    loss_var.backward()  
    self.optimizer.step()  
    del loss_var ①
```

- ① This is the key.

Due to the way that Python garbage collection works, having a loop variable like `loss_var` means that it won't be freed until it's either explicitly deleted (as we do here), or it's overwritten by the next iteration of the loop. If the `del` statement were removed from the above block, `loss_var` would continue to exist after the first loop body finished.

That means that without the `del`, by the second time the loop body executed we would still be holding onto the first loop's gradients while computing the second forward pass. Depending on exactly how much free RAM the GPU has, this might result in out-of-memory issues since the old reference wouldn't be able to be released that GPU memory to be reaped until the return value of `computeBatchLoss` has already been computed and about to be assigned to `loss_var`.

So instead we `del loss_var` at the end of the loop.

11.4.2 The `computeBatchLoss` function

The compute batch loss function is called by both the training and validation loops. As the name suggests, it computes the loss over a batch of samples. In addition, the function also computes and record per-sample information about the output the model is producing. This lets us compute things like what our percentage of correct answers is per-class, which allows us to hone in on areas where our model is having difficulty.

Of course, as the function name implies, the core functionality of the function is around feeding the batch into the model, and computing the per-batch loss. We're using `CrossEntropyLoss`¹²⁰

, just like in chapter 7, 7. Unpacking the batch tuple, moving the tensors to the GPU, and invoking the model should all feel familiar after the training work done in chapter 7.

Listing 11.13 p2ch11/training.py, line 207: class LunaTrainingApp.computeBatchLoss

```
def computeBatchLoss(self, batch_ndx, batch_tup, batch_size, metrics_g):
    input_t, label_t, _series_list, _center_list = batch_tup

    input_g = input_t.to(self.device, non_blocking=True)
    label_g = label_t.to(self.device, non_blocking=True)

    logits_g, probability_g = self.model(input_g)

    loss_func = nn.CrossEntropyLoss(reduction='none') ①
    loss_g = loss_func(
        logits_g,
        label_g[:, 1], ②
    )
    # ... line 220
    return loss_g.mean() ③
```

- ① The reduction='none' gives us loss per sample.
- ② This is the index of the one-hot-encoded class.
- ③ This recombines the loss-per-sample into a single value.

Here, we are *not* using the default behavior to get a loss value averaged over the batch. Instead, we are getting a tensor of loss values, one per sample. This lets us track the individual losses, which means we can aggregate them as we wish (per class, for example). We'll see that in action in just a moment. For now, we'll return the mean of those per-sample losses, which is equivalent to the batch loss. In situations where you don't want to keep statistics per-sample, using the loss averaged over the batch is perfectly fine. If that's the case is highly dependent on your project and goals.

Once that's done, we've fulfilled our obligations to the calling function in terms of what's needed to do backpropagation and weight updates. Before we do that, however, we also want to actually record our per-sample stats for posterity (and later analysis). We'll use the `metrics_t` parameter passed in to accomplish this.

Listing 11.14 p2ch11/training.py, line 26

```
METRICS_LABEL_NDX=0    ①
METRICS_PRED_NDX=1
METRICS_LOSS_NDX=2
METRICS_SIZE = 3

# ... line 207
def computeBatchLoss(self, batch_ndx, batch_tup, batch_size, metrics_g):
    # ... line 220
    start_ndx = batch_ndx * batch_size
    end_ndx = start_ndx + label_t.size(0)  ②

    metrics_g[METRICS_LABEL_NDX, start_ndx:end_ndx] = label_g[:,1]
    metrics_g[METRICS_PRED_NDX, start_ndx:end_ndx] = probability_g[:,1]
    metrics_g[METRICS_LOSS_NDX, start_ndx:end_ndx] = loss_g

    return loss_g.mean()  ③
```

- ① These named array indexes are declared at module-level scope.
- ② The last batch might be smaller than our specified batch size if the total number of samples is not an integer multiple of the batch size.
- ③ We use `detach`, since none of our metrics need to hold onto gradients.
- ④ Again, this is the loss over the entire batch.

By recording the label, prediction, and loss for each and every training (and later, validation) sample, we have a wealth of detailed information we can use to investigate the behavior of our model. For now, we're going to focus on compiling per-class statistics, but we could easily use this information to find the sample that is worst-classified, and start to investigate why. Again, for some projects this kind of information will be less interesting, but it's good to remember that you've got these kinds of options available when needed.

11.4.3 The validation loop is similar

The validation loop in 11.7 looks very similar to training, but is somewhat simplified. The key difference is that validation is read-only. Specifically, the loss value returned is not used, and the weights are not updated.

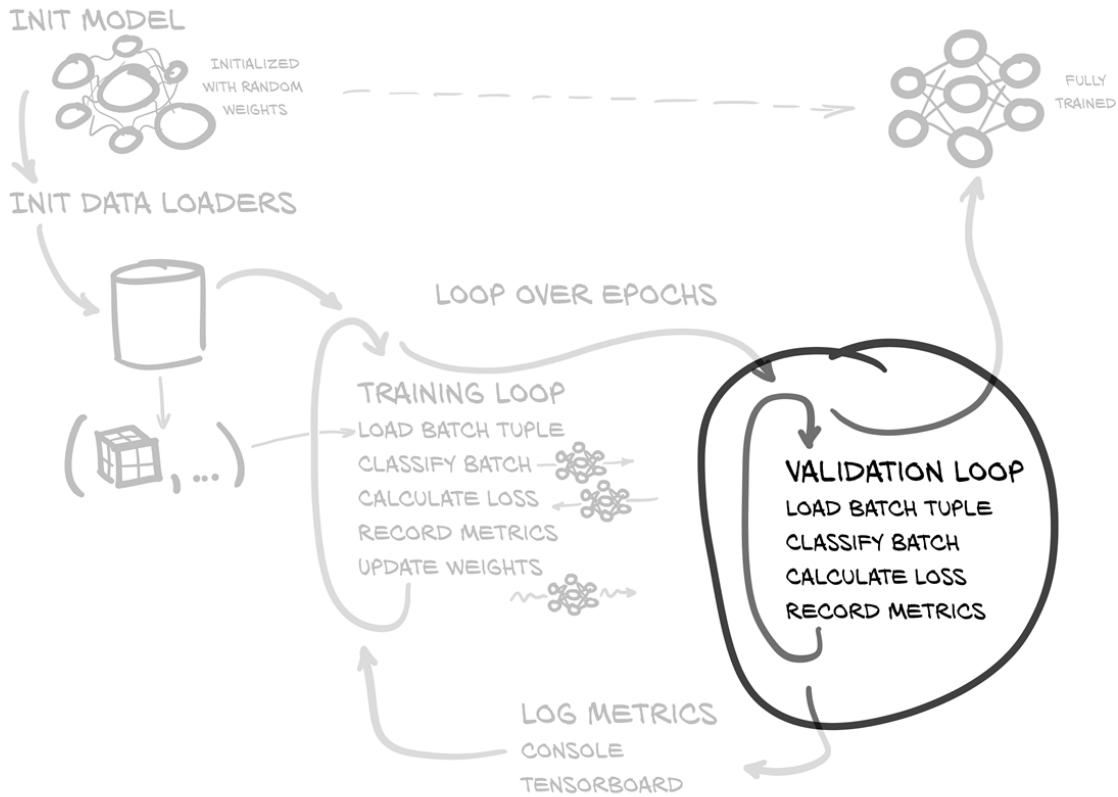


Figure 11.7 The training and validation script we will implement in this chapter, with a focus on the per-epoch validation loop.

Nothing about the model should have changed between the start and end of the process. In addition, it's quite a bit faster due to the `with torch.no_grad()` context manager explicitly informing PyTorch that no gradients need to be computed.

Listing 11.15 p2ch11/training.py, line 127: class LunaTrainingApp.main

```

def main(self):
    for epoch_ndx in range(1, self.cli_args.epochs + 1):
        # ... line 149
        valMetrics_t = self.doValidation(epoch_ndx, val_dl)
        self.logMetrics(epoch_ndx, 'val', valMetrics_t)

    # ... line 191
def doValidation(self, epoch_ndx, val_dl):
    with torch.no_grad():
        self.model.eval() ①
        valMetrics_g = torch.zeros(METRICS_SIZE, len(val_dl.dataset)).to(self.device)
        batch_iter = enumerateWithEstimate(
            val_dl,
            "E{} Validation ".format(epoch_ndx),
            start_ndx=val_dl.num_workers,
        )
        for batch_ndx, batch_tup in batch_iter:
            self.computeBatchLoss(batch_ndx, batch_tup, val_dl.batch_size, valMetrics_g)

    return valMetrics_g.to('cpu')

```

- ① This call turns off training-time behavior.

Without needing to update network weights (recall that doing so would violate the entire premise of the validation set; something we never want to do!), we don't need to use the loss returned from `computeBatchLoss`, nor do we need to reference the optimizer. All that's left is the call to `computeBatchLoss`. We avoid memory issues here by not assigning the loss returned by `computeBatchLoss` to a variable at all. CPython garbage collects the returned value immediately. Note that we are still collecting metrics in the `valMetrics_g`, even though we aren't using the overall per-batch loss for anything.

11.5 Outputting performance metrics

The last thing we do per-epoch is to log our performance metrics for this epoch. As we can see in 11.8, once we've logged metrics, we return to the training loop for the next epoch of training. Logging results and progress as we go is important, since it's possible that training goes off the rails ("does not converge" in the parlance of deep learning), and we want to be able to notice this and stop spending time training a model that's not working out. In the less catastrophic cases, it's good to be able to keep an eye on how your model behaves.

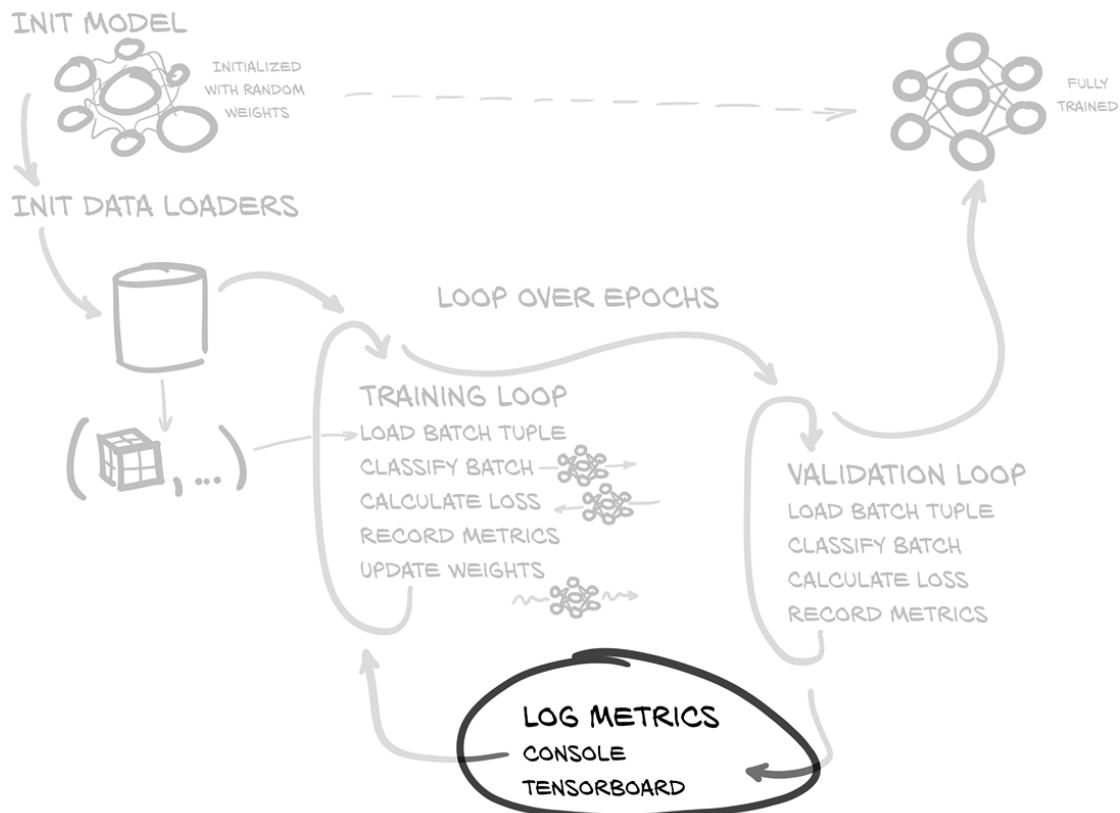


Figure 11.8 The training and validation script we will implement in this chapter, with a focus on the metrics logging to at the end of each epoch.

Logging progress per-epoch is what we were collecting results in `trnMetrics_g` and `valMetrics_g` for earlier. Each of these two tensors now contain everything that we need to be

able to compute our percent correct and average loss per-class for our training and validation runs. Doing this per-epoch is a common choice, though somewhat arbitrary. In future chapters we'll see how we can manipulate the size of our epochs such that we're getting feedback on training progress at a reasonable rate.

11.5.1 The `logMetrics` function

Let's talk about the high-level structure of the `logMetrics` function. The signature looks like this:

Listing 11.16 p2ch11/training.py, line 230: class LunaTrainingApp.logMetrics

```
def logMetrics(
    self,
    epoch_ndx,
    mode_str,
    metrics_t,
):
```

We use the `epoch_ndx` purely for display while logging our results. The `mode_str` argument will tell us if the metrics are for training or validation.

We will consume either `trnMetrics_t` or `valMetrics_t` which gets passed in as the `metrics_t` parameter. Recall that both of those inputs are a tensor of floating point values that we filled with data during `computeBatchLoss`, and then transferred back to the CPU right before we returned them from `doTraining` and `doValidation`. Both tensors have 3 rows and as many columns as we have samples (training samples or validation samples, depending). As a reminder, those three rows correspond to these constants:

Listing 11.17 p2ch11/training.py, line 26

```
METRICS_LABEL_NDX=0 ①
METRICS_PRED_NDX=1
METRICS_LOSS_NDX=2
METRICS_SIZE = 3
```

- ① These are declared at module-level scope.

SIDE BAR Tensor masking and boolean indexing

Masked tensors are a common usage pattern that might be opaque for readers that have not encountered them before. Some readers might be familiar with the concept from Numpy called masked arrays. Both tensor and array masks behave the same way.

For readers who aren't familiar with masked arrays, there's an excellent page in the Numpy documentation ¹²¹ that describes the behavior well. PyTorch purposefully uses the same syntax and semantics as Numpy.

CONSTRUCTING MASKS

Next, we are going to construct masks that will let us limit our metrics to only the benign or malignant samples. We will also count total samples per class, as well as the number of samples we classified correctly.

Listing 11.18 p2ch11/training.py, line 242: class LunaTrainingApp.logMetrics

```
benLabel_mask = metrics_t[METRICS_LABEL_NDX] <= 0.5
benPred_mask = metrics_t[METRICS_PRED_NDX] <= 0.5

malLabel_mask = ~benLabel_mask
malPred_mask = ~benPred_mask
```

While we don't assert it here, we know that all of the values stored in `metrics_t[METRICS_LABEL_NDX]` belong to the set `{0.0, 1.0}` since we know that our malignancy labels are simply `True` or `False`. By comparing to `0.5`, we get an array of binary values where a `True` value corresponds to a non-malignant label for the sample in question.

We do a similar comparison to create the `benPred_mask`, **but we must remember that the `METRICS_PRED_NDX` values are the malignancy predictions produced by our model, and can be any floating point value between 0.0 and 1.0 inclusive**. That doesn't change our comparison, but it does mean the actual value can be close to `0.5`. The malignant masks are simply the inverse of the benign masks.

NOTE

While other projects can utilize similar approaches, it's important to realize that we're taking some shortcuts allowed by this being a binary classification problem. If your next project has more than two classes, or has samples that belong to multiple classes at the same time, you'll have to use some more complicated logic to build similar masks.

Next, we use those masks to compute some per-label statistics and store them in a dictionary `metrics_dict`:

Listing 11.19 p2ch11/training.py, line 248: class LunaTrainingApp.logMetrics

```
ben_count = benLabel_mask.sum()
mal_count = malLabel_mask.sum()

ben_correct = (benLabel_mask & benPred_mask).sum()
mal_correct = (malLabel_mask & malPred_mask).sum()

metrics_dict = {}
metrics_dict['loss/all'] = metrics_t[METRICS_LOSS_NDX].mean()
metrics_dict['loss/ben'] = metrics_t[METRICS_LOSS_NDX, benLabel_mask].mean()
metrics_dict['loss/mal'] = metrics_t[METRICS_LOSS_NDX, malLabel_mask].mean()

metrics_dict['correct/all'] = (mal_correct + ben_correct) / metrics_t.shape[1] * 100
metrics_dict['correct/ben'] = (ben_correct) / ben_count * 100
metrics_dict['correct/mal'] = (mal_correct) / mal_count * 100
```

First we will compute the average loss over the entire epoch. Since the loss is the single metric that is being minimized during training, we're always going to want to be able to keep track of it. Then we will limit the loss averaging to only those samples with a benign label using the `benLabel_mask` we just made. We also do the same with the malignant loss. Computing a per-class loss like this can be useful if one class is persistently harder to classify than another, since that knowledge can help drive investigation and improvements.

We'll close out the calculations with determining the fraction of samples we classified correctly, as well as the fraction correct from each label. Since we will display these numbers as percentages in a moment, we also multiply these values by 100. Similar to the loss, we can use these numbers to help guide our efforts when making improvements. After the calculations, we then log our results with three calls to `log.info`:

Listing 11.20 p2ch11/training.py, line 273: class LunaTrainingApp.logMetrics

```
log.info(
    ("E{{} {:8} {loss/all:.4f} loss, "
     + "{correct/all:-5.1f}% correct, "
    ).format(
        epoch_ndx,
        mode_str,
        **metrics_dict,
    )
)
log.info(
    ("E{{} {:8} {loss/ben:.4f} loss, "
     + "{correct/ben:-5.1f}% correct ({ben_correct:} of {ben_count:})"
    ).format(
        epoch_ndx,
        mode_str + '_ben',
        ben_correct=ben_correct,
        ben_count=ben_count,
        **metrics_dict,
    )
)
log.info( ①
    # ... line 303
)
```

- ① The 'mal' logging is similar to the 'ben' logging above.

The first log has values computed from all of our samples, and is tagged `/all`, while the benign and malignant values are tagged `/ben` and `/mal`. We don't show the third logging statement for malignant values here; it's identical to the second except for swapping "ben" for "mal" in all cases.

11.6 Running the training script

Now that we've completed the core of the `training.py` script, we're going to actually start running it. This will initialize and train our model and print statistics about how well the training is going. The idea is to get this kicked off to run in the background while we're covering the model implementation in detail.

Hopefully we'll have results to look at once we're done.

We're running this script from the main code directory; it should have subdirs of `p2ch11`, `util`, etc. The `python` that we're running should have all the libraries listed in `requirements.txt` installed.

NOTE Information about how to set up a `virtualenv` with the correct libraries can be found in [Appendix](#).

```
$ python -m p2ch11.training ①
Starting LunaTrainingApp,
Namespace(batch_size=256, channels=8, epochs=20, layers=3, num_workers=8)
<p2ch11.dsets.LunaDataset object at 0x7fa53a128710>: 495958 training samples
<p2ch11.dsets.LunaDataset object at 0x7fa537325198>: 55107 validation samples
Epoch 1 of 20, 1938/216 batches of size 256
E1 Training ----/1938, starting
E1 Training    16/1938, done at 2018-02-28 20:52:54, 0:02:57
...
```

- ① This is the command line for linux/bash. Windows users will probably need to invoke `python` differently, depending on the install method used.

As a reminder, we also provide a Jupyter Notebook that contains invocations of the training application:

Listing 11.21 code/p2_run_everything.ipynb

```
# In[5]:
run('p2ch11.prepcache.LunaPrepCacheApp')

# In[6]:
run('p2ch11.training.LunaTrainingApp', '--epochs=1')
```

If the first epoch seems to be taking a very long time (more than ten or twenty minutes), it might be related to needing to prepare the cached data needed by `LunaDataset`. See 10.4.1 in the previous chapter for details on the caching. The exercises for last chapter included writing a script to pre-stuff the cache in an efficient manner. We also provide the `prepcache.py` file to do the same; it can be invoked with `python -m p2ch11.prepcache`. Since we repeat our `dsets.py` files per-chapter, the caching will need to be repeated for every chapter. This is

somewhat space and time inefficient, but it means that we can keep the code for each chapter much more well-contained. For your future projects, we recommend reusing your cache more heavily.

Once training is underway, we want to make sure we're using the computing resources at hand the way we expect. An easy way to tell if the bottleneck is data loading or computation is to wait a few moments after the script starts to train (look for output like `E1 Training 16/7750, done at...`), then check both `top` and `nvidia-smi`.

- If the 8 python worker processes are consuming >80% CPU, then it's likely that the cache needs to be prepared (we know this because the authors have made sure that there aren't generally CPU bottlenecks; this won't be generally true).
- If `nvidia-smi` reports that the GPU-Util is >80%, then you're saturating your GPU.

The intent is that the GPU is saturated; we want to be using as much of that computing power as we can to complete epochs quickly. A single NVIDIA GTX 1080 Ti should complete an epoch in under 15 minutes. Since our model is relatively simple, it doesn't take a lot of CPU pre-processing for the CPU to be the bottleneck. When working with models with greater depth (or more needed calculations in general), processing each batch will take longer, which will increase the amount of CPU processing we can do before the GPU has to idle, waiting until input is ready.

11.6.1 Needed data for training

If the number of samples is lower than 495958 for training or 55107 for validation it might make sense to do some sanity checking to make sure that the full data is present and accounted for. Make sure for your future projects that your `Dataset` is returning the number of samples that you expect.

First, let's take a look at the basic directory structure of our `data-unversioned/part2/luna` directory:

```
$ ls -lp data-unversioned/part2/luna/
subset0/
subset1/
...
subset9/
```

Then, let's make sure that we've got one `.mhd` and one `.raw` file for each series UID:

```
$ ls -lp data-unversioned/part2/luna/subset0/
1.3.6.1.4.1.14519.5.2.1.6279.6001.105756658031515062000744821260.mhd
1.3.6.1.4.1.14519.5.2.1.6279.6001.105756658031515062000744821260.raw
1.3.6.1.4.1.14519.5.2.1.6279.6001.108197895896446896160048741492.mhd
1.3.6.1.4.1.14519.5.2.1.6279.6001.108197895896446896160048741492.raw
...
```

And that we've got the overall correct number of files:

```
$ ls -1 data-unversioned/part2/luna/subset?/* | wc -l
1776
$ ls -1 data-unversioned/part2/luna/subset0/* | wc -l
178
...
$ ls -1 data-unversioned/part2/luna/subset9/* | wc -l
176
```

If all of these seem right, but things still aren't working, ask on Manning LiveBook ¹²² and hopefully someone can help get things sorted out.

11.6.2 Interlude: the `enumerateWithEstimate` function

Working with deep learning involves a lot of waiting. We're talking about real-world, sitting around, glancing at the clock on the wall, a watched pot never boils (but you could fry an egg on the GPU), straight up *boredom*.

The only thing worse than sitting staring at a blinking cursor that hasn't moved in over an hour is flooding your screen with:

```
2020-01-01 10:00:00,056 INFO training batch 1234
2020-01-01 10:00:00,067 INFO training batch 1235
2020-01-01 10:00:00,077 INFO training batch 1236
2020-01-01 10:00:00,087 INFO training batch 1237
...etc...
```

At least the quietly blinking cursor doesn't blow out your scrollback buffer!

Fundamentally, while doing all this waiting we want to answer the question "do I have time to go refill my water glass?" with follow-up questions about having time to:

- Brew a cup of coffee?
- Grab dinner?
- Grab dinner in Paris? 123 124

To answer these pressing questions, we're going to use our `enumerateWithEstimate` function. Usage looks like the following:

```
>>> for i, _ in enumerateWithEstimate(list(range(234)), "sleeping"):
...     time.sleep(random.random())
...
2020-01-01 11:12:41,892 WARNING sleeping ----/234, starting
2020-01-01 11:12:44,542 WARNING sleeping    4/234, done at 2020-01-01 11:15:16, 0:02:35
2020-01-01 11:12:46,599 WARNING sleeping    8/234, done at 2020-01-01 11:14:59, 0:02:17
2020-01-01 11:12:49,534 WARNING sleeping   16/234, done at 2020-01-01 11:14:33, 0:01:51
2020-01-01 11:12:58,219 WARNING sleeping   32/234, done at 2020-01-01 11:14:41, 0:01:59
2020-01-01 11:13:15,216 WARNING sleeping   64/234, done at 2020-01-01 11:14:43, 0:02:01
2020-01-01 11:13:44,233 WARNING sleeping  128/234, done at 2020-01-01 11:14:35, 0:01:53
2020-01-01 11:14:40,083 WARNING sleeping ----/234, done at 2020-01-01 11:14:40
>>>
```

8 lines of output for over 200 iterations, lasting about 2 minutes. Even given the wide variance of `random.random()` the function had a pretty decent estimate after 16 iterations (in less than 10

seconds). For loop bodies with a more constant timing, the estimates stabilize even more quickly.

In terms of behavior, `enumerateWithEstimate` is almost identical to the standard `enumerate` (the differences are things like how our function returns a generator, while `enumerate` returns a specialized `<enumerate object at 0x...>`).

Listing 11.22 util/util.py, line 161: def enumerateWithEstimate

```
def enumerateWithEstimate(
    iter,
    desc_str,
    start_ndx=0,
    print_ndx=4,
    backoff=2,
    iter_len=None,
):
```

However, the side effects (logging, specifically) are what make the function interesting. Rather than get lost in the weeds trying to cover every detail of the implementation, interested readers can consult the function docstring¹²⁵ to get information on the function parameters and desk-check the implementation.

Deep learning projects can be very time intensive. Knowing when something is expected to finish means that you can use your time until then wisely, and it can also clue you in that something isn't working properly (or an approach is unworkable) if the expected time to completion is much larger than expected.

11.7 Evaluating the model: Getting 99.7% correct means we're done, right?

Let's take a look at some output from our training script. As a reminder, we've run this with the command line `python -m p2ch11.training`.

Listing 11.23 Abridged logging output from training.

```
E1 Training ----/969, starting
...
E1 LunaTrainingApp
E1 trn      2.4576 loss,  99.7% correct
...
E1 val      0.0172 loss,  99.8% correct
...
```

After one epoch of training, both the training and validation set show at least 99.7% correct results. That's an A+! Time for a round of high-fives, or at least a satisfied nod-and-smile. We just solved cancer! ...Right?

Well, no.

Let's take a closer look at that epoch 1 output:

Listing 11.24 Less-abridged logging output from training.

```
E1 LunaTrainingApp
E1 trn      2.4576 loss,  99.7% correct,
E1 trn_ben  0.1936 loss,  99.9% correct (494289 of 494743)
E1 trn_mal  924.34 loss,   0.2% correct (3 of 1215)
...
E1 val      0.0172 loss,  99.8% correct,
E1 val_ben  0.0025 loss, 100.0% correct (494743 of 494743)
E1 val_mal  5.9768 loss,   0.0% correct (0 of 1215)
```

On the validation set, we're getting benign nodules 100% correct, but the malignant nodules are 100% wrong. The network is just classifying everything as not-cancer! 99.7% just means that only approximately 0.3% of the samples are malignant.

After 10 epochs, the situation is only marginally better:

```
E10 LunaTrainingApp
E10 trn      0.0024 loss,  99.8% correct
E10 trn_ben  0.0000 loss, 100.0% correct
E10 trn_mal  0.9915 loss,   0.0% correct
E10 val      0.0025 loss,  99.7% correct
E10 val_ben  0.0000 loss, 100.0% correct
E10 val_mal  0.9929 loss,   0.0% correct
```

The classification output remains the same - none of the malignant samples are correctly identified. It's interesting that we're starting to see some decrease in the `val_mal` loss, however, while not seeing a corresponding increase in the `val_ben` loss. That implies that the network *is* learning something. Unfortunately, it's learning it very, very slowly.

Even worse, this particular failure mode is the most dangerous in the real world! We want to avoid the situation where we classify a malignant tumor as benign, as that would not facilitate a patient getting the treatment they need. It's important to understand the consequences for misclassification for all your projects, as that can have a large impact on how you design, train, and evaluate your model. We'll discuss this more in the next chapter.

Before we get to that, however, we need to upgrade our tooling to make the results easier to understand. I'm sure our readers love to squint at columns of numbers as much as anyone, but pictures are worth a thousand words. Let's graph some of these metrics.

11.8 Graphing training metrics with TensorBoard

We're going to be using a tool called "TensorBoard" as a quick and easy way to get our training metrics out of our training loop and into some pretty graphs. This will allow us to follow the *trends* of those metrics, rather than only looking at the instantaneous values per-epoch. It gets much, much easier to know if a value is an outlier or just the latest in a trend when you're looking at a visual representation.

"Hey, wait," you might be thinking, "isn't TensorBoard part of the TensorFlow project? What's it doing here in my PyTorch book?"

Well, yes, it is part of another deep learning framework, but our philosophy is "use what works." There's no reason to restrict ourselves by not using a tool because it's bundled with another project we're not using. Both the PyTorch and TensorBoard devs agree, because they collaborated to add official support for TensorBoard into PyTorch. TensorBoard is great, and it's got some easy to use PyTorch APIs that let us hook data from just about anywhere into it for quick and easy display. If you stick with deep learning, you'll probably be seeing (and using) a *lot* of TensorBoard.

In fact, if you've been running the chapter examples, you should already have some data on disk ready and waiting to be displayed. Let's see how to run it, and look at what it can show us.

11.8.1 Running TensorBoard

By default, our training script will write metrics data to the `runs/` subdirectory. If you list the directory content, you might see something like this:

Listing 11.25 Bash shell session

```
$ ls -lA runs/p2ch11/
total 24
drwxrwxr-x 2 elis elis 4096 Sep 15 13:22 2020-01-01_12.55.27-trn-dlwpt/ ①
drwxrwxr-x 2 elis elis 4096 Sep 15 13:22 2020-01-01_12.55.27-val-dlwpt/ ①
drwxrwxr-x 2 elis elis 4096 Sep 15 15:14 2020-01-01_13.31.23-trn-dwlpt/ ②
drwxrwxr-x 2 elis elis 4096 Sep 15 15:14 2020-01-01_13.31.23-val-dwlpt/ ②
```

- ① This would be the single-epoch run from earlier.
- ② This would be the more recent ten-epoch training run.

To get the `tensorboard` program, install the `tensorflow`¹²⁶ python package. Since we're not actually going to use TensorFlow proper, it's fine if you install the default CPU-only package. If you have another version of TensorBoard installed already, using that one is fine too. Either make sure that the appropriate directory is on your path, or invoke it with `..../path/to/tensorboard --logdir runs/`. It doesn't really matter where you invoke it from, so long as you use the `--logdir` argument to point it at where your data is stored. It's a good idea to segregate your data into separate folders, as TensorBoard can get a bit unwieldy once you get over ten or twenty experiments. You'll have to decide the best way to do that for each project as you go. Don't be afraid to move data around after the fact if you need to.

Let's start TensorBoard now:

Listing 11.26 Bash shell session

```
$ tensorboard --logdir runs/
2020-01-01 12:13:16.163044: I tensorflow/core/platform/cpu_feature_guard.cc:140]     ①
      Your CPU supports instructions that this TensorFlow binary was not compiled to use: AVX2 FMA
①
TensorBoard 1.14.0 at http://localhost:6006/ (Press CTRL+C to quit)
```

- ① These messages might be different or not present for you; that's fine.

Once that's done, you should be able to point your browser at localhost:6006¹²⁷ and see the main dashboard. 11.9 shows us what that looks like:

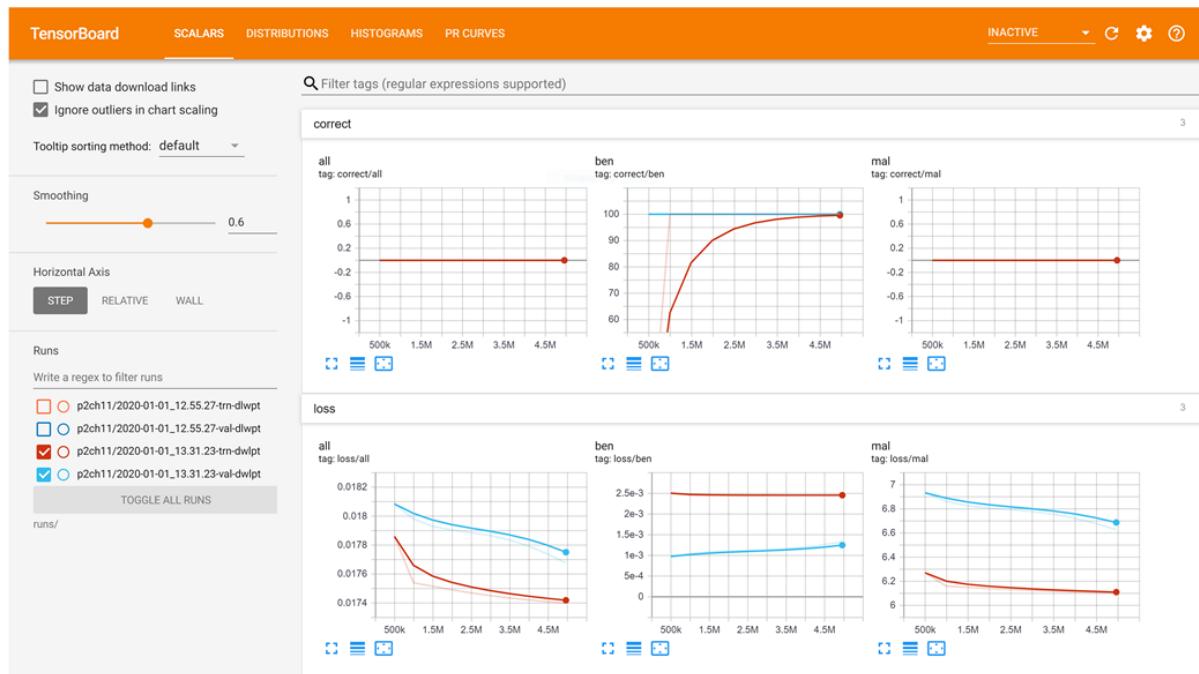


Figure 11.9 The main TensorBoard UI, showing a paired set of training and validation runs.

Along the top of the browser window you should see the orange header. The right side has the typical widgets for settings, a link to the GitHub repository, and the like. We can ignore those for now. The left has header items for the data types we've provided. You should have at least the following:

- Scalars (the default tab)
- Histograms
- Precision-Recall Curves (shown as "PR Curves")

You might see "Distributions" as well as the second UI tab (to the right of "Scalars" in 11.9). We won't be using or discussing those here. Let's make sure we've selected "Scalars" by clicking on it.

On the left side there is a set of controls for display options, as well as a list of runs that are present. The smoothing option can be useful if you have particularly noisy data; it will calm things down so that you can pick out the overall trend. The original non-smoothed data will still be visible in the background as a faded line in the same color. 11.10 shows this, though it might be difficult to discern when printed in black and white.

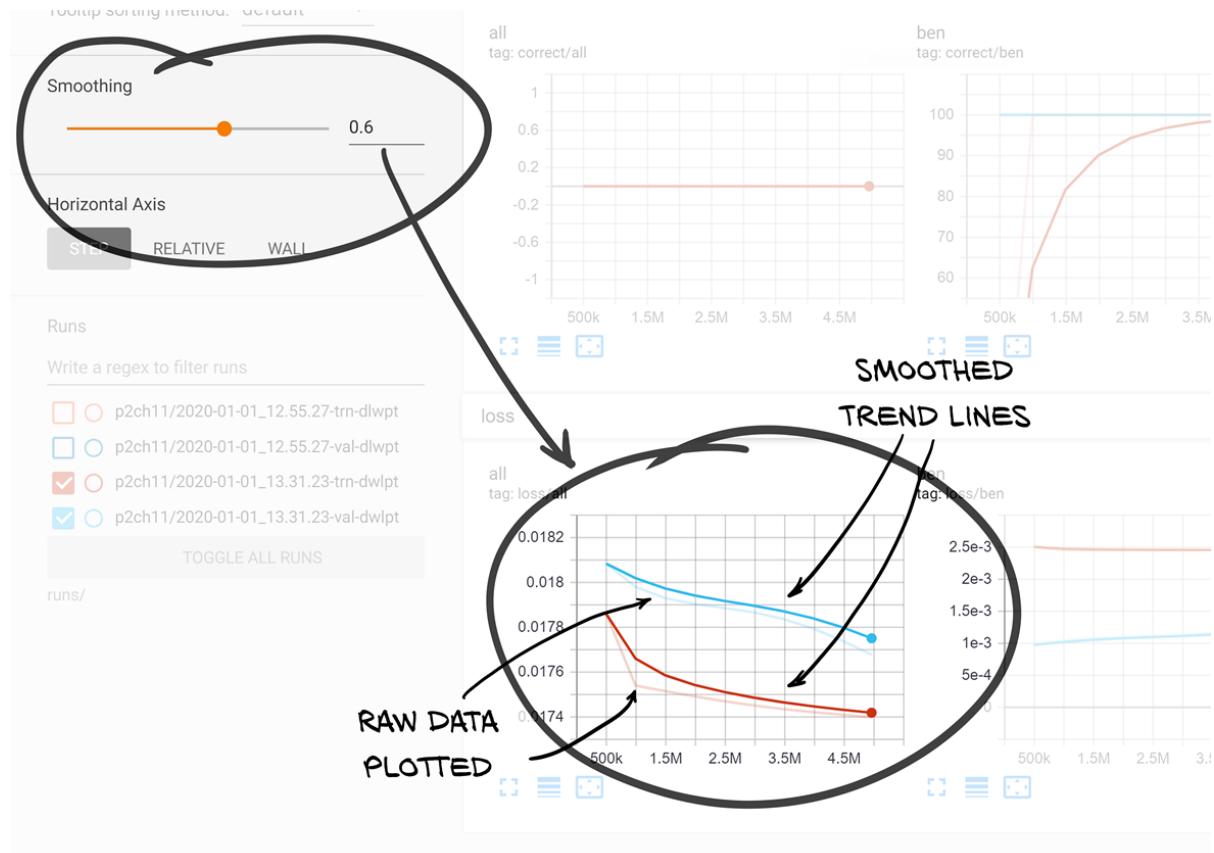


Figure 11.10 The TensorBoard sidebar with Smoothing set to 0.6 and two runs selected for display.

Depending on how many times you've run the training script, you might have multiple runs to select from. With too many runs being rendered, the graphs can get overly noisy, so don't hesitate to deselect runs that aren't of interest at the moment.

If you want to permanently remove a run, the data can be deleted from disk while TensorBoard is running. We can do this to get rid of experiments that crashed, had bugs, didn't converge, or are so old they're no longer interesting. The number of runs can grow pretty quickly, so it can be helpful to prune it often, and to rename or move runs that are particularly interesting to a more permanent directory so they don't get deleted by accident. To remove both the `train` and `validation` runs, you can execute the following (after changing the chapter, date, and time to match the run you're wanting to remove):

Listing 11.27 Bash shell session

```
$ rm -rf runs/p2ch11/2020-01-01_12.02.15_*
```

Keep in mind that removing runs will cause the runs that are later in the list to move up, which will result in them being assigned new colors.

Okay, let's get to the point of TensorBoard: the pretty graphs! The main part of the screen should be filled with data from our training and validation metrics gathering, like 11.11:

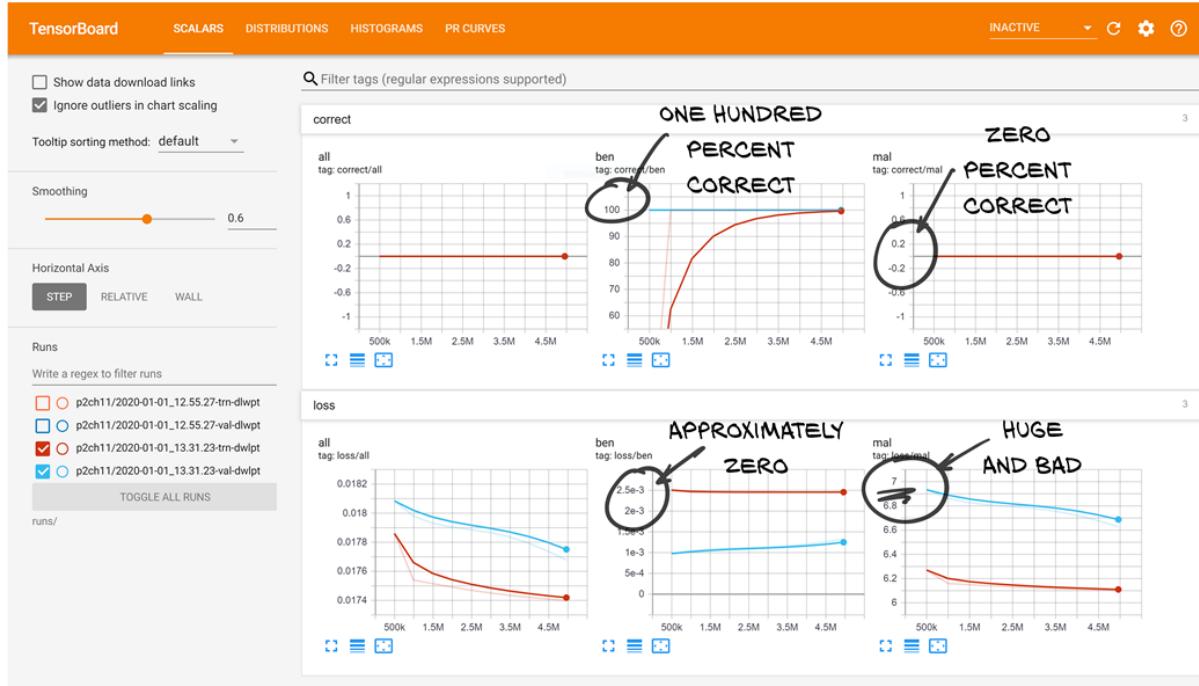


Figure 11.11 The main TensorBoard data display area showing us that our results on malignant nodules are downright awful.

That's much easier to parse and absorb than `E1 trn_mal 924.34 loss, 0.2% correct (3 of 1215)`! While we're going to save discussion of what these graphs are telling us for 11.9, now would be a good time to make sure that it's clear what these numbers correspond to from our training program. Take a moment to cross-reference the numbers you get by mousing over the lines with the numbers spit out by `training.py` during the same training run. You should see a direct correspondence between the "Value" column of the tooltip and the values printed during training.

Once you're comfortable and confident that you understand exactly what TensorBoard is showing you, we'll move on and discuss how to get these numbers to show up in the first place.

11.8.2 Adding tensorboard support to our metrics logging function

We are going to be using the `torch.utils.tensorboard` module to write data in a format that TensorBoard will consume. This will allow us to write metrics for this and any other project quickly and easily.

TensorBoard supports a mix of numpy arrays and PyTorch tensors, but since we don't have any reason to put our data into numpy arrays, we are going to be using PyTorch tensors exclusively.

The first thing that we're going to need to do is to create our `SummaryWriter` objects (which we imported from `torch.utils.tensorboard`). The only parameter we're going to pass in is the `log_dir`, which we will initialize to something like `runs/p2ch11/2020-01-01_12.55.27-trn-dlwpt`. We can add a "comment" argument to our training script to change the `dlwpt` to something more informative. See `python -m p2ch11.training --help` for more information.

We create two writers, one each for the training and validation runs. Those writers will be reused for every epoch. This function gets called from `main()` rather than `{uu}init{uu}` since there's a side-effect of the directories for the runs having been created, which also shows up in tensorboard. We want to wait to do that as long as possible, so that we minimize the number of runs with no data.

Listing 11.28 p2ch11/training.py, line 119: class LunaTrainingApp.initTensorboardWriters

```
def initTensorboardWriters(self):
    if self.trn_writer is None:
        log_dir = os.path.join('runs', self.cli_args.tb_prefix, self.time_str)

        self.trn_writer = SummaryWriter(log_dir=log_dir + '-trn_cls-' + self.cli_args.comment)
        self.val_writer = SummaryWriter(log_dir=log_dir + '-val_cls-' + self.cli_args.comment)
```

If you recall, the first epoch is a bit of a mess, with the early output in the training loop being essentially random. When we save the metrics from that first batch those random results end up skewing things a bit. As we can recall from 11.10, TensorBoard has smoothing to remove noise from the trend lines, which helps somewhat.

Another approach could be to skip metrics for the first epoch's training data entirely, though our models train quickly enough that it's still useful to see the first epoch's results. Feel free to change this behavior as you see fit, though the rest of part 2 will continue with this pattern of including the first, noisy training epoch.

TIP

If you end up doing a lot of experiments that result in exceptions or killing the training script relatively quickly, you might be left with a number of junk runs cluttering up your `runs/` directory. Don't be afraid to clean those out!

WRITING SCALARS TO TENSORBOARD

Writing scalars is straightforward. We can take the `metrics_dict` we've already constructed, and pass in each key/value pair to the `writer.add_scalar` method. The `torch.utils.tensorboard.SummaryWriter` class has the `add_scalar` method¹²⁸, with signature:

Listing 11.29 PyTorch torch/utils/tensorboard/writer.py, line 267: class SummaryWriter

```
def add_scalar(self, tag, scalar_value, global_step=None, walltime=None):
    # ...
```

The `tag` parameter tells tensorboard which graph we're adding values to, the `scalar_value` parameter is our data point's Y axis value. The `global_step` parameter acts as the X-axis value.

As you may recall, we updated the `totalTrainingSamples_count` variable inside of the `doTraining` function. We'll use `totalTrainingSamples_count` as the X-axis of our TensorBoard plots by passing it in as the `global_step` parameter.

Here's what that looks like in our code:

Listing 11.30 p2ch11/training.py, line 307: class LunaTrainingApp.logMetrics

```
for key, value in metrics_dict.items():
    writer.add_scalar(key, value, self.totalTrainingSamples_count)
```

Note that the slashes in our key names (e.g. `'loss/all'`) will result in TensorBoard grouping the charts by the substring before the `'/'`.

The documentation suggests that we should be passing in the epoch number as the `global_step` parameter, but that results in some complications. By using the number of training samples presented to the network, we can do things like change the number of samples per epoch and still be able to compare those future graphs to the ones we're creating now. Saying that a model trains in half the number of epochs is meaningless if each epoch takes four times as long!

Keep in mind that this might not be standard practice, however, so expect to see a variety of values used for the global step.

11.9 Why is the model not learning to detect malignant tumors?

Our model is clearly learning *something* — the loss trend lines are consistent as epochs increase, and the results are repeatable. There is a disconnect, however, between what the model is learning and what we *want* it to learn. What's going on? Let's use a quick metaphor to illustrate the problem.

Imagine that a professor gives students a final exam consisting of 100 true/false questions. The students have access to previous versions of this professor's tests going back 30 years, every time there are only *one or two* questions with a true answer. The other 98 or 99 are false, every time.

Assuming that the grades aren't on a curve and instead have a typical scale of 90% correct or better being an A, etc. then it is trivial to get an A+. Just mark every question as false! Let's imagine that this year, there is only one true answer. A student like the one on the left in 11.12 that mindlessly marked every answer as false would get a 99% on the final, but wouldn't have really demonstrated that they had learned anything (beyond how to cram from old tests, of course). That's basically what our model is doing right now.

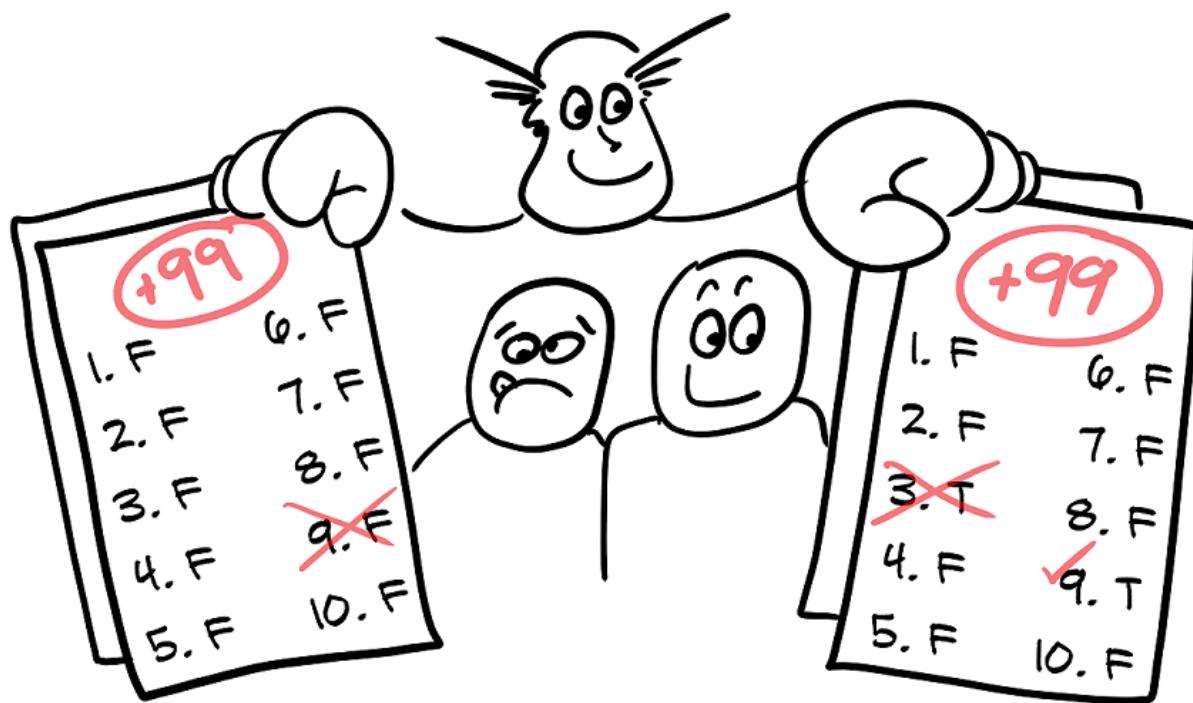


Figure 11.12 A professor giving two students the same grade, despite different levels of knowledge. Question 9 is the only question with an answer of "true."

Contrast that with a student like the one on the right that also got 99% of the questions correct. Intuition tells us that the student on the right in 11.12 probably has a much better grasp of the material than the all-false student. Finding the one true question while only getting one answer wrong is pretty difficult!

Unfortunately, neither our students' grades nor our model's grading scheme reflect this gut feeling.

We have a similar situation, where 99.7% of the answers to "is this nodule malignant?" is "nope." Our model is taking the easy way out, and answering "false" on every question.

Still, if we look back at our model's numbers more closely, the loss on the training and validation sets *is* decreasing! The fact that we're getting any traction at all on the cancer detection problem should give us hope. It will be the work of the next chapter to realize this potential. We'll start next chapter by introducing some new, relevant terminology, and then coming up with a better grading scheme that doesn't lend itself to being gamed quite as easily as what we've done so far.

11.10 Conclusion

We've come a long way this chapter — we now have a model, a training loop, and are able to consume the data we produced last chapter. We've got our metrics being logged to the console, as well as graphed visually.

While our results aren't usable yet, we're actually closer than it might seem. In chapter 12 we will improve the metrics that we're using to track our progress, and use that to inform the changes we need to make to get our model producing reasonable results.

11.11 Exercises

1. Implement a program that iterates through a `LunaDataset` instance by wrapping it in a `DataLoader` instance, and times how long it takes to do so. Compare these times to the times from the exercises in the last chapter. Make sure to be aware of the state of the cache when running the script.
 - A. What impact does setting `num_workers=...` to 0, 1, and 2 have?
 - B. What are the highest values your machine will support for a given combination of `batch_size=...` and `num_workers=...` without running out of memory?
2. Reverse the sort order of the `noduleInfo_list`. How does that change the behavior of the model after one epoch of training?
3. Change `logMetrics` to alter the naming scheme of the runs and keys that get used in TensorBoard.
 - A. Experiment with different forward slash placement for keys passed into `writer.add_scalar`.
 - B. Have both training and validation runs use the same writer, and add the `trn` or `val` string to the name of the key.
 - C. Customize the naming of the log directory and keys to suit your tastes.

11.12 Summary

- `DataLoader`s can be used to load data from arbitrary `Dataset`s in multiple processes. This allows otherwise-idle CPU resources to be devoted to preparing data to feed to the GPU.
- `DataLoader`s will load multiple samples from a `Dataset` and collate them into a batch. PyTorch models expect to process batches of data, not individual samples.
- `DataLoader`s can be used to manipulate arbitrary `Dataset`s by changing the relative frequency of individual samples. This allows for "aftermarket" tweaks to a `Dataset`, though it might make more sense to change the `Dataset` implementation directly.
- We will use PyTorch's `torch.optim.SGD` (Stochastic Gradient Descent) optimizer with a learning rate of 0.001 and a momentum of 0.99. This is a reasonable default for many deep learning projects.
- Our initial model for classification will be very similar to the model we used in chapter 8. This lets us get started with a model that we have reason to believe will be effective. We can revisit the model design once we believe it's the thing preventing our project from performing better.
- Choice of metrics that we monitor during training is important. It is easy to accidentally pick metrics that are misleading about how the model is performing. Using overall percentage of samples classified correctly is not useful for our data. Next chapter will detail how to evaluate and choose better metrics.
- TensorBoard can be used to display a wide range of metrics visually. This makes it much easier to consume certain forms of information, particularly trend data, as it changes per epoch of training.

12

Monitoring Metrics: Precision, Recall, and Pretty Pictures

This chapter covers:

- Defining precision, recall, true/false positives/negatives, how they relate to one another, and what they mean in terms of our model's performance.
- A new quality metric, the F1 score, and its strengths compared to other possible quality metrics.
- Updating our `logMetrics` function to compute and store precision, recall, and F1 score.
- Balancing our `LunaDataset` to address the training issues uncovered at the end of chapter 8.
- Using TensorBoard to graph our quality metrics as each epoch of training occurs, and verifying that our work to balance the data results in an improved F1 score.

NOTE

MEAP readers, please be aware that some of the code on github for this chapter has been updated, and now performs better than it did when this chapter was first written. In particular, some of the TensorBoard graphs will look different. This will be corrected in future updates.

The close of the last chapter left us in a predicament. While we were able to get the mechanics of our deep learning project in place, none of the results were actually useful; the network simply classified everything as benign! To make matters worse, the results seemed great on the surface, since we were looking at the percent of the training and validation sets that were classified correctly. With our data heavily skewed towards benign samples, blindly calling everything benign is a quick and easy way for our model to score well.

Too bad doing so makes it basically useless!

This chapter is all about how to measure, quantify, express, and then improve upon how well our model is doing its job.

We will be dealing with the issues we're facing, like excessive focus on a single, narrow metric and the resulting behavior being useless in the general sense. In order to make some of this chapter's concepts a bit more concrete, we are first going to employ a metaphor that puts our troubles in more tangible terms.

After that, we will develop a graphical language to represent some of the core concepts needed to formally discuss the issues with the implementation from the last chapter. Once we have those concepts solidified, we'll touch on some math using those concepts that will encapsulate a more robust way of grading our model's performance and condensing it down into a single number. We will implement the formula for those new metrics, employ some visualization tools, and look at the how the resulting values change epoch-by-epoch during training. Finally, we'll make some much-needed changes to our `LunaDataset` implementation with an aim at improving our training results, and then see if those experimental changes have the expected impact on our performance metrics.

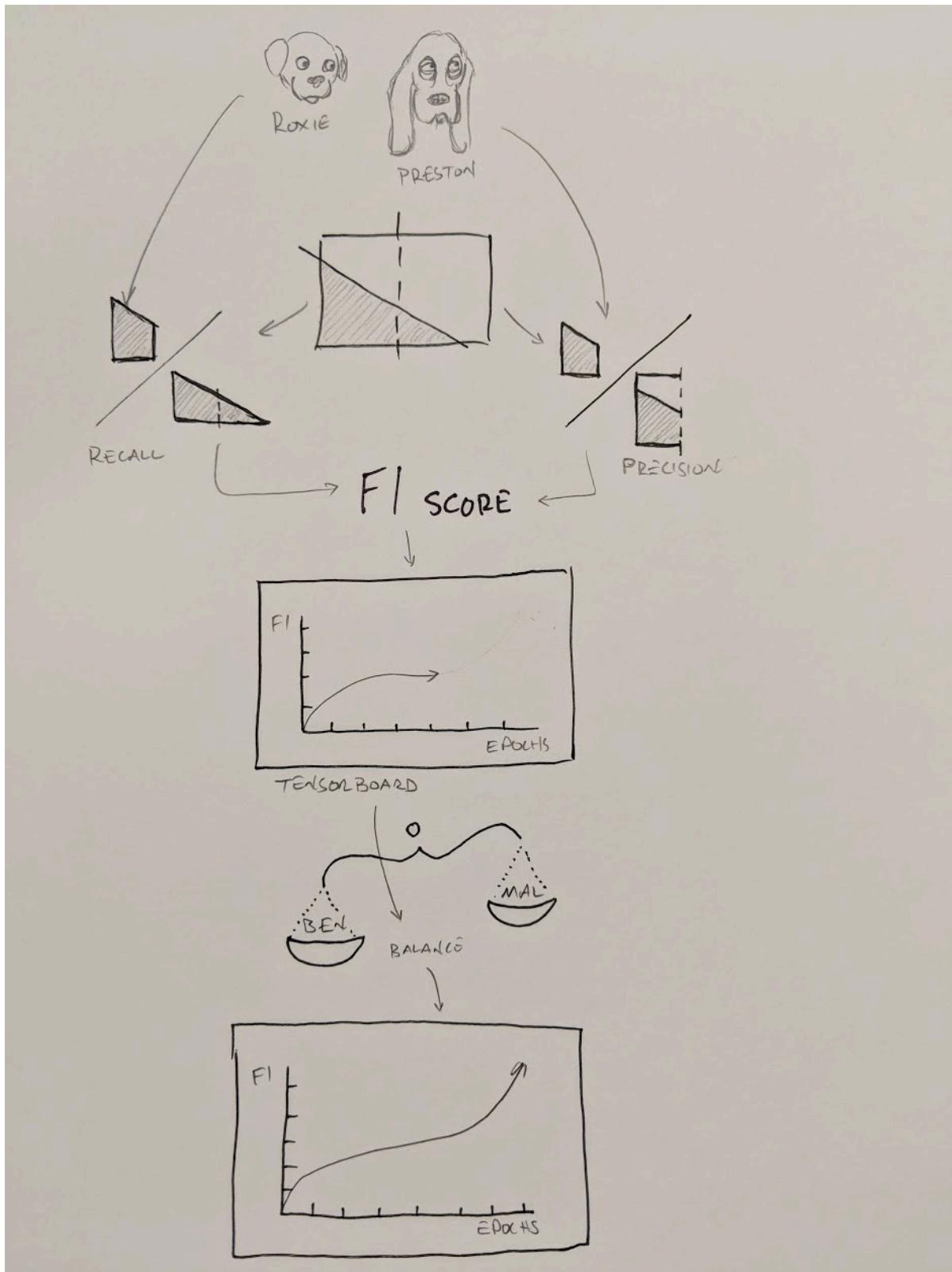


Figure 12.1 TODO IMAGE: mental model

By the time we're through this chapter, our trained model will be performing much better. While it won't be ready to drop into clinical use just yet, it will be capable of producing results that are clearly better than random. This will mean that we have a workable implementation of step 3,

nodule classification, and once we're done we can begin to think about how to incorporate steps 1 (segmentation) and 2 (clustering) to the project.

12.1 Good dogs versus bad guys: false positives and false negatives

Instead of models and tumors, we're going to consider two guard dogs, both fresh out of obedience school. They both want to alert us to burglars — a rare but serious situation that requires prompt attention.

Unfortunately, while both dogs are good dogs, neither are good *guard* dogs. Roxie barks at just about everything, while Preston will bark almost exclusively at burglars, but only if he happens to be awake when they arrive.

Roxie *will* alert you to the burglar, just about every time. She will also alert you to fire engines, the mail carrier, squirrels, passerby, etc. If you follow up on every bark, you'll almost never get robbed (only the sneakiest of sneak-thieves can slip past). Perfect! ...Except that being that diligent means that you're not really saving any work by having a guard dog. Instead, you'll be up every couple of hours, flashlight in hand, due to Roxie having smelled a mouse, or heard an owl, or seen a late bus wander by.

Roxie has a problematic number of **false positives**.

A **false positive** is an event that is classified as of interest or as a member of the desired class (positive, as in "yes, that's the type of thing I'm interested in knowing about"), but that in truth is **not** actually of interest.

For Roxie, these would be those fire engines, squirrels, etc. For the cancer detection problem, it's when an actually benign nodule is flagged as possibly malignant and need of a biopsy.

Contrast with **true positive** which are items of interest that are classified correctly.

Meanwhile, if Preston barks, call the police, since that means that someone's almost certainly broken in. Preston is a deep sleeper, however, and the sound of an in-progress home invasion isn't likely to rouse him, so you're still going to get robbed just about every time someone tries. Again, while it's better than nothing, you're not really ending up with the peace of mind that motivated getting a dog in the first place.

Preston has a problematic number of **false negatives**.

A **false negative** is an event that is classified as not of interest or not a member of the desired class (negative, as in "no, that's not the type of thing I'm interested in knowing about"), but that in truth **is** actually of interest.

For Preston, these would be the robberies that he sleeps through. For the cancer detection problem, it's when a malignant tumor goes undetected.

Contrast with **true negative** which are uninteresting items that are correctly identified as such.

Just to complete the metaphor, last chapter's model is basically a guard *cat* who refuses to meow at anything that isn't a can of tuna.

Our earlier focus at the end of the last chapter was on percent correct for the overall training and validation sets. Clearly, that wasn't a great way to grade ourselves, and as we can see from each of our dogs' myopic focus on a single metric — like number of true positives or true negatives — we need a metric with a broader focus to capture our overall performance.

12.2 Graphing the positives and negatives

Let's start developing the visual language that we'll use to describe true/false positives/negatives. Consider the following diagram of positive events (or samples) in the lower left, and negative events in the upper right:

PLACEHOLDER

Figure 12.2 TODO IMAGE: Dangerous things on the left, safe things on the right; quiet things on top, loud things on the bottom.

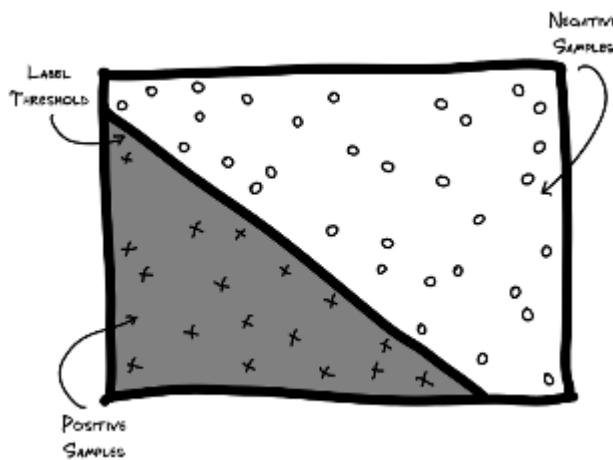


Figure 12.3 Positive samples and negative samples mapped into a 2D representation.

Each of the Xs and Os represent a discrete event (say, someone approaching the house, or a specific nodule), with the Xs as positive events (burglars, or cancer) and the Os as negative events (mail carriers, or benign nodules).

The actual input data that we're going to use has high dimensionality — there's a ton of CT

voxel values to consider, along with more abstract things like nodule size, overall location in the lungs, etc. The job of our model is to map each of these events into this rectangle in such a way that we can separate those positive and negative events cleanly using a single vertical line. This is done by the `nn.Linear` layers at the end of our model. The position of the vertical line corresponds exactly to the `classificationThreshold_float` we saw in the last chapter, in section 11.5.1. There, we chose the value 0.5 as our threshold.

Note that in reality, the data presented is not two-dimensional; it goes from very-high-dimensional after the second-to-last layer, to one-dimensional (here, our X axis) at the output — just a single scalar per sample (which is then bisected by the classification threshold). Here, we use the second dimension (the Y axis) to represent per-sample features that our model cannot see or use. Things like age or gender of the patient, location of the nodule in the lung, or even local aspects of the nodule that the model hasn't utilized. It also gives us a convenient way to represent confusion between benign and malignant samples.

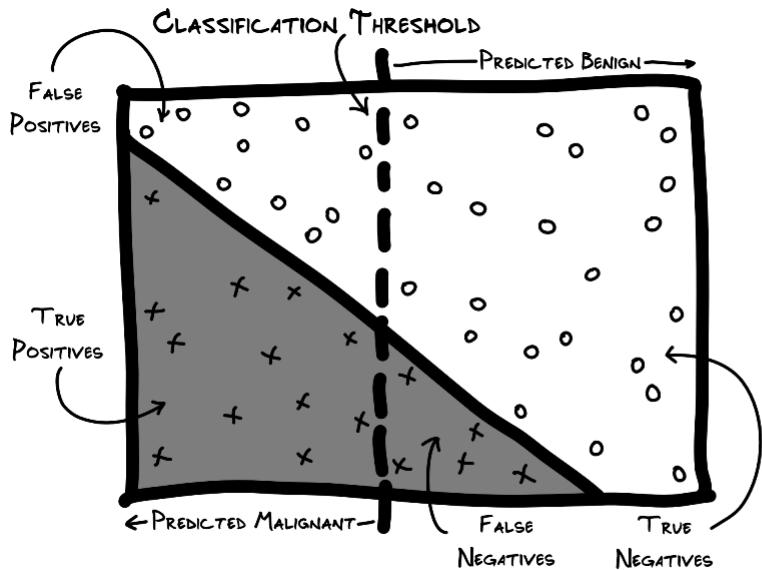


Figure 12.4 2D representation with a classification threshold.

We've added our classification threshold to our image; the exact position shown here was chosen somewhat arbitrarily to be "kind of in the middle-ish." This threshold separates our results into four buckets: true positives and true negatives (both of which are classified correctly); false positives and false negatives (both of which are misclassified).

The areas in Figure-10.3 will be the basis that we use to discuss model performance, since we can use the ratios between these values to construct increasingly complex metrics that we can use to objectively measure how well we are doing.

As they say, "the proof is in the proportions.¹²⁹" Next, we'll use those proportions to start defining better metrics.

12.2.1 Recall

Roxie's strength is recall. Recall is basically "did you miss any interesting events?" To improve recall, minimize false negatives. In guard dog terms, that means if you're unsure, bark at it, just in case.

NOTE

In some contexts, recall is referred to as "sensitivity."

More formally, **recall is the ratio of the true positives to the union of true positives and false negatives.**

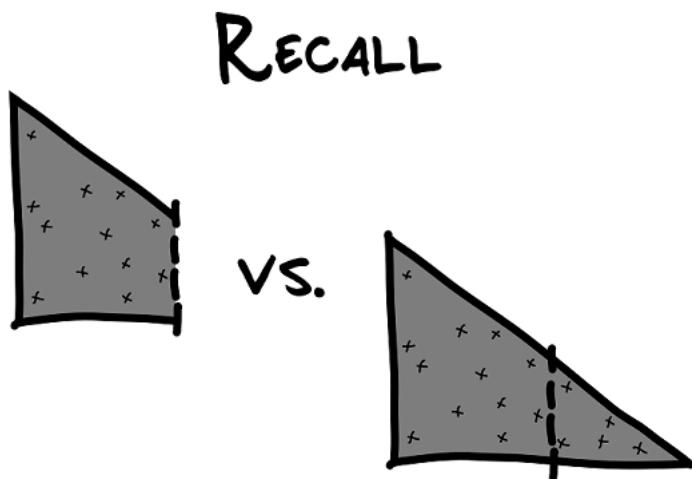


Figure 12.5 Recall is the ratio of the true positives to the union of true positives and false negatives.

Roxie accomplishes having an incredibly high recall by pushing her classification threshold all the way to the right, such that it totally encompasses all of the positive events:

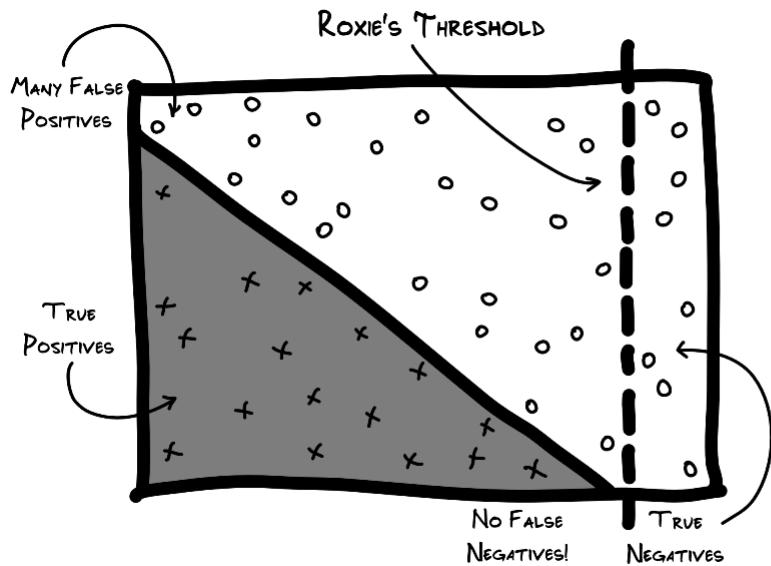


Figure 12.6 Roxie's choice of threshold prioritizes minimizing false negatives.

Note how doing so means that her recall value is 1.0, which means that 100% of the robbers were barked at. Since that's how Roxie defines success, in her mind, she's doing a great job. Never mind the huge expanse of false positives!

12.2.2 Precision

Preston's strength is precision, which is basically "did you ever flag something you shouldn't have?" To improve precision, minimize false positives. Preston won't bark at something unless he's certain it's a burglar.

More formally, **precision is the ratio of the true positives to the union of true positives and false positives.**

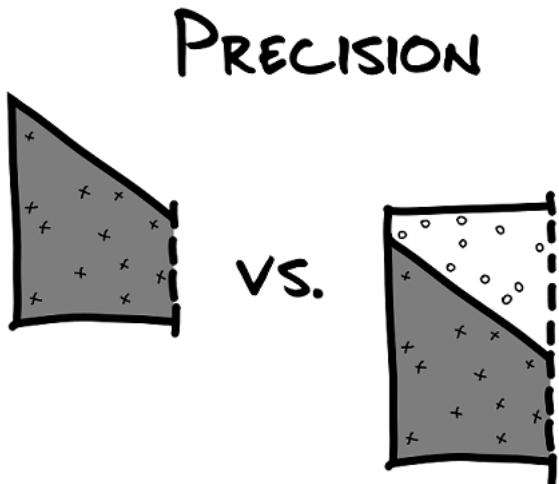


Figure 12.7 Precision is the ratio of the true positives to the union of true positives and false positives.

Preston accomplishes having an incredibly high precision by pushing his classification threshold all the way to the left, such that it excludes as many negative events as he can manage:

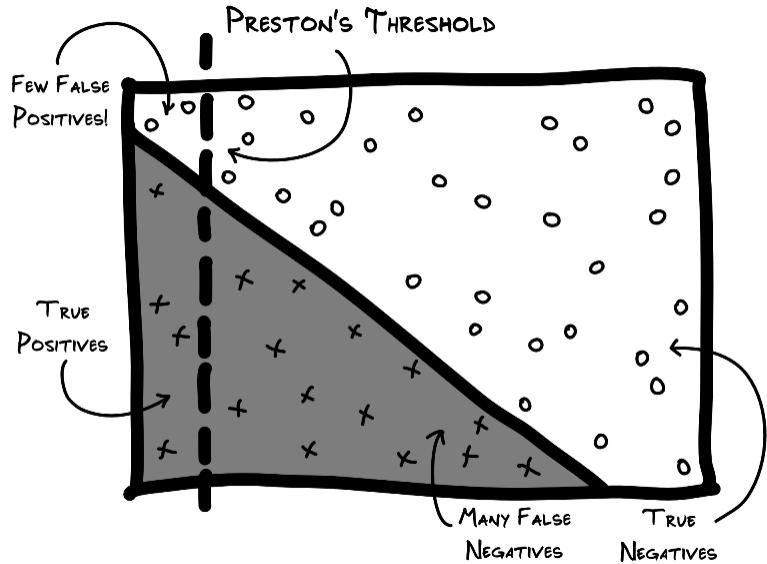


Figure 12.8 Preston's choice of threshold prioritizes minimizing false positives.

Opposite of Roxie's approach, this means that Preston has a precision of nearly 1.0, which means that almost 100% of the things that he barked at were robbers. This also matches his definition of being a good guard dog, even though in doing so it means that there are a large number of events that pass undetected.

12.2.3 Implementing precision and recall in `logMetrics`

Both precision and recall are valuable metrics to be able to track during training, since they provide important insight into how the model is behaving. If either of them drops to zero (like we saw last chapter!), it's likely that our model has started to behave in a degenerate manner. We can use the exact details of the behavior to guide where to investigate and experiment with getting training back on track. We'd like to update the `logMetrics` function to add them to the output we see each epoch, to compliment the loss and correctness metrics we already have.

Since we've been defining them in terms of "true positives" and the like thus far, we will continue to do so in the code. It turns out that we are already computing some of the values that we need, though we had named them differently.

Listing 12.1 p2ch12/training.py, line 290: class LunaTrainingApp.logMetrics

```
ben_count = int(benLabel_mask.sum())
mal_count = int(malLabel_mask.sum())

trueNeg_count = ben_correct = int((benLabel_mask & benPred_mask).sum())
truePos_count = mal_correct = int((malLabel_mask & malPred_mask).sum())

falsePos_count = ben_count - ben_correct
falseNeg_count = mal_count - mal_correct
```

Here, we can see that `benCorrect_count` is the same thing as `trueNeg_count`! That actually makes sense, since benign is our "negative" value (as in "a negative diagnosis"), and if the classifier got the prediction correct, then that's a true negative. Similarly, correctly labeled malignant samples are true positives.

We do need to add the variables for our false positive and false negative values. That's straightforward, since we can take the total number of benign labels and subtract the count of the correct ones. What's left is the count of benign samples that were misclassified *as positive*. Hence, they are false positives. Again, the false negative calculation is of the same form, but uses malignant counts.

With those values, we can compute `precision` and `recall` and store them in our `metrics_dict`:

Listing 12.2 p2ch12/training.py, line 308: class LunaTrainingApp.logMetrics

```
precision = metrics_dict['pr/precision'] = \
    truePos_count / np.float64(truePos_count + falsePos_count)
recall = metrics_dict['pr/recall'] = \
    truePos_count / np.float64(truePos_count + falseNeg_count)
```

Note the double assignment; while having separate `precision` and `recall` variables isn't strictly necessary, they improve the readability of the next section.

We also extend the logging statement in `logMetrics` to include the new values, but we skip the implementation for now (we'll revisit the logging again later in the chapter).

12.2.4 Our ultimate performance metric: the F1 score

While useful, neither of precision or recall entirely captures what we need to be able to evaluate a model. As we've seen with Roxy and Preston, it's possible to game either one individually by manipulating our classification threshold, resulting in a model that scores well on one or the other, but does so at the expense of any real-world utility. We need something that combines both of those values in a way that prevents such gamesmanship.

The generally accepted way of doing this is by using the F1 score.¹³⁰ As with other metrics, F1 score ranges between zero (a classifier with no real-world predictive power) and one (a classifier

that has perfect predictions). We will update `logMetrics` to include this as well:

Listing 12.3 p2ch12/training.py, line 313: class LunaTrainingApp.logMetrics

```
metrics_dict['pr/f1_score'] = \  
    2 * (precision * recall) / (precision + recall)
```

At first glance, this might seem more complicated than we need, and it might not be immediately obvious how the F1 score behaves when trading off precision for recall or vice versa. This formula has a lot of nice properties, however, and it compares favorably to several other, simpler alternatives that we might consider.

One immediate possibility for a scoring function is to average the values for precision and recall together. Unfortunately, this gives both `avg(p=1.0, r=0.0)` and `avg(p=0.5, r=0.5)` the same score of 0.5, and as we discussed earlier, a classifier with either of precision or recall of zero is usually worthless. Giving something useless the same non-zero score as something useful disqualifies averaging as a meaningful metric immediately.

Still, let's visually compare averaging and F1.

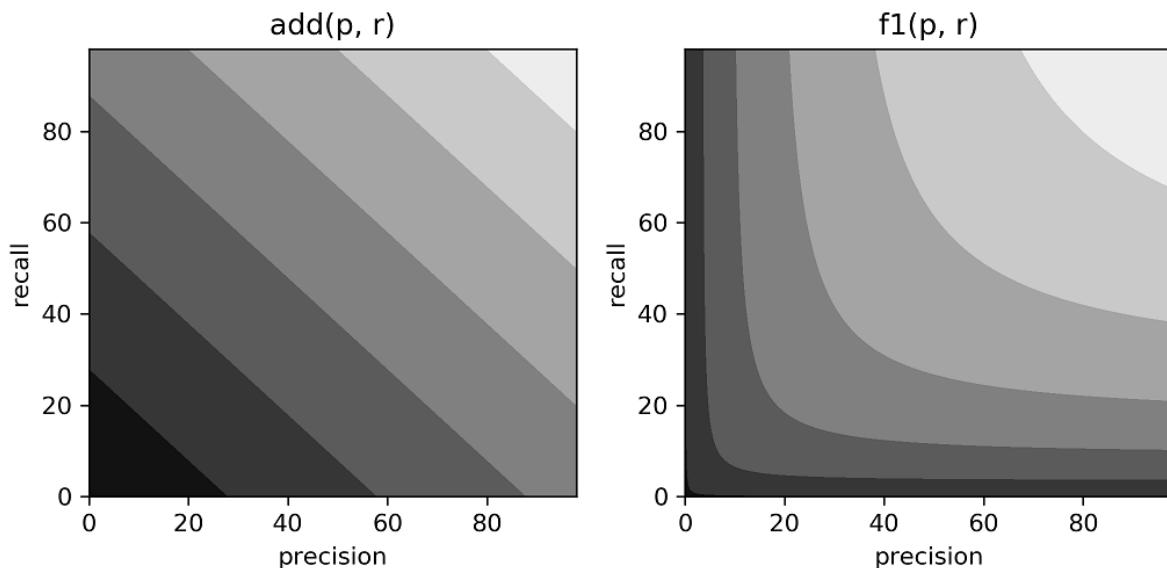


Figure 12.9 Computing final score with $\text{add}(p, r)$. Lighter values are closer to 1.0.

NOTE

What we are actually doing here is taking the arithmetic mean¹³¹ of the precision and recall, both of which are *rates* rather than countable scalar values. Taking the arithmetic mean of rates doesn't typically give meaningful results. The F1 score is another name for the harmonic mean¹³² of the two rates, which is a more appropriate way of combining those kinds of values.

A few things stand out. First, we can see a lack of a curve or elbow to the contour lines for

averaging. That's what lets our precision or recall skew to one side or the other! There will *never* be a situation where it doesn't make sense to maximize score by having 100% recall (the Roxie approach), and then eliminating whichever of the false positives that are easy to eliminate. That puts a floor on the addition score of 0.5 right out of the gate! Having a quality metric that is trivial to score at least 50% on doesn't feel right.

Contrast that with the F1 score; when recall is high but precision is low, trading off a lot of recall for even a little precision will move the score closer to that balanced sweet spot. There's a nice, deep elbow that is easy to slide into. That encouragement to have balanced precision and recall is what we want from our grading metric.

Let's say that we're still wanting a simpler metric, however, but one that doesn't reward skew at all. In order to correct for the weakness of addition, we might take the minimum of precision and recall:

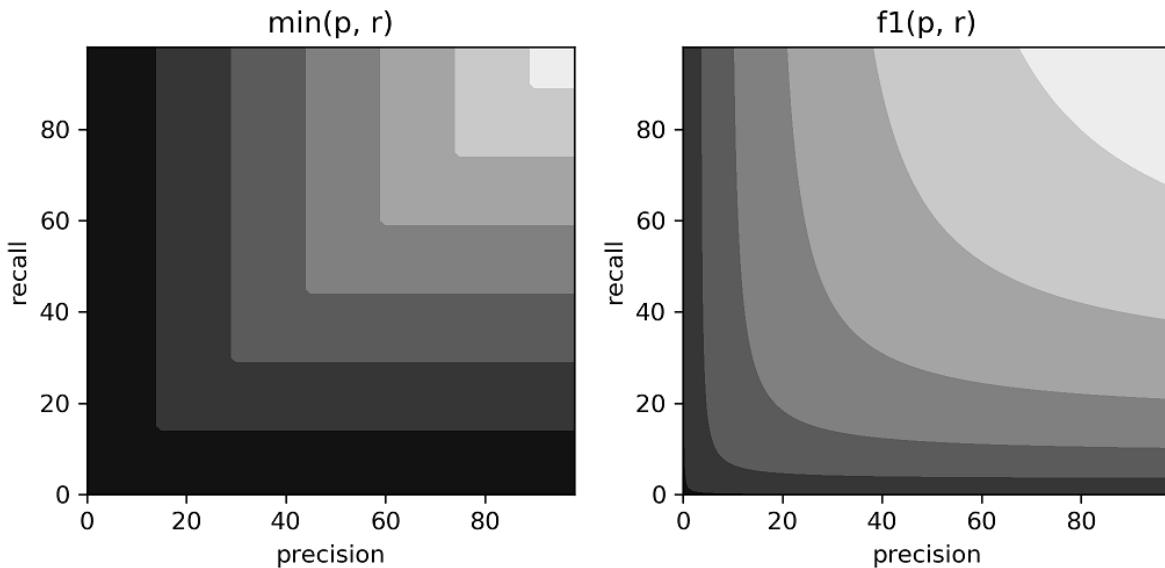


Figure 12.10 Computing final score with $\min(p, r)$.

This is nice, because if either value is zero, the score is also zero, and the only way to get a score of 1.0 is to have both values be 1.0. However, it still leaves something to be desired, since making a model change that increased the recall to from 0.7 to 0.9 while leaving precision constant at 0.5 wouldn't improve the score at all, nor would dropping recall down to 0.6! While this metric is certainly penalizing having an imbalance between precision and recall, it isn't capturing a lot of nuance about the two values. As we have seen, it's easy to trade one off for the other, simply by moving the classification threshold. We'd like our metric to reflect those trades.

Still, there might be something simpler still to find that would meet our goals. We could instead multiply the two values together:

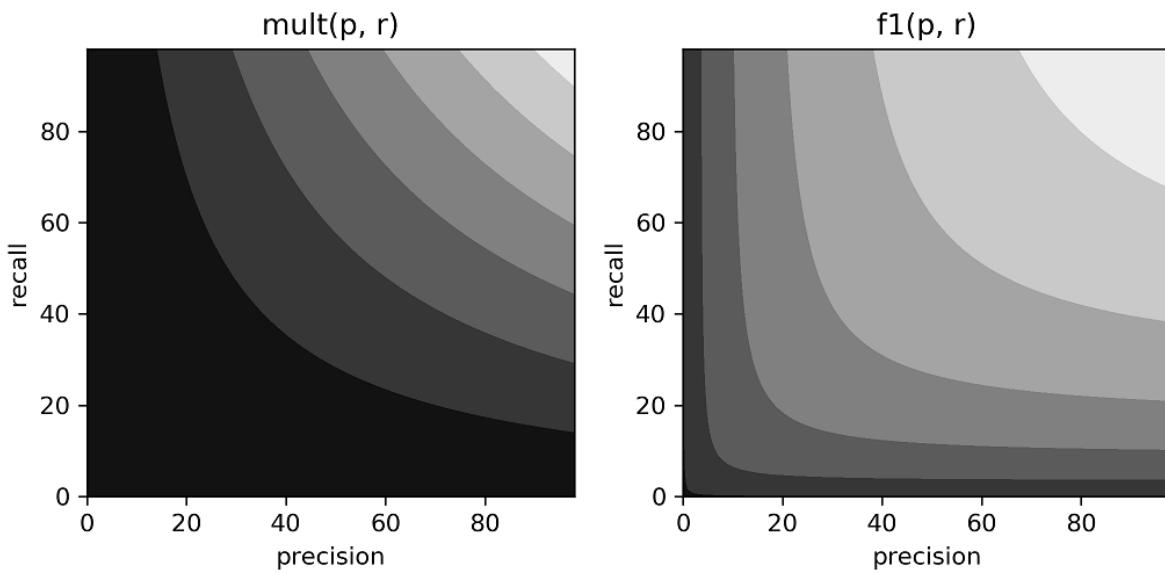


Figure 12.11 Computing final score with $\text{mult}(p, r)$.

NOTE Similarly here, we're taking the geometric mean¹³³ of two rates, which also doesn't produce meaningful results.

This keeps the nice property that if either value is zero, the score is zero, and a score of 1.0 means that both inputs are perfect. // TDE AZ: May want to build a table out to put the numbers side by side for easier reference (inversion matrix) It also favors a balanced trade off between precision and recall at low values, though when getting closer to perfect results it becomes more linear. That's not great, since we really need to push both up to have meaningful improvement at that point.

There's also the issue of having almost the entire quadrant from $(0, 0)$ to $(0.5, 0.5)$ be very close to zero. As we'll see, having a metric that's sensitive to changes in that region is going to be important, especially in the early stages of our model design.

While using multiplication as our scoring function is feasible (it doesn't have any immediate disqualifications the way the previous scoring functions did), we will be using the F1 score to evaluate our model's performance going forward.

UPDATING THE LOGGING OUTPUT TO INCLUDE PRECISION, RECALL, AND F1 SCORE

Now that we have our new metrics, adding them to our logging output is pretty straightforward. We'll include precision, recall, and F1 in our main logging statement for each of our training and validation sets:

Listing 12.4 p2ch12/training.py, line 316: class LunaTrainingApp.logMetrics

```
log.info(
    "E{} {:.8} {loss/all:.4f} loss, "
    + "{correct/all:-5.1f}% correct, "
    + "{pr/precision:.4f} precision, " ①
    + "{pr/recall:.4f} recall, "        ①
    + "{pr/f1_score:.4f} f1 score"     ①
).format(
    epoch_ndx,
    mode_str,
    **metrics_dict,
)
)
```

- ① Format string updated.

In addition, we'll also include exact numbers of our correctly identified and total samples for each of benign and malignant samples:

Listing 12.5 p2ch12/training.py, line 328: class LunaTrainingApp.logMetrics

```
log.info(
    "E{} {:.8} {loss/ben:.4f} loss, "
    + "{correct/ben:-5.1f}% correct ({ben_correct:} of {ben_count:})"
).format(
    epoch_ndx,
    mode_str + '_ben',
    ben_correct=ben_correct,
    ben_count=ben_count,
    **metrics_dict,
)
)
```

The new version of the malignant logging statement looks much the same.

12.2.5 How does our model perform with our new metrics?

Now that we have our shiny new metrics implemented, let's take them for a spin.

Listing 12.6 Bash shell session

```
$ ../../venv/bin/python -m p2ch12.training
Starting LunaTrainingApp...
...
E1 LunaTrainingApp

.../p2ch12/training.py:274: RuntimeWarning: invalid value encountered in double_scalars
    metrics_dict['pr/f1_score'] = 2 * (precision * recall) / (precision + recall) ①

E1 trn      0.0025 loss, 99.8% correct, 0.0000 prc, 0.0000 rcl, nan f1
E1 trn_ben  0.0000 loss, 100.0% correct (494735 of 494743)
E1 trn_mal  1.0000 loss, 0.0% correct (0 of 1215)

.../p2ch12/training.py:269: RuntimeWarning: invalid value encountered in long_scalars
    precision = metrics_dict['pr/precision'] = truePos_count / (truePos_count + falsePos_count)

E1 val      0.0025 loss, 99.8% correct, nan prc, 0.0000 rcl, nan f1
E1 val_ben  0.0000 loss, 100.0% correct (54971 of 54971)
E1 val_mal  1.0000 loss, 0.0% correct (0 of 136)
```

- ① The exact count and line number of these `RuntimeWarning` lines might be different from run to run.

Bummer. We've got some warnings, and given that we've got `nan` values, they're probably related to dividing by zero. Let's see what we can figure out.

First, since *none* of the malignant samples in the training set are getting classified as malignant, that means that both precision and recall are zero, which results in our F1 score calculation dividing by zero.

Second, for our validation set, our `truePos_count` and `falsePos_count` are both going to be zero due to *nothing* be flagged as malignant. Then, it follows that the denominator of our precision calculation is also zero. That makes sense, as that's where we're seeing another `RuntimeWarning`.

There are a handful of benign training samples that are classified as malignant (494735 of 494743 are classified as benign, so that leaves eight samples misclassified). While that might seem odd at first, recall that we are collecting our training results *throughout the epoch*, rather than using the model's end-of-epoch state like we do for the validation results. That means that the first batch is literally producing random results. A few of the samples from that first batch getting flagged as malignant isn't surprising.

NOTE

Due to both the random initialization of the network weights and the random ordering of training samples, individual runs will likely exhibit slightly different behavior. Having exactly reproducible behavior can be desirable, but is out of scope for what we're trying to do in part 2.

Well, that was somewhat painful. Switching to our new metrics resulted in going from "A+" to

"zero, if you're lucky," and if we're not lucky, the score is so bad that *it's not even a number*.

Ouch.

That said, in the long run this is good for us. We've known that our model's performance was garbage since the last chapter. If our metrics told us anything *but* that, that would point to a fundamental flaw in the metrics!

12.3 What does an ideal data set look like

Before we start crying into our cups over this sorry state of affairs, let's instead think about what we actually want our model to be doing. Recall Figure-10.3 earlier, and the following discussion on classification threshold. Getting better results by moving the threshold is going to have limited effectiveness — there's just too much overlap between the positive and negative classes to work with.¹³⁴ Instead, we want to see an image like this:

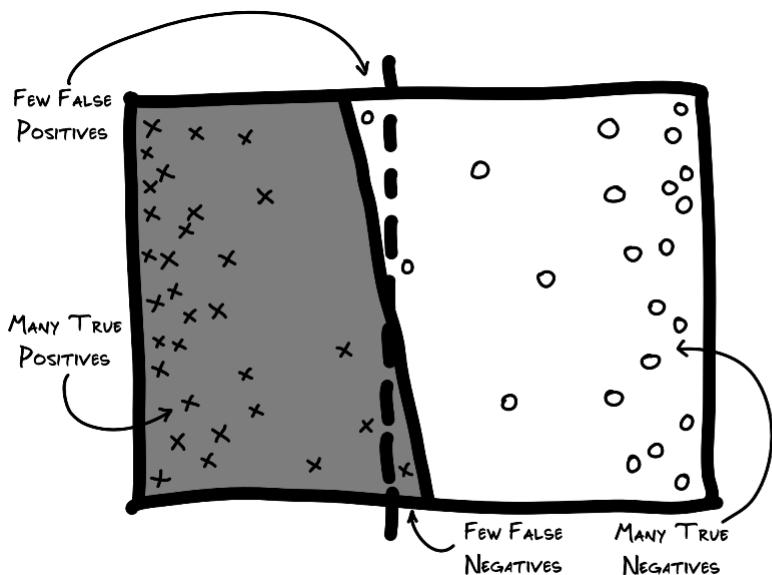


Figure 12.12 A well-trained model can cleanly separate data, making it easy to pick a classification threshold with few tradeoffs.

Here, our label threshold is nearly vertical. That's what we want, because it means that the label threshold and our classification threshold can line up reasonably well. Similarly, most of the samples are concentrated at either end of the diagram. Both of these things require that our data be easily separable, and that our model have the capacity to perform that separation. Our model currently has enough capacity that it's not the issue. Instead, let's take a look at our data.

Recall that our data is wildly imbalanced. There's a 400:1 ratio of malignant samples to benign ones. That's crushingly imbalanced! Here's what that looks like:

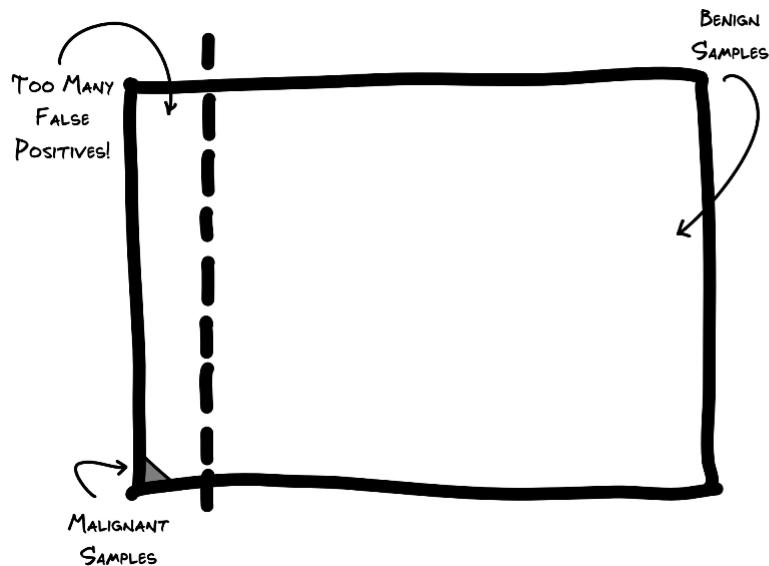


Figure 12.13 An imbalanced data set that understates the problem by half (this shows approximately a 200:1 ratio).

No wonder our malignant samples are getting lost in the crowd!

Now, let's be perfectly clear: when we're done, our model is going to be able to handle this kind of data imbalance just fine. We could probably even train the model all the way there without changing the balancing, assuming we were willing to wait for a gajillion epochs first.¹³⁵ But we're busy people with things to do, so rather than cook our GPU until the heat death of the universe, let's change the class balance we are training with.

12.3.1 Making the data look less like the actual and more like the "ideal"

The best thing to do would be to have there be relatively more malignant samples. During the initial epoch of training when we're going from randomized chaos to something more organized, having so few training samples be malignant means that they get drowned out.

The method by which this happens is somewhat subtle, however. Recall that since our network weights are randomized, the per-sample output of the network is also randomized (but clamped to the range [0-1]). Our loss function is `nn.CrossEntropyLoss`, which means that predictions numerically close to the correct label do not result in much change to the network, while predictions that are different from the correct answer are responsible for a much greater change to the weights.

Since the output is random when the model is initialized with random weights, we can assume that of our ~500k training samples¹³⁶, we'll have the following approximate groups:

1. 250k benign samples will be predicted to be benign (0.0 to 0.5), and result in at most a small change to the network weights towards predicting benign.
2. 250k benign samples will be predicted to be malignant (0.5 to 1.0), and result in a large

swing towards the network weights predicting benign.

3. 500 malignant samples will be predicted to be benign, and result in a swing towards the network weights predicting malignant.
4. 500 malignant samples will be predicted to be malignant, and result in almost no change to the network weights.

NOTE

Keep in mind that the actual prediction are real numbers between 0.0 and 1.0 inclusive, so these groups are not going to have strict delineations.

Here's the kicker, though: groups 1 and 4 can be of *any size* and they will continue to have close-to-zero impact on training. The only thing that matters is that groups 2 and 3 be able to counteract the pull of each other enough to prevent the network from collapsing to a degenerate "only output one thing" state. Since group 2 is 500 times larger than group 1, that implies that 499 training batches will be 100% benign, and will only pull all model weights towards predicting benign. That's what produces the degenerate behavior we've been seeing.

Instead, we'd like to have just as many malignant samples as benign ones. For the first part of training, then, half of both labels will be classified incorrectly, meaning that groups 2 and 3 should be roughly equal size. That would result in the tug-of-war evening out, and the model having a chance of learning to discriminate between the two classes. Since our LUNA data has only a small, fixed number of malignant samples, we'll have to settle for taking the samples that we have and presenting them more often during training.

SIDEBAR**Discrimination**

Here, we define discrimination as "the ability to separate two classes from each other." Building and training a model that can tell malignant tumors from benign nodules is the entire point of what we're doing in part 2.

There are other definitions of discrimination that are more problematic. While out of scope for the discussion of work in part 2, there is a larger issue with models trained from real-world data. If that real-world data set is collected from sources that have a real-world-discriminatory bias (for example, racial bias in arrest and conviction rates, or anything collected from social media) and that bias is not corrected for during data set preparation or training, then the resulting model will continue to exhibit the same biases present in the training data. Just as in humans, racism is learned.

This means that almost any model trained from Internet-at-large data sources will be compromised in some fashion, unless there was extreme care taken to scrub those biases from the model. Note that like our goal here in part 2, this is considered an unsolved problem.

Recall our professor from last chapter that has a final with 99 false answers and 1 true one. The next semester, s/he was told "you should have a more even balance of true and false answers,"

and so s/he decided to add a midterm with 99 true answers and 1 false one. "Problem solved!" Clearly, the correct approach is to intermix true and false answers in a way that doesn't allow the students to exploit the larger structure of the tests to answer things correctly.

While a student would pick up on a pattern of "odd questions are true, even questions are false", the batching system used by PyTorch doesn't allow the model to "notice" or utilize that kind of pattern.

We will not be doing any balancing for validation, however. Our model needs to function well in the real world, and the real world is imbalanced (after all, that's where we got the data from!).

SAMPLERS CAN RESHAPE DATA SETS

One of the optional arguments to `DataLoader` is `sampler=...`. This allows the `DataLoader` to override the iteration order native to the `Dataset` passed in, and instead shape, limit, or re-emphasize the underlying data as desired. This can be incredibly useful when working with a `Dataset` that isn't under your control. Taking a public data set and reshaping it to meet your needs is far less work than reimplementing that data set from scratch.

The downside is that many of the mutations that one could accomplish with samplers require that we break encapsulation of the underlying `Dataset`. For example, let's assume we have a dataset like CIFAR-10¹³⁷ that consists of ten classes, equally weighted, and we want to instead have one class (say, "airplane") now comprise 50% of all of the training images. We could decide to use the `WeightedRandomSampler`¹³⁸ and weight each of the "airplane" sample indexes higher, but constructing the `weights` argument requires that we know in advance which indexes are airplanes.

As we recall, the `Dataset` API only specifies that subclasses provide `len` and `getitem`, but there is nothing direct we can use to ask "which samples are airplanes?" We'd either have to load up every sample beforehand to inquire about the class of that sample, or we'd have to break encapsulation and hope that the information we need is easily obtained from looking at the internal implementation of the `Dataset` subclass.

Since neither of those options is particularly ideal in cases where we have control over the `Dataset` directly, the code for part 2 will be implementing any needed data shaping inside the `Dataset` subclasses instead of relying on an external sampler.

12.3.2 Changes to training.py, dset.py to balance benign and malignant samples

We are going to directly change our `LunaDataset` to present a balanced, one-to-one ratio of benign and malignant samples for training. We will keep separate lists of benign training samples and malignant training samples, and alternate returning samples from each of those two lists. This will prevent the degenerate behavior of the model scoring well by simply answering "false" to every sample presented. In addition, the malignant and benign classes will be intermixed, so that the weight updates are forced to discriminate between the classes.

Let's add a `ratio_int` to `LunaDataset` that will control the label for the Nth sample, as well as keep track of our samples separated by label:

Listing 12.7 p2ch12/dsets.py, line 199: class LunaDataset

```
class LunaDataset(Dataset):
    def __init__(self,
                 val_stride=0,
                 isValSet_bool=None,
                 ratio_int=0,
                 ):
        self.ratio_int = ratio_int
        # ... line 210
        self.benign_list = [nt for nt in self.noduleInfo_list if not nt.isMalignant_bool]
        self.malignant_list = [nt for nt in self.noduleInfo_list if nt.isMalignant_bool]
        # ... line 241

    def shuffleSamples(self): ①
        if self.ratio_int:
            random.shuffle(self.benign_list)
            random.shuffle(self.malignant_list)
```

- ① We will call this at the top of each epoch to randomize the order of samples being presented.

With this, we now have dedicated lists for each label. Using these lists it becomes much easier to return the label we want for a given index into the dataset. In order to make sure we're getting the indexing right, we should sketch out the ordering that we want. Let's assume a `ratio_int` of 2, meaning a 2:1 ratio of benign to malignant samples. That would mean every third index should be malignant.

DS Index	0	1	2	3	4	5	6	7	8	9	...
Label	M	b	b	M	b	b	M	b	b	M	
Mal Index	0		1			2			3		
Ben Index	0	1		2	3		4	5			

The relationship between the Dataset index and the malignant index is simple; divide the DS index by three and then round down. The benign index is slightly more complicated, in that you have to subtract one from the DS index, then subtract the most recent malignant index as well.

Implemented in our `LunaDataset` class, that looks like:

Listing 12.8 p2ch12/dsets.py, line 262: class LunaDataset.{uu}getitem{uu}

```
def __getitem__(self, ndx):
    if self.ratio_int: ①
        malignant_ndx = ndx // (self.ratio_int + 1)

        if ndx % (self.ratio_int + 1): ②
            benign_ndx = ndx - 1 - malignant_ndx
            benign_ndx %= len(self.benign_list) ③
            nodule_tup = self.benign_list[benign_ndx]
        else:
            malignant_ndx %= len(self.malignant_list) ③
            nodule_tup = self.malignant_list[malignant_ndx]
    else:
        nodule_tup = self.noduleInfo_list[ndx]
```

- ① A ratio_int of zero means use the native balance.
- ② Non-zero remainder means that this should be a benign sample.
- ③ Overflow results in wrap-around.

That can get a little hairy, but if you desk check it out, it will make sense. Keep in mind that with a low ratio, we'll run out of malignant samples before exhausting the Dataset. We take care of that by taking the modulus of the `malignant_ndx` before indexing into `self.malignant_list`. While the same should never happen with `benign_ndx`, we do the modulus anyway, just in case we later decide to make some change that might cause it to overflow.

We'll also make a change to our dataset's length. While this isn't strictly necessary, it's nice to speed up individual epochs. What we're going to do is hardcode our `{uu}len{uu}` to be 200k.

Listing 12.9 p2ch12/dsets.py, line 256: class LunaDataset.{uu}len{uu}

```
def __len__(self):
    if self.ratio_int:
        return 200000
    else:
        return len(self.noduleInfo_list)
```

We're not tied to a specific number of samples any longer, and presenting "a full epoch" doesn't really make sense when we would have to repeat malignant samples many, many times to present a balanced training set.

By picking 200k samples, we reduce the time between starting a training run and seeing results (faster feedback is always nice!), and we give ourselves a nice, clean number of samples-per-epoch. Feel free to adjust the length of an epoch to meet your needs.

For completeness, we also add a command-line parameter:

Listing 12.10 p2ch12/training.py, line 31: class LunaTrainingApp

```
class LunaTrainingApp:  
    def __init__(self, sys_argv=None):  
        # ... line 52  
        parser.add_argument('--balanced',  
            help="Balance the training data to half benign, half malignant.",  
            action='store_true',  
            default=False,  
        )
```

- ① Here we rely on python's `True` being convertible to a `1`.

And then pass that parameter into the `LunaDataset` constructor:

Listing 12.11 p2ch12/training.py, line 137: class LunaTrainingApp.initTrainDl

```
def initTrainDl(self):  
    train_ds = LunaDataset(  
        val_stride=10,  
        isValSet_bool=False,  
        ratio_int=int(self.cli_args.balanced), ①  
    )
```

- ① Here we rely on python's `True` being convertible to a `1`.

I think we're set; let's run it!

12.3.3 Contrasting training with a balanced `LunaDataset` to previous runs

as a reminder, unbalanced training had

Listing 12.12 Bash shell session

```
$ python -m p2ch12.training  
...  
E1 LunaTrainingApp  
E1 trn      0.0024 loss,  99.8% correct  
E1 trn_ben  0.0000 loss, 100.0% correct  
E1 trn_mal  0.9953 loss,   0.0% correct  
E1 val      0.0025 loss,  99.7% correct  
E1 val_ben  0.0000 loss, 100.0% correct  
E1 val_mal  0.9972 loss,   0.0% correct
```

but run with `--balanced` and we get

Listing 12.13 Bash shell session

```
$ python -m p2ch12.training --balanced
...
E1 LunaTrainingApp
E1 trn      0.1972 loss, 72.1% correct, 0.7485 prc, 0.6669 rcl, 0.7053 f1
E1 trn_ben  0.1897 loss, 77.6% correct (38798 of 50000)
E1 trn_mal  0.2048 loss, 66.7% correct (33344 of 50000)
E1 val      0.0499 loss, 94.3% correct, 0.0363 prc, 0.8603 rcl, 0.0697 f1
E1 val_ben  0.0498 loss, 94.4% correct (51866 of 54971)
E1 val_mal  0.1045 loss, 86.0% correct (117 of 136)
```

This seems much better! We've given up about 5% correct answers on the benign samples to gain 86% correct malignant answers. Seems like we are back up into a solid B range again!¹³⁹

As in the last chapter, however, this is deceptive. Since there are 400 times as many benign samples as malignant ones, even getting just 1% wrong means that we'd be incorrectly classifying benign samples as malignant four times more often than there are actually malignant samples in total!

Still, this is clearly better than we had been doing previously, and is much better than a random coin-flip (to say nothing of the outright wrong behavior from before). In fact, we've even crossed over into (almost) legitimately useful in real-world scenarios. Recall our overworked radiologist poring over each and every speck of a CT? Well, now we've got something that can do a somewhat reasonable job of screening out 95% of the false positives. That's a huge help, since it translates into something like a 10-fold increase in productivity for the machine-assisted human.

Of course, there's still that pesky issue of the 14% of malignant samples that got missed that we should probably deal with. Perhaps some additional epochs of training would help? Let's see:

Listing 12.14 Bash shell session

```
$ python -m p2ch12.training --balanced --epochs 20
...
E2 LunaTrainingApp
E2 trn      0.1054 loss,  87.4% correct, 0.8947 prc, 0.8488 rcl, 0.8711 f1
E2 trn_ben  0.0858 loss,  90.0% correct (45005 of 50000)
E2 trn_mal  0.1250 loss,  84.9% correct (42439 of 50000)
E2 val      0.0713 loss,  91.1% correct, 0.0239 prc, 0.8750 rcl, 0.0464 f1
E2 val_ben  0.0713 loss,  91.1% correct (50101 of 54971)
E2 val_mal  0.1038 loss,  87.5% correct (119 of 136)
...
E5 LunaTrainingApp
E5 trn      0.0545 loss,  93.8% correct, 0.9431 prc, 0.9327 rcl, 0.9379 f1
E5 trn_ben  0.0501 loss,  94.4% correct (47188 of 50000)
E5 trn_mal  0.0589 loss,  93.3% correct (46637 of 50000)
E5 val      0.0551 loss,  93.8% correct, 0.0359 prc, 0.9265 rcl, 0.0691 f1
E5 val_ben  0.0550 loss,  93.8% correct (51588 of 54971)
E5 val_mal  0.0695 loss,  92.6% correct (126 of 136)
...
E10 LunaTrainingApp
E10 trn     0.0173 loss,  97.9% correct, 0.9773 prc, 0.9815 rcl, 0.9794 f1
E10 trn_ben 0.0198 loss,  97.7% correct (48860 of 50000)
E10 trn_mal 0.0149 loss,  98.1% correct (49073 of 50000)
E10 val     0.0190 loss,  97.8% correct, 0.0939 prc, 0.8971 rcl, 0.1700 f1
E10 val_ben 0.0189 loss,  97.9% correct (53794 of 54971)
E10 val_mal 0.0893 loss,  89.7% correct (122 of 136)
...
E20 LunaTrainingApp
E20 trn     0.0086 loss,  99.0% correct, 0.9884 prc, 0.9912 rcl, 0.9898 f1
E20 trn_ben 0.0102 loss,  98.8% correct (49416 of 50000)
E20 trn_mal 0.0071 loss,  99.1% correct (49559 of 50000)
E20 val     0.0114 loss,  98.7% correct, 0.1457 prc, 0.8676 rcl, 0.2495 f1
E20 val_ben 0.0111 loss,  98.7% correct (54279 of 54971)
E20 val_mal 0.1269 loss,  86.8% correct (118 of 136)
```

Ugh, that's a lot of text to scroll past to get to the numbers we're interested in. Let's power through, and focus on the `val_mal` xx.x% correct numbers. After our second epoch we were at 87.5%, on our fifth epoch we peaked with 92.6%, and then by epoch twenty we'd dropped down to 86.8%; that's *below* our second epoch!

NOTE

As mentioned before, expect each run to have unique behavior due to random initialization of network weights, and random selection and ordering of training samples per epoch.

The training set numbers don't seem to be having the same problem. Benign training samples are classified correctly 98.8% of the time, and malignant ones are 99.1% correct.

That's a clear sign of overfitting.

/// TODO LUCA: we've touched upon overfitting in Part 1. We can make a reference here. ///
TODO elis: fix this transition

12.4 Revisiting the problem of over-fitting

We touched on the concept of over-fitting in chapter 5, and now it's time to take a closer look at how to handle this common situation. Our goal with training a model is to teach it to recognize the *general properties* of the classes we are interested in, as expressed in our data set. Those general properties are present in some or all samples of the class, and can be *generalized* and used to predict samples that haven't been trained on. When the model starts to learn *specific properties* of the training set, over-fitting occurs, and the model starts to lose the ability to generalize.

In case that's a bit too abstract, let's use an analogy.

12.4.1 An over-fit face-to-age prediction model

Let's pretend we have a model that takes an image of a human face as input and outputs a predicted age in years. A good model would pick up on age signifiers like wrinkles, gray hair, hairstyle, clothing choices, and similar, and use those to build a general model of what different ages look like. When presented with a new picture it will consider things like "conservative haircut" and "reading glasses" and "wrinkles" to conclude "around 65 years old."

An over-fit model, by contrast, will instead remember specific people by remembering identifying details. "That haircut and those glasses mean it's Frank. He's 62.8 years old." "Oh, that scar means it's Harry. He's 39.3." And so on. When shown a new person, the model wouldn't recognize the person, and would have absolutely no idea what age to predict.

Even worse, if shown a picture of Frank Jr. (the spittin' image of his dad, at least when he's wearing his glasses!), the model would say "I think that's Frank. He's 62.8 years old." Never mind that Junior is 25 years younger!

Over-fitting is usually due to having too few training samples when compared to the ability of the model to just memorize the answers. The median human can memorize the birthdays of their immediate family, but would have to resort to generalizations when predicting the ages of any group larger than a small village.

12.4.2 Detecting over-fitting

Generally, if your model's performance is improving on your training set, while getting worse on your validation set, the model has started over-fitting. We have already seen evidence of this — recall the malignant samples being classified exactly-wrong were diminishing for our training set, but increasing for our validation set. That's over-fitting.

Likewise, we can see similar trends when we look at both our percent correctly classified graph:

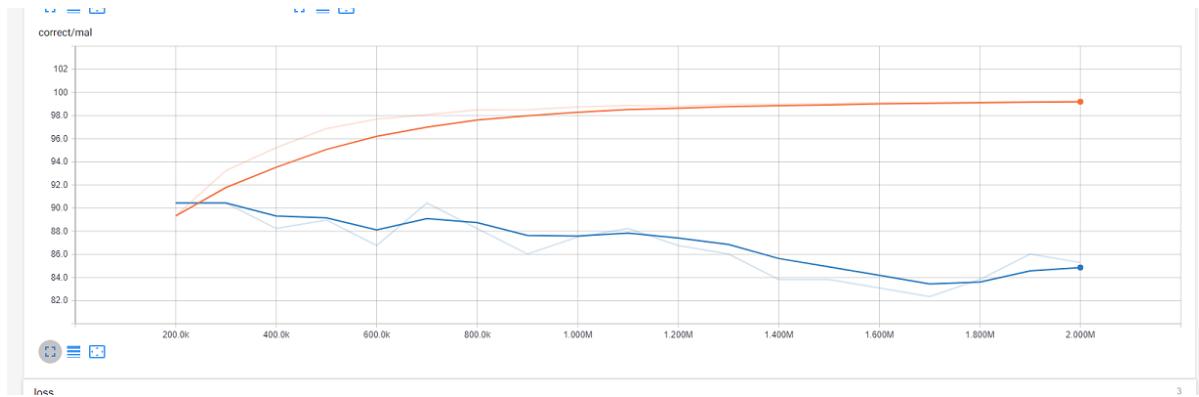


Figure 12.14 The main TensorBoard data display area.

And we can see the same in our loss graph:

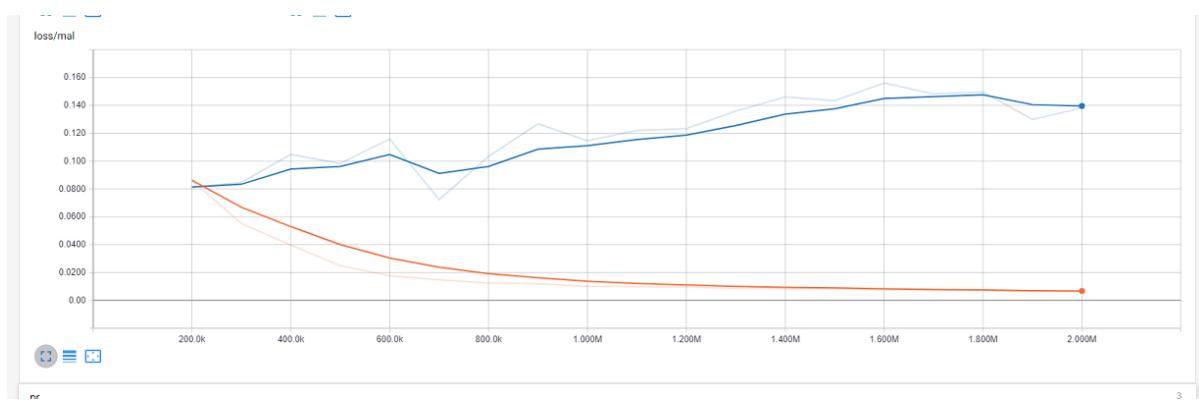


Figure 12.15 The main TensorBoard data display area.

When the losses for our train set and validation set are diverging, that's almost certainly a sign of over-fitting.

Also note that we are not seeing the same divergent behavior for our benign samples. That's because we have 400 times more benign samples, so it's much, much harder for the model to remember individual details. Our malignant training set only has 1215 samples, though. While we repeat those samples multiple times, that doesn't make them harder to memorize. What's happening is the model is shifting from generalized principles to essentially memorizing quirks of those 1215 samples, and claiming that anything that's not one of those few samples is benign. Now, there clearly is still some generalization going on, since we are still classifying over 80% of the malignant validation set correctly. We just need to change up how we're training the model so that our training set and validation set are both trending in the right direction.

WRITING HISTOGRAMS TO TENSORBOARD

Histograms give us a view into the data that lets us see if and where there are numerical clusters in the spread of our data. What we're going to use this for here is to plot out where each of the benign samples is landing in terms of our predicted classification (and the same for malignant, but we'll focus on benign for right now). Since the vast majority of the benign samples will have a predicted classification of something very close to 0.0, we are going to exclude those samples from our histogram. They're not interesting since we're getting them right. We'll refer to any sample that has a prediction within 0.01 of the correct label to be "perfect" from here on out (even though that's not strictly true, since only predictions of 0.0 and 1.0 would actually be perfect).

NOTE

In general, shaping the data you display is an important part of getting quality information from the data. Getting the right things on-screen will typically require some iteration of careful thought and experimentation. Don't hesitate to tweak what you're showing, but also take care to remember if you change the definition of a particular metric without changing the name. It can be easy to compare apples to oranges unless you're disciplined about naming schemes or removing now-invalid runs of data.

We construct our `benHist_mask` by requiring that both the label be benign, and the prediction be greater than 0.01.

Listing 12.15 p2ch12/training.py, line 362: class LunaTrainingApp.logMetrics

```
bins = [x/50.0 for x in range(51)]  
  
benHist_mask = benLabel_mask & (metrics_t[METRICS_PRED_NDX] > 0.01)  
malHist_mask = malLabel_mask & (metrics_t[METRICS_PRED_NDX] < 0.99)  
  
if benHist_mask.any():  
    writer.add_histogram(  
        'is_ben',  
        metrics_t[METRICS_PRED_NDX, benHist_mask],  
        self.totalTrainingSamples_count,  
        bins=bins,  
    )  
if malHist_mask.any():  
    writer.add_histogram(  
        'is_mal',  
        metrics_t[METRICS_PRED_NDX, malHist_mask],  
        self.totalTrainingSamples_count,  
        bins=bins,  
    )
```

Once that's done, we can call `writer.add_histogram` with a label, the data, and the `global_step` step set to our number of training samples presented; this is similar to the scalar call earlier. We also pass in the `bins` set to a fixed scale.

NOTE

The default bins parameter on some versions of TensorboardX results in histograms that are incredibly wide, which makes for a graph that isn't particularly useful for data in the range [0, 1]. Make sure you set the bins appropriately for your use case.

Once that's done, we can take a look at our prediction distribution for benign samples and how it evolves over each epoch:

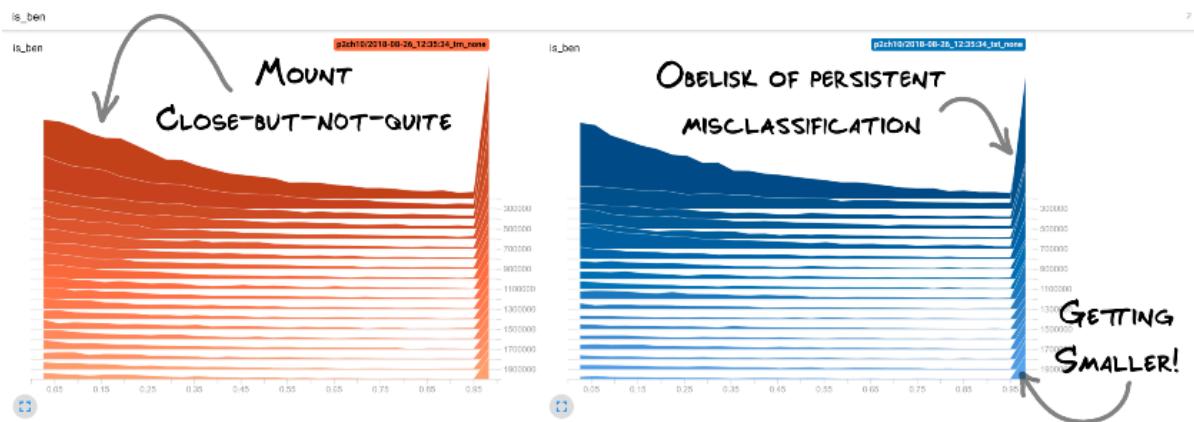


Figure 12.16 TensorBoard histogram display of the prediction for benign samples. The numerical values are not important.

There are two main features of both of these histograms that are striking. First, is the mountain of samples on the left that start off *close* to being classified perfectly that gets smaller as the epochs increase. Since we screen out the perfectly classified samples (which would be an astronomical spike on the left edge) and we don't see those samples shifting around to anywhere else on our histogram (nothing else is growing), we can safely assume that they're transitioning from close to perfect. Great!

The second feature is the spike of benign samples that persist in being labeled as incredibly malignant. The number of them reduces over time, but even 20 epochs on it's still clearly an issue. Keep this gradual reduction over time in mind as you finish out the chapter; it will be an important clue before we're done.

Now let's take a look at a similar graph for the malignant samples:

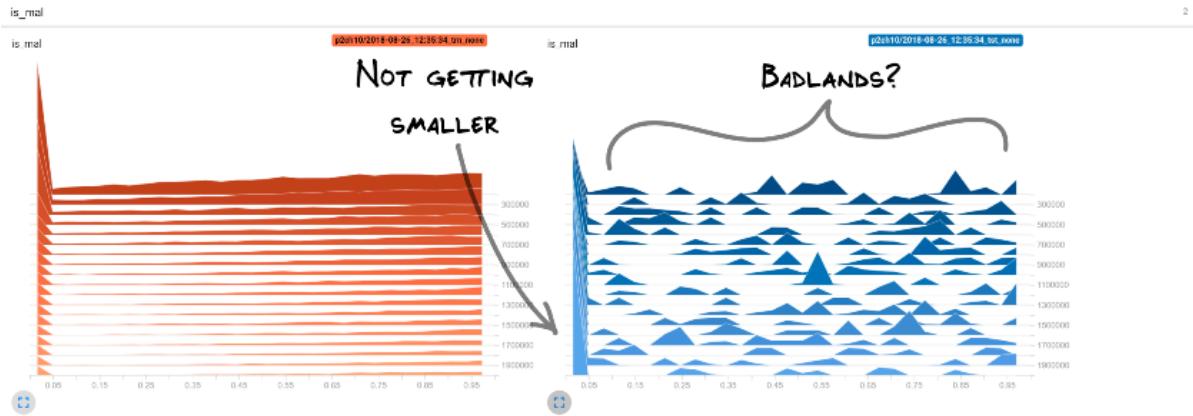


Figure 12.17 TensorBoard histogram display of the prediction for malignant samples. Some of the trends provide clues that training is not progressing well. The numerical values are not important.

This is reversed left/right from above, due to wanting to push the predictions for malignant samples towards 1.0 instead of 0.0. Let's focus on the training set for a moment (on the left in Figure-10.6). The training set mountain near almost-perfect still exists and shrinks for several epochs. We see a drop-off in our rate of change around 700k or 800k samples presented, so around the end of epoch 8. Similarly to the benign data above, the spike at exactly-wrong classification continues to shrink.

Unfortunately, the validation data shown on the right does not have the same pattern as the others. While there's still a spike of exactly-wrong results, we don't see the mountain of close results, nor are the in-between results obviously reducing. Even worse, the spike of exactly-wrong results is getting larger as we add epochs!

We need to take our nodule data and do the equivalent of doubling our face-to-age data by putting novelty glasses with a fake nose and mustache on copies of every face. We're going to augment our data.

12.5 Data Augmentation

We *augment* a data set by applying synthetic alterations to individual samples, resulting in a new data set with an effective size that is larger than the original. The typical goal is for the alterations to result in a synthetic sample that remains representative of the same general class as the source sample, but cannot be trivially memorized alongside the original. When done properly, this augmentation can increase the training set size beyond what the model is capable of memorizing, resulting in the model being forced to increasingly rely on generalization, which is exactly what we want.

Of course, not all augmentations are equally useful. Going back to our example of a face-to-age prediction model, we could trivially change the four corner pixels of each image to a random RGB value, which would result in a data set four billion times larger the original. Of course, this

wouldn't be particularly useful, since the model can pretty trivially learn to ignore the image corners, and the rest of the image remains as easy to memorize as the single, un-augmented original image. Contrast that approach with flipping the image left-to-right. Doing so would only result in a data set twice as large as the original, but each image would be quite a bit more useful for training purposes. The general properties of aging are not correlated left-to-right, so a mirrored images remains representative. Similarly, it's rare for facial pictures to be symmetrical, so a mirrored version is unlikely to be trivially memorized alongside the original.

12.5.1 Specific Data Augmentation Techniques

We are going to implement five specific types of data augmentation. Our implementation is going to allow us to experiment with any or all of them, individually or in aggregate. The five techniques are:

- Mirroring the image up-down, left-right, and/or front-back
- Shifting the image around by a few pixels
- Scaling the image up or down
- Rotating the image around the head-foot axis
- Adding noise to the image

For each technique, we want to make sure that our approach is going to maintain the training sample's representative nature, while being different enough that the sample is useful to train with.

We are going to define a function `foo` that will be responsible for taking our standard chunk-of-CT-with-nodule-inside and modifying it. The main approach is that we are going to define an affine transformation matrix¹⁴⁰ and use that with the PyTorch `affine_grid`¹⁴¹ and `grid_sample`¹⁴² functions to resample our nodule.

Listing 12.16 p2ch12/dsets.py, line 134: def getCtAugmentedNodule

```
def getCtAugmentedNodule(
    augmentation_dict,
    series_uid, center_xyz, width_irc,
    use_cache=True):
    if use_cache:
        ct_chunk, center_irc = getCtRawNodule(series_uid, center_xyz, width_irc)
    else:
        ct = getCt(series_uid)
        ct_chunk, center_irc = ct.getRawNodule(center_xyz, width_irc)

    ct_t = torch.tensor(ct_chunk).unsqueeze(0).unsqueeze(0).to(torch.float32)
```

Here, we are first going to obtain our `ct_chunk`, either from the cache or directly by loading the CT (something that will come in handy once we are creating our own nodule centers), and then converting it to a tensor. Next we have the affine grid and sampling code:

Listing 12.17 p2ch12/dsets.py, line 146: def getAugmentedNodule

```
transform_t = torch.eye(4).to(torch.float64)
# ... ①
# ... line 179
affine_t = F.affine_grid(
    transform_t[:3].unsqueeze(0).to(torch.float32),
    ct_t.size(),
)
augmented_chunk = F.grid_sample(
    ct_t,
    affine_t,
    padding_mode='border'
).to('cpu')
# ... line 196
return augmented_chunk[0], center_irc
```

- ① Modifications to `transform_tensor` will go here.

Without anything additional, this function won't do much. Let's see what it takes to add in some actual transforms.

MIRRORING

When mirroring a sample, we keep the pixel values exactly the same, and only change the orientation of the image. Since there's no strong correlation between cancer growth and left-right or front-back, we should be able to flip those without changing the representative nature of the sample. The index axis (referred to as Z in patient coordinates) corresponds to the direction of gravity in an upright human, however, so there's a possibility of difference in the top and bottom of a tumor. We are going to assume it's fine, since quick visual investigation doesn't show any gross bias. Were we working towards a clinically relevant project, we'd need to confirm that assumption with an expert.

Listing 12.18 p2ch12/dsets.py, line 149: def getAugmentedNodule

```
for i in range(3):
    if 'flip' in augmentation_dict:
        if random.random() > 0.5:
            transform_t[i,i] *= -1
```

The `grid_sample` function maps the range [-1, 1] to the extents of both the old and new tensors (the rescaling happens implicitly if the sizes are different). This range mapping means that to mirror the data, all we need to do is multiply the relevant element of the transformation matrix by -1.

SHIFTING BY A RANDOM OFFSET

Shifting the nodule around shouldn't make a huge difference, since convolutions are translation-independent. What will make a difference is that the offset might not be an integer number of voxels; instead the data will be resampled using trilinear interpolation, which can introduce some slight blurring.

Listing 12.19 p2ch12/dsets.py, line 149: def getCtAugmentedNodule

```
for i in range(3):
    # ... line 154
    if 'offset' in augmentation_dict:
        offset_float = augmentation_dict['offset']
        random_float = (random.random() * 2 - 1)
        transform_t[3,i] = offset_float * random_float
```

Note that our 'offset' parameter is the maximum offset expressed in the same scale as the [-1, 1] range that grid sample function expects.

SCALING

Scaling the image slightly is very similar to mirroring and shifting.

Listing 12.20 p2ch12/dsets.py, line 149: def getCtAugmentedNodule

```
for i in range(3):
    # ... line 159
    if 'scale' in augmentation_dict:
        scale_float = augmentation_dict['scale']
        random_float = (random.random() * 2 - 1)
        transform_t[i,i] *= 1.0 + scale_float * random_float
```

Since `random_float` is converted to be in the range [-1, 1], it doesn't actually matter if we add or subtract `scale_float * random_float` from 1.0.

ROTATING

Rotation is the first augmentation technique we're going to use where we have to carefully consider our data to make sure we don't make a mistake that breaks our sample with a conversion that makes it no longer representative. Recall that our CT slices have uniform spacing along the rows and columns (X and Y axes), but the index (or Z) direction the voxels are non-cubic. That means that we can't treat those axes as interchangeable.

One option is to resample our data so that our resolution along the index axis is the same as along the other two, but that's not a true solution because the data along that axis would be very blurry and smeared. Even if we interpolate more voxels, the fidelity of the data would remain poor.

Instead, we are going to treat that axis as special, and confine our rotations to the X-Y plane.

- be careful with caching

12.5.2 Seeing the improvement from data augmentation

TODO: finish this section.

Run the following training scripts, and then observe in TensorBoard.

```
$ .venv/bin/python -m p2ch12.training.2 p2ch12.training --epochs 10 \
    --balanced sanity-bal

$ .venv/bin/python -m p2ch12.training.2 p2ch12.training --epochs 10 \
    --balanced --augment-flip    sanity-bal-flip

$ .venv/bin/python -m p2ch12.training.2 p2ch12.training --epochs 10 \
    --balanced --augment-shift  sanity-bal-shift

$ .venv/bin/python -m p2ch12.training.2 p2ch12.training --epochs 10 \
    --balanced --augment-scale  sanity-bal-scale

$ .venv/bin/python -m p2ch12.training.2 p2ch12.training --epochs 10 \
    --balanced --augment-rotate sanity-bal-rotate

$ .venv/bin/python -m p2ch12.training.2 p2ch12.training --epochs 10 \
    --balanced --augment-noise   sanity-bal-noise

$ .venv/bin/python -m p2ch12.training.2 p2ch12.training --epochs 10 \
    --balanced --augmented sanity-bal-aug
```

12.6 Conclusion

We spent a lot of time and energy this chapter reformulating how we think about our model’s performance. It’s easy to be misled by poor methods of evaluation, and having a strong intuitive understanding of the factors that feed into evaluating a model well is crucial. Once those fundamentals are internalized, it’s much easier to spot when we’re being led astray.

We’ve also learned about how to deal with data sources that aren’t well-populated enough. Being able to synthesize representative training samples is incredibly useful. The situations where one has too much training data are rare indeed!

Now that we have a classifier that is performing reasonably, we’ll next turn our attention to automatically finding nodules to classify. Chapter 13 will start there, and then in chapter 14 we will feed those nodules back into our classifier we’ve developed here.

12.7 Exercises

1. The F1 score can be generalized out to support values other than 1.
 - A. Read en.wikipedia.org/wiki/F1_score and implement F2 and F0.5 scores.
 - B. Determine which of F1, F2, and F0.5 makes the most sense for this project. Track that value and compare and contrast with the F1 score.¹⁴³
2. Implement a `WeightedRandomSampler` approach to balancing the benign and malignant training samples for a `LunaDataset` with `ratio_int` set to 0.
 - A. How did you get the required information about the class of each sample?
3. Experiment with different class balancing schemes.
 - A. What ratio results in the best score after two epochs? After 20?
 - B. What if the ratio is a function of the `epoch_ndx`?
4. Experiment with different data augmentation approaches.
 - A. Can any of the existing approaches be made more aggressive (noise, offset, etc.)?
 - B. Research data augmentation that other projects have used. Are any applicable here?
5. Change the initial normalization from `nn.BatchNorm` to something custom, and retrain the model.
 - A. Can you get better results using a fixed normalization?
 - B. What normalization offset and scale make sense?
 - C. Do non-linear normalizations like square roots help?
6. What other kinds of data can Tensorboard display besides the ones we've covered here?
 - A. Can you have it display information about the weights of your network?
 - B. What about intermediate results from running your model on a particular sample?
 - i. Does having the backbone of the model wrapped in an instance of `nn.Sequential` hinder this effort?

12.8 Summary

- A binary label and a binary classification threshold combine to partition the data set into four quadrants; true positives, true negatives, false negatives, and false positives. These four quantities provide the basis for our improved performance metrics.
- Recall is the ability of a model to maximize true positives. Roxie had high recall by barking at just about everything.
- Precision is the ability of a model to minimize false positives. Preston had high recall by only barking when a burglar stepped on him.
- The F1 score combines precision and recall into a single metric that describes model performance. We use F1 score to determine if a change to training or the model results in a practical improvement or not.
- Balancing the training set to have an equal number of benign and malignant samples results in the model training well (defined as having a positive and increasing F1 score).
- Data augmentation takes existing organic data samples and modifies them such that the resulting augmented sample is non-trivially different from the original, but remains representative of samples of the same class. This allows additional training without over-fitting in situations where data is limited.
- Common data augmentation strategies include changes in orientation, mirroring, deformation, and adding noise. Depending on the project, other more specific strategies may also be relevant.

13

Using Segmentation To Find Suspected Nodules

This chapter covers:

- Chapter 13 will implement step 2, segmentation.
- a bulleted list
- of high-level topics
- that you teach
- in the chapter

In the last four chapters, we have accomplished a lot. We've learned about CT scans and lung tumors, datasets and data loaders, and metrics and monitoring. We have also *applied* many of the things we learned in part 1, and have a working classifier. We are still operating in a somewhat artificial environment, however, since we still require hand-annotated nodule information to load into our classifier. We don't have a good way of creating that input automatically. Plugging in overlapping 32x32x32 patches of data would result in $31 \times 31 \times 7 = 6727$ patches per CT, or about 10x the number of annotated samples we have. We'd need to overlap the edges; our classifier expects the nodule to be centered, and even then the inconsistent positioning will probably present issues.

That's not to say that it's impossible to handle these issues! This book is going to be using the multiple-step approach, however, due to there being a large number of well-performing open source projects that handle it similarly. In addition, we feel that the multi-step approach allows for a more smooth introduction of new concepts, making the learning process easier.

In any case, it's now time to fix the issue of still needing a human in the loop to provide those annotations. In this chapter we are going to take raw CT scans, find the nodules, and then get

ready to pass their location information off to the classifier we've built previously. To do that, we have to flag voxels that look like they might be part of a nodule, a process known as "segmentation."

- bigger picture - go back to overall MM
 - zoom in on part where we figure out where to get crops from

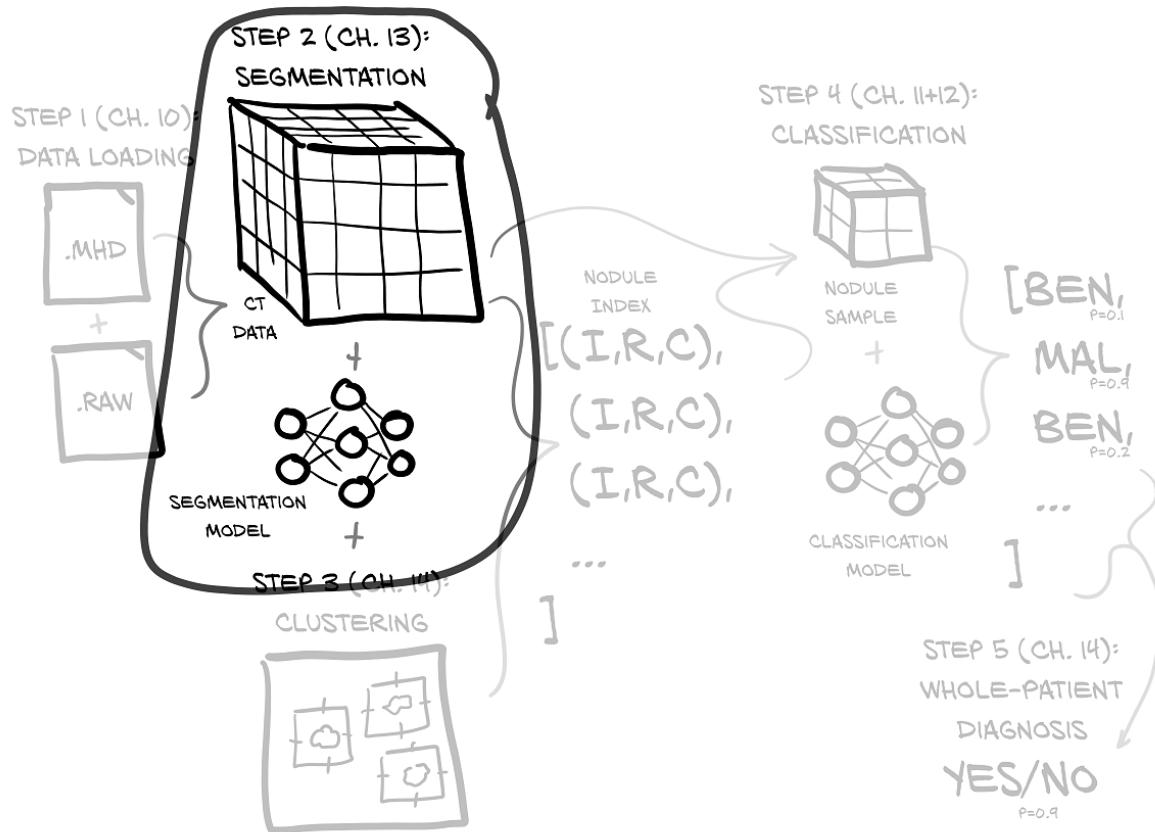


Figure 13.1 Our end-to-end lung cancer detection project, with a focus on this chapter's topic; step 2, segmentation.

By the time we're done with this chapter, we'll be have gotten our project to the finish line! We'll be able to take raw CT scans to suspected locations of malignancy, and from there into individual nodule classifications, and finally to whole-patient diagnosis predictions. To get started, we need to talk about segmentation.

NOTE

As we go through code examples in this chapter, we're going to be relying on you checking the code from github for much of the larger context. We'll be omitting code that's uninteresting or similar to what's come before in earlier chapters, so that we can focus on the crux of the issue at hand.

13.1 Segmentation is per-pixel classification

Segmentation is the act of dividing up an image into distinct chunks or regions. Often, this is used for object detection, in which the system answers questions of the form "Where is the cat?" Obviously, most pictures of a cat have a lot of non-cat in them; there's the table or the wall in the background, the keyboard they're sitting on, that kind of thing. Being able to say "this pixel is part of the cat, and this other pixel is part of the wall" requires a fundamentally different model output, and a different internal structure from the classification models we've worked with thus far. If your project requires differentiating between a near cat and a far cat, or a cat on the left vs. a cat on the right, then segmentation is probably the right approach.

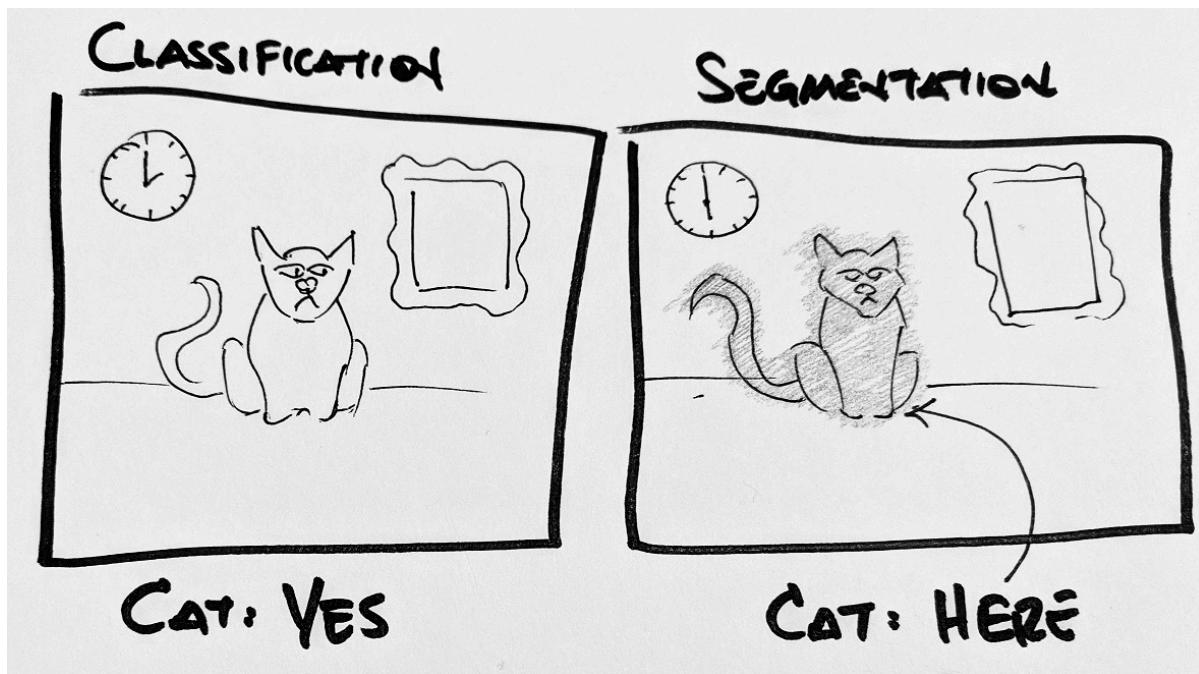


Figure 13.2 Classification results in one or more binary flags, while segmentation produces a mask or heatmap.

The image-consuming models that we've implemented so far can be thought of as a funnel or magnifying glass that takes a large bunch of pixels and focuses them down into a single point. "Yes, this huge pile of pixels has a cat in it, somewhere," or "No, no cats here." This is great when you don't care where the cat is, just that there is (or isn't) one in the image.

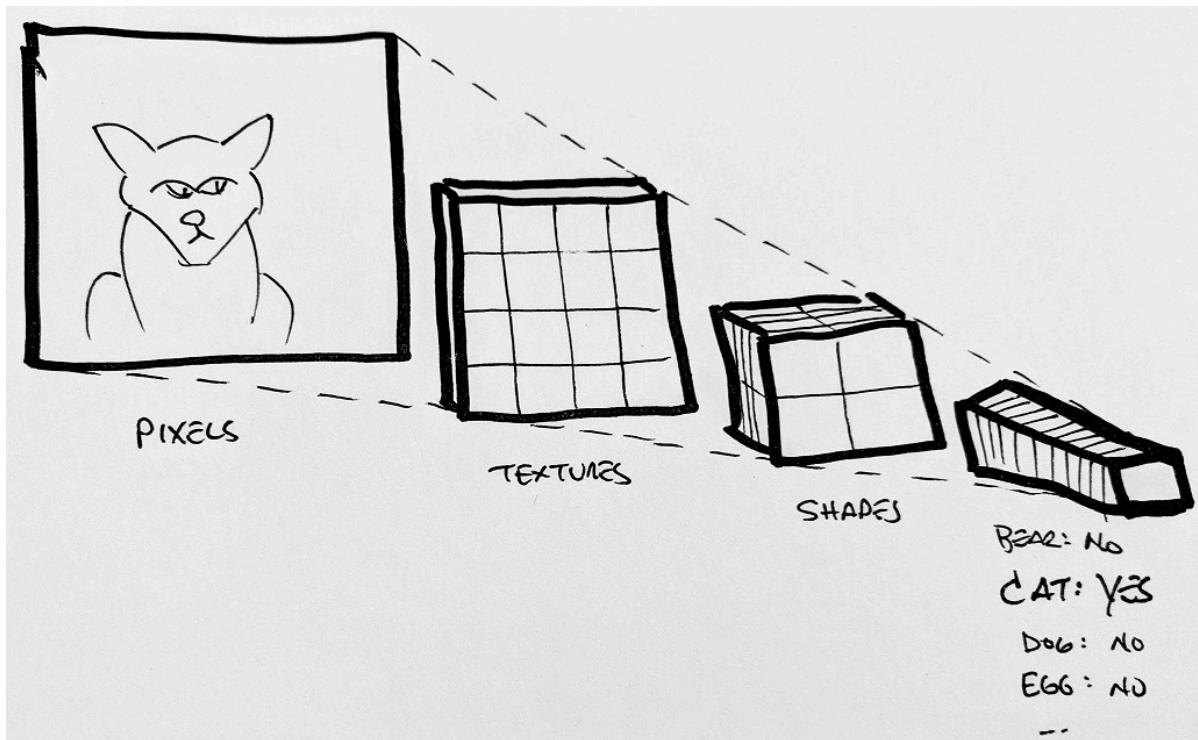


Figure 13.3 The magnifying glass model structure for classification.

Repeated layers of convolution and downsampling mean that the model starts with specific, detailed detectors for things like texture and color, and then builds up higher-level conceptual feature detectors that finally result in "cat" vs. "dog." Due to the increasing receptive field of the convolutions after each downsampling layer, those higher-level detectors can use information from an increasingly large area of the input image.

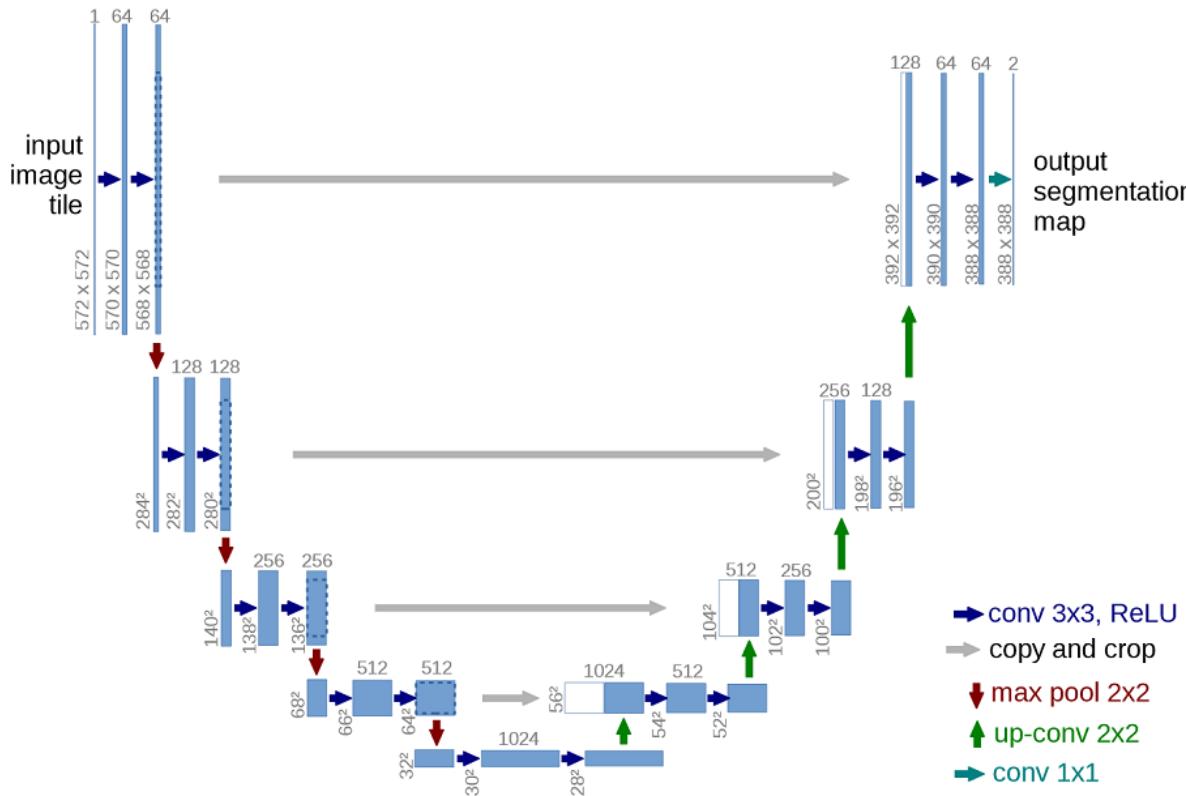
One simple model to use for segmentation would be to have repeated convolutional layers without any downsampling. Given appropriate padding, that would result in output the same size as the input (good), but a very limited receptive field (bad) due to the limited reach based on how much overlap multiple layers of small convolutions will have. The classification model uses each downsampling layer to double the effective reach of the following convolutions, and without that increase in effective field size, each segmented pixel will only be able to consider a very local neighborhood.

NOTE

Assuming 3x3 convolutions, the receptive field size for a simple model of stacked convolutions is $2 * L + 1$, with L being the number of convolutional layers.

13.1.1 The UNet architecture

The UNet architecture¹⁴⁴ was an early breakthrough for image segmentation. Prior work had attempted to address the limited receptive field size of fully convolutional networks by using a design that copied, inverted, and appended the focusing portions of an image classification network to create a symmetrical model that goes from fine detail to wide receptive field and back to fine detail. These network designs had problems converging, most likely due to the loss of spatial information during downsampling. To address this, the UNet authors added skip connections.



13.1.2 An off-the-shelf model: adding UNet to our project

Rather than implementing UNet from scratch, we're going to appropriate an existing implementation from an open-source repository on GitHub. The one at [jvanvugt/pytorch-unet](https://github.com/jvanvugt/pytorch-unet)¹⁴⁵ seems to meet our needs well. It's MIT licensed¹⁴⁶, contained in a single file, and has a number of parameter options for us to tweak. The file is included in our code repository at `util/unet.py`, along with a link to the original repository, and the full text of the license used.

We are going to make a few modifications to the classic UNet setup. First, we're going to pass all of the input through Batch Normalization. This will allow us to not have to normalize the data ourselves in the dataset, while still getting the benefits of normalized data. Second, since the output values are unconstrained, we are going to pass the output through an `nn.Sigmoid` layer to clamp the output to the range [0, 1].

This can be done rather simply by implementing a model with three attributes, one each for the two features we want to add, and one for the UNet itself. We will also pass along any keyword arguments into the UNet constructor.

Listing 13.1 p2ch13/model_seg.py, line 13: class UNetWrapper

```
class UNetWrapper(nn.Module):
    def __init__(self, **kwargs):
        super().__init__()

        self.input_batchnorm = nn.BatchNorm2d(kwargs['in_channels'])
        self.unet = UNet(**kwargs)
        self.final = nn.Sigmoid()

        self._init_weights()
```

The `forward` method is a similarly straightforward sequence. We could use an instance of `nn.Sequential` like we saw in chapter 8, 11.3.2, but we'll be explicit here for clarity.

Listing 13.2 p2ch13/model_seg.py, line 40: class UNetWrapper.forward

```
def forward(self, input_batch):
    bn_output = self.input_batchnorm(input_batch)
    un_output = self.unet(bn_output)
    fn_output = self.final(un_output)

    return fn_output
```

Note that we're using `nn.BatchNorm2d` here. This is because UNet is fundamentally a two-dimensional segmentation model. We could adapt the implementation to use 3D convolutions, etc. but the memory usage would be too high to fit into typical GPUs. Instead, we'll adapt our 3D data to be segmented slice-at-a-time.

13.2 A 3D Dataset in 2D

The original UNet implementation did not use padded convolutions, which means that while the output segmentation map was smaller than the input, every pixel of that output had a fully-populated receptive field. None of the input pixels that fed into the determination of that output pixel were padded, fabricated, or otherwise incomplete. This means that the output will tile perfectly, and so can be used with images of any size (except at the edges of the input image, where some context will be missing by definition).

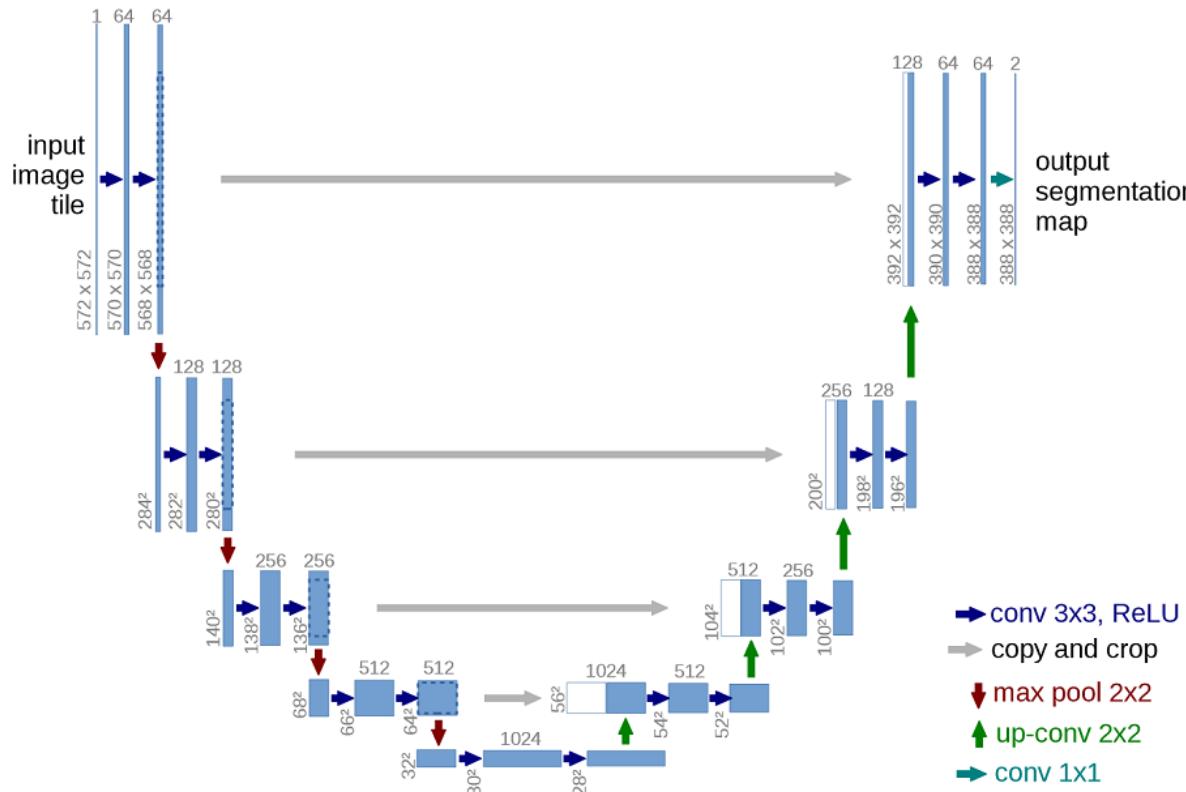


Figure 13.5 From the UNet paper.

There are two problems with us taking the same approach for our problem.

13.2.1 UNet has very specific input size requirements

The first is that the sizes of the input and output patches are very specific. In order to have the two-pixel loss per convolution line up evenly before and after downsampling (especially when considering the further convolutional shrinkage at that lower resolution), there are only certain input sizes that will work. The UNet paper used 572×572 image patches, which resulted in 388×388 output maps. The input images are bigger than our 512×512 CT slices, and the output is quite a bit smaller! That would mean that any nodules near the edge of the CT scan slice wouldn't get segmented at all. While this setup works well when dealing with very large images, it's not ideal for our use case.

We will address this issue by setting the `padding` flag of the UNet constructor to `True`. This will mean that we can use input images of any size, and will get output of the same size. We will lose some fidelity near the edges of the image, but that's a compromise we'll have to live with.

13.2.2 UNet in 3D would use too much RAM

The second issue is that our 3D data doesn't line up exactly with the 2D expected input of UNet. Simply taking our 512x512x128 image and feeding it into a converted-to-3D UNet class won't work, because we'll exhaust our GPU memory. Each image is 2^9 by 2^9 by 2^7 , with 2^2 bytes per voxel. The first layer of UNet is 64 channels, or 2^6 . That's an exponent of $9+9+7+2+6=33$, or 8GB *just for the first convolutional layer*. There are two convolutional layers, so 16GB, then each downsampling halves the resolution but doubles the channels, which is another 2GB each layer after the first downsample (remember, halving the resolution results in one eighth the data since we're working with 3D data), so we've hit 20GB before we even hit the second downsample, much less anything on the upsample side of the model or anything dealing with autograd.

Instead, we're going to treat each slice as a 2D segmentation problem, and cheat our way around the issue of context in the 3rd dimension by providing neighboring slices as separate channels. Instead of the traditional "Red," "Green," "Blue" channels that we're familiar with from photographic images, our main channels will be "two slices above," "one slice above," "the slice we're actually segmenting," "one slice below," and so on.

This isn't without tradeoffs, however. We lose the direct spatial relationship between slices when represented as channels, but this is a close enough approximation that it should work well enough. We also lose the wider receptive field in the depth dimension that would come from a true 3D segmentation. Since CT slices are often thicker than the resolution in rows and columns, we do get somewhat of a wider view than it seems we might. Of course, we ignore the exact thickness, so that's something our model will have to learn to be robust against.

In general, there isn't an easy flow chart or rule of thumb that can give canned answers to questions about which trade-offs to make, or if a given set of compromises compromise too much. Careful experimentation is key, however, and systematically testing hypothesis after hypothesis can help narrow down which changes and approaches are working well for the problem at hand. While it's tempting to make a flurry of changes while waiting for the last set of results to compute, *resist that impulse*.

That's important enough to repeat: *do not test multiple modifications at the same time*.

There is far too high a chance that one of the changes will interact poorly with the other, and you'll be left without solid evidence that either one is worth investigating more.

With that said, let's start building out our segmentation dataset.

13.2.3 Building the ground truth data

The first thing that we'll need to address is that we have a mismatch between our human-labeled training data, and the actual output we want to get from our model. We have annotated points, but we want a per-voxel mask that indicates if any given voxel is part of a nodule or not. We'll have to build that mask ourselves from the data we have, and then do some manual checking to make sure that the routine that builds the mask is performing well.

Validating these manually constructed heuristics at scale can be difficult. We aren't going to attempt to do anything comprehensive when it comes to making sure that each and every nodule is properly handled by our heuristics. If we had more resources, approaches like "pay someone to create and/or verify everything by hand" might be an option, but since this isn't a well-funded endeavor, we're going to rely on checking a handful of samples and using a very simple "does the output look reasonable?" approach.

To that end, we're going to design our approaches and our APIs to make it easy to investigate the intermediate steps that our algorithms are going through. While this might result in slightly clunky function calls returning huge tuples of intermediate values, being able to easily grab results and plot them in a notebook makes the clunk worth it.

Other design decisions are going to be driven by some aspects of how `DataLoader` interacts with our `Dataset` subclass, particularly surrounding amortizing work. We'll call those out explicitly when we get to them.

BOUNDING BOXES

We are going to start by converting the nodule locations that we have into bounding boxes that cover the entire nodule. If we assume that the nodule locations are roughly centered, we can trace outward from that point in all three dimensions until we hit low-density voxels, indicating that we've hit normal lung tissue.

Here's what that looks like in code.

Listing 13.3 p2ch13/dsets.py, line 116: class Ct.buildAnnotationMask

```
center_irc = xyz2irc(
    noduleInfo_tup.center_xyz,      ①
    self.origin_xyz,
    self.vxSize_xyz,
    self.direction_tup,
)
ci = int(center_irc.index)
cr = int(center_irc.row)
cc = int(center_irc.col)

index_radius = 2
try:
    while self.hu_a[ci + index_radius, cr, cc] > threshold_hu and \
        self.hu_a[ci - index_radius, cr, cc] > threshold_hu:
        index_radius += 1
except IndexError:
    index_radius -= 1
```

- ① The `noduleInfo_tup` here is the same as we've seen previously, e.g. as returned by the `getNoduleInfoList` function.

Note that we stop incrementing the "radius" *after* the density drops below threshold, so that our bounding box should contain a one-voxel border of low density tissue (at least on one side; since nodules can be adjacent to regions like the lung wall, we have to stop for air on either side). Since we check both `center_index + index_radius` and `center_index - index_radius` against that threshold, that one-voxel boundary will only exist on the edge closest to our nodule location. This is why we need those locations to be relatively centered. Since some nodules are adjacent to the boundary between the lung and denser tissue like muscle or bone, we can't trace each direction independently, as some edges would end up incredibly far away from the actual nodule.

We then repeat the same radius-expansion process with `row_radius` and `col_radius` (this code omitted for brevity). Once that's done, we can build a slice tuple that we can use to set our bounding box mask array to `True` (we'll see the definition of `boundingBox_ary` in just a moment; it's not surprising).

Listing 13.4 p2ch13/dsets.py, line 155: class Ct.buildAnnotationMask

```
slice_tup = (
    slice(ci - index_radius, ci + index_radius + 1),
    slice(cr - row_radius, cr + row_radius + 1),
    slice(cc - col_radius, cc + col_radius + 1),
)
boundingBox_a[slice_tup] = True
```

Okay, let's wrap all this up in a function, as well as doing some cleanup on the resulting mask. We'll elide the code we've already looked at, which is all inside the body of the function's loop.

Listing 13.5 p2ch13/dsets.py, line 112: class Ct.buildAnnotationMask

```
def buildAnnotationMask(self, noduleInfo_list, threshold_hu = -500):
    boundingBox_a = np.zeros_like(self.hu_a, dtype=np.bool)

    for noduleInfo_tup in noduleInfo_list:
        # ... line 160
        boundingBox_a[slice_tup] = True

    thresholded_a = boundingBox_a & (self.hu_a > threshold_hu)
    mask_a = morph.binary_dilation(thresholded_a, iterations=2) ①

    return mask_a, thresholded_a, boundingBox_a
```

① This is `scipy.ndimage.morphology.binary_dilation`.

The first bit of cleanup we do is to take the intersection between the bounding box mask and the tissue that's denser than our threshold of 0.5 g/cc. That's going to clip off the corners of our boxes (at least the ones not embedded in the lung wall), and make it conform to the contours of the nodule a bit better. After that, we dilate the mask by a couple of pixels. The intent is that if there are any less dense, wispy or spiked structures next to some of the nodules (called *spiculation*, these are an indicator of malignancy), they will be included in the mask.

Let's take a look at what these masks look like in practice. Additional images in full color can be found in the `p2ch13_explore_data.ipynb` notebook.

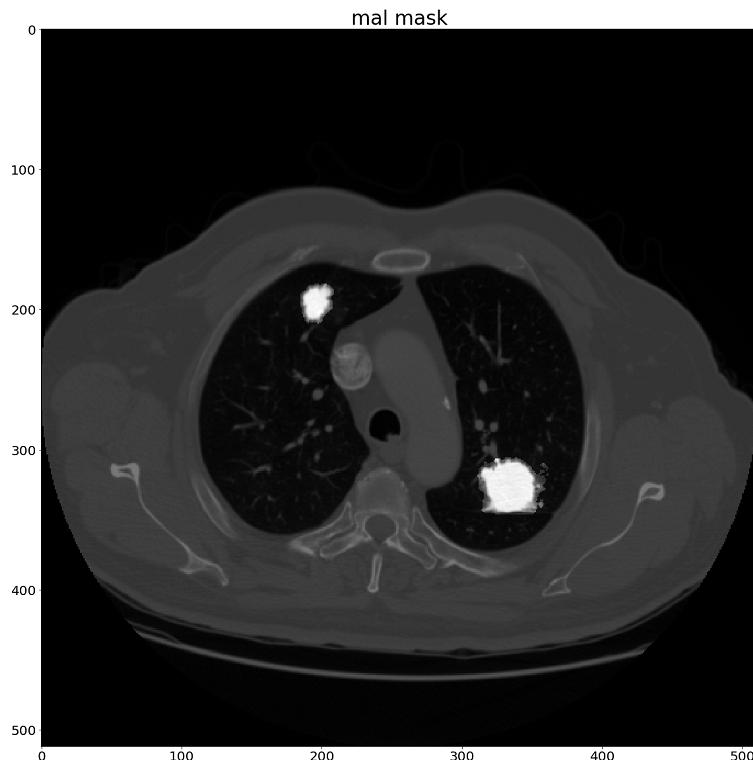


Figure 13.6 Two malignant modules from the mal_mask, highlighted in white.

Next we'll go about adding this and other masks to our CT class.

ADDING MASKS TO CT INIT

Now that we can take a list of nodule information tuples and turn them into at CT-shaped binary "is this a nodule" mask, let's go and embed those masks into our CT object. First, we'll filter our nodule info into a list containing only benign nodules, then use that to build the annotation mask. Finally, we'll collect the set of unique array indexes that have at least one voxel of the benign mask. We'll use this to shape class balancing during training.

Listing 13.6 p2ch13/dsets.py, line 78: class Ct.{uu}init{uu}

```
def __init__(self, series_uid, buildMasks_bool=True):
    # ... line 97
    noduleInfo_list = getNoduleInfoList()
    self.benignInfo_list = [ni_tup
        for ni_tup in noduleInfo_list
        if not ni_tup.isMalignant_bool
        and ni_tup.series_uid == self.series_uid]
    self.benign_mask = self.buildAnnotationMask(self.benignInfo_list)[0]
    self.benign_indexes = sorted(set(self.benign_mask.nonzero()[0]))
```

The malignant version follows the same design.

Listing 13.7 p2ch13/dsets.py, line 78: class Ct.{uu}init{uu}

```
def __init__(self, series_uid, buildMasks_bool=True):
    # ... line 97
    noduleInfo_list = getNoduleInfoList()
    # ... line 105
    self.malignantInfo_list = [ni_tup
        for ni_tup in noduleInfo_list
        if ni_tup.isMalignant_bool      ①
        and ni_tup.series_uid == self.series_uid]
    self.malignant_mask = self.buildAnnotationMask(self.malignantInfo_list)[0]
    self.malignant_indexes = sorted(set(self.malignant_mask.nonzero()[0]))
```

- ① The only algorithmic difference between this and the benign snippet above is the lack of the `not` here.

Next we're going to implement some basic image processing that will help us identify where the lungs are in our images. This will be useful to help constrain where our nodules stop and the rib cage wall starts.

IMPLEMENTING THE LUNG MASK

We are going to create a few binary masks that include the lungs, nodules, and a thin shell of tissue from the boundary between the lungs and ribs, diaphragm, or other bordering organs. Assuming we create this mask consistently, we can use it as an input to our model to help focus our segmentation on the anatomy we care about. We'll be applying a series of image processing routines from the `scipy.ndimage` package to produce a corresponding series of binary masks. While only the last few masks we create will be of actual use to the larger project, we will return all of the intermediate results to make it easier to debug what's going on if an image isn't handled well.

Here's what the first few masks look like, along with an unmasked CT slice:

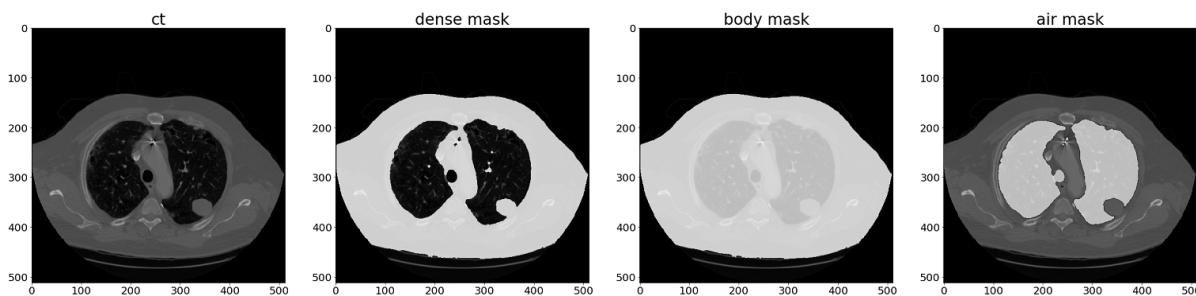


Figure 13.7 A slice of CT, followed by the mask of dense tissue, the body, and the air inside the lungs.

The code that generated this first set of masks is:

Listing 13.8 p2ch13/dsets.py, line 167: class Ct.build2dLungMask

```
def build2dLungMask(self, mask_ndx):
    raw_dense_mask = self.hu_a[mask_ndx] > -300
    dense_mask = morph.binary_closing(raw_dense_mask, iterations=2)
    dense_mask = morph.binary_opening(dense_mask, iterations=2)

    body_mask = morph.binary_fill_holes(dense_mask)
    air_mask = morph.binary_fill_holes(body_mask & ~dense_mask)
    air_mask = morph.binary_erosion(air_mask, iterations=1)
```

We start by taking all of the dense matter in the CT slice, and doing some cleaning on it by using the `binary_closing` and `binary_opening` functions from `scipy.ndimage.morphology`. The `binary_closing` function closes small holes in the mask by successive increasing of the mask by `iterations=2` pixels, then decreases the mask by the same amount. The net effect is that holes 4 pixels across or thinner get completely closed. The following `binary_opening` call does the opposite; small blobs less than 4 pixels across are completely eroded away, and the remaining mask is then increased back out to roughly where the mask edge was originally.

NOTE

Much of this complication is meant to handle the cases where the CT scan is relatively noisy. See if you can find some cases where the CT is particularly noisy!

The `body_mask` and `air_mask` are then straightforward hole filling and boolean operations on the `dense_mask`.

From there, we create the lung mask, which is used as input to the model, as well as the nodule, benign, and malignant masks. Including the lung mask can help training by focusing the model on the parts of the CT that are relevant to our problem. Doing this segmentation assistance will help avoid feeding spurious non-nodules into our classifier later on (assuming we create the lung mask properly in all cases — not always true!).

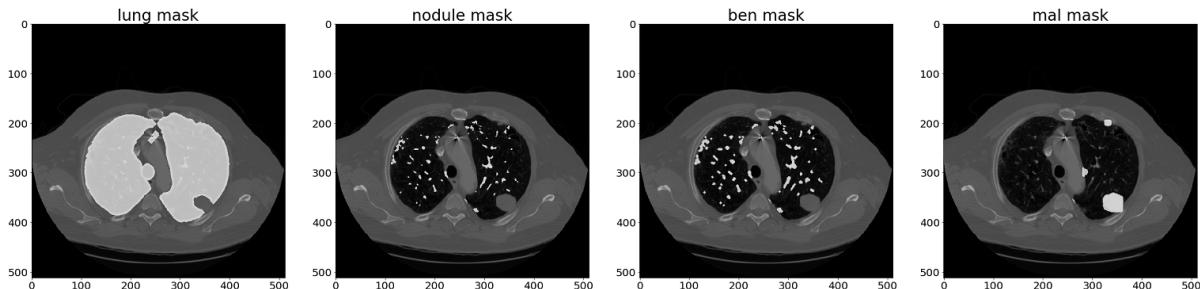


Figure 13.8 Possible nodules, the lung mask, and finally our benign and malignant masks used for training.

The `nodule_mask` is all of the dense parts inside the lungs that are large enough to pass through the `binary_opening` filter. Starting from there, the `ben_mask` makes each remaining nodule a bit larger, and then removes anything included in the `mal_mask`, which we calculated earlier during CT initialization.

Listing 13.9 p2ch13/dsets.py, line 176: class Ct.build2dLungMask

```
lung_mask = morph.binary_dilation(air_mask, iterations=5)

raw_nodule_mask = self.hu_a[mask_ndx] > -600
raw_nodule_mask &= air_mask
nodule_mask = morph.binary_opening(raw_nodule_mask, iterations=1)

ben_mask = morph.binary_dilation(nodule_mask, iterations=1)
ben_mask &= ~self.malignant_mask[mask_ndx]

mal_mask = self.malignant_mask[mask_ndx]
```

After that, we return a named tuple with all of the relevant mask arrays.

Note that here we are implementing a two-dimensional algorithm. While it would make just as much sense to run this function over the entire three-dimensional CT array during initialization of the `ct` object, doing so would be rather slow. By designing this function to be called for only a

single slice at a time, we are able to invoke it once per call to `Dataset.{uu}getitem{uu}`, which amortizes the expense of the computation much more evenly over each training sample. This allows our `DataLoader` instance to not get bottle-necked when every hundredth (or so) training sample (the first for that CT) takes several orders of magnitude more time than the others. It also allows us to randomize the order of CTs we pull samples from, since we won't be paying to compute the masks for the entire array for each sample we want to load.

13.2.4 Implementing the `Luna2dSegmentationDataset`

We are going to take a different approach to the training and validation split in this chapter. We are going to have two classes, one acting as a general base class suitable for validation data, and one for the training set, with randomization and augmentation. While this is somewhat more complicated in some ways, this approach actually simplifies the logic of selecting randomized training samples and the like. It becomes extremely clear which code paths can impact both training and validation, and which are isolated to training only. Without this, we found that some of the logic can become nested or intertwined in ways that make it hard to follow.

The data that we produce is going to be two-dimensional CT slices, with multiple channels. Several of the channels are going to be adjacent slices of CT, and one of the channels will be a binary mask that marks the lung. We'll need to produce one sample per slice of CT, for each CT we have. Since CTs are of different sizes, we're going to introduce a new function that caches the size of each CT scan to disk. We need this to be able to quickly construct the full size of a validation set without having to load each CT at `Dataset` initialization. We'll continue to use the same caching decorator as before. This means that we'll need to run the `precache.py` script again before we start any training runs.

Listing 13.10 p2ch13/dsets.py, line 239

```
@raw_cache.memoize(typed=True)
def getCtSampleSize(series_uid):
    ct = Ct(series_uid, buildMasks_bool=False)
    return len(ct.benign_indexes)
```

The majority of the `Luna2dSegmentationDataset.{uu}init{uu}` method is similar to what we've seen before. We have a new `contextSlices_count` parameter, as well as an `augmentation_dict` similar to what we introduced last chapter. The handling for the flag indicating if this is meant to be a training or validation set is going to have to change somewhat. Since we're no longer training on individual nodules, we are going to have to partition the list of series into training and validation sets.

Listing 13.11 p2ch13/dsets.py, line 453: class Luna2dSegmentationDataset.{uu}init{uu}

```
if isValSet_bool:
    assert val_stride > 0, val_stride
    self.series_list = self.series_list[::-val_stride]
    assert self.series_list
elif val_stride > 0:
    del self.series_list[::-val_stride]
    assert self.series_list

self.sample_list = []
for series_uid in self.series_list:
    if fullCt_bool:
        self.sample_list.extend([(series_uid, ct_ndx) for ct_ndx in
            range(getCt(series_uid).hu_a.shape[0])])
    else:
        self.sample_list.extend([(series_uid, ct_ndx) for ct_ndx in
            range(getCtSampleSize(series_uid))])
```

We also add similar code to the classification Dataset, since we want to make sure that we don't cross-pollute our validation data. Training any of the models used on the validation data would invalidate our results.

Our `{uu}getitem{uu}` implementation is also going to get a little bit more fancy. We're going to allow calling it with a `tuple` in addition to the standard `int`. The `int` argument will get the usual Nth sample from our sample list, while the `tuple` will allow us to specify exactly what CT slice index to fetch, as well as turn on or off augmentation for the duration of the call.

Listing 13.12 p2ch13/dsets.py, line 478: class Luna2dSegmentationDataset.{uu}getitem{uu}

```
def __getitem__(self, ndx):
    if isinstance(ndx, int):
        series_uid, sample_ndx = self.sample_list[ndx % len(self.sample_list)]
        ct = getCt(series_uid)    ①
        ct_ndx = self.sample_list[sample_ndx][1]
        useAugmentation_bool = False
    else:
        series_uid, ct_ndx, useAugmentation_bool = ndx
        ct = getCt(series_uid)    ①
```

- ① Calling `getCt` with same argument multiple times is "free" due to in-memory caching.

Our training subclass will take advantage of this feature, as will our visualization and debugging code.

Once we have our `ct` and `ct_ndx` specified, we can go and get our context slices pulled out of the CT (bounds checking omitted).

Listing 13.13 p2ch13/dsets.py, line 488: class Luna2dSegmentationDataset.{uu}getitem{uu}

```
ct_t = torch.zeros((self.contextSlices_count * 2 + 1 + 1, 512, 512))

start_ndx = ct_ndx - self.contextSlices_count
end_ndx = ct_ndx + self.contextSlices_count + 1
for i, context_ndx in enumerate(range(start_ndx, end_ndx)):
    ct_t[i] = torch.from_numpy(ct.hu_a[context_ndx].astype(np.float32))
ct_t /= 1000
```

Next we'll call our `build2dLungMask` method, and use the resulting `mask_tup` to construct our `nodule_t` we'll be using as our label data for training. Note that we're dividing the `ct_t` by 1000 above in anticipation of concatenating the HU data from the CT scan with boolean values from the lung mask. As we discussed in chapter 10, 10.2.1, if we left these values raw, the -1000 to +1000 range on the HU would totally dominate the 0 to 1 range of the boolean values. The model would have an incredibly difficult time telling the true and false values apart, especially after passing them through a batch normalization layer!

Listing 13.14 p2ch13/dsets.py, line 499: class Luna2dSegmentationDataset.{uu}getitem{uu}

```
mask_tup = ct.build2dLungMask(ct_ndx)

ct_t[-1] = torch.from_numpy(mask_tup.lung_mask.astype(np.float32)) ❶

nodule_t = torch.from_numpy(
    (mask_tup.mal_mask | mask_tup.ben_mask).astype(np.float32)
).unsqueeze(0)
# ... line 528
return ct_t, nodule_t, label_int, ben_t, mal_t, ct.series_uid, ct_ndx
```

- ❶ Here we fill in the final slice with the `lung_mask`.

The remainder of the tuple we return, `...label_int, ben_t, mal_t, ct.series_uid, ct_ndx`, are all information that we include for debugging and display. We don't need any of it for training.

13.3 Updating the training script

Overall, the training script `p2ch13/train_seg.py` is very similar to what we used for classification training in chapter 12. We've also copied the classification training script forward into `p2ch13/train_cls.py`. Any significant changes will be covered here in the text, but be aware that some of the minor tweaks might get skipped. For the full story, check the source.

Our `initModel` method will be the same, except that we're now using the `UNetWrapper` class.

Listing 13.15 p2ch13/train_seg.py, line 129: class LunaTrainingApp.initModel

```
def initModel(self):
    model = UNetWrapper(
        in_channels=8,
        n_classes=1,
        depth=4,
        wf=3,
        padding=True,
        batch_norm=True,
        up_mode='upconv',
    )

    if self.use_cuda:
        # ... line 144
    return model
```

We've got 8 input channels, $3 + 3$ context slices, 1 slice that is what we're segmenting, and 1 slice for the body mask. We have one output class, indicating if this voxel is part of a nodule or not. The `depth` parameter controls how deep the U goes; each downsampling operation adds one to the depth. Using `wf=3` means that the first layer will have $2^{**wf} == 8$ layers, which doubles with each downsampling. We want the convolutions to be padded, so that we get an output image the same size as our input. We also want batch normalization inside the network after each activation function, and our upsampling function do be upconvolution, as implemented by `nn.ConvTranspose2d`.¹⁴⁷

13.3.1 Getting images into tensorboard

One of the nice things about working on segmentation tasks is that the output is easily represented visually. Being able to eyeball our results can be a huge help determining if a model is progressing well (but perhaps needs more training), or if it has gone off the rails and we need to stop training. There are many ways we could package up our results as images; Tensorboard has great support for this kind of data, and it's already integrated with our training runs.

Let's see what it takes to get everything hooked up.

We're going to add a `logImages` function to our main application class, and call it with both our training and validation data loaders. After `doTraining`, there's no particular ordering required between any of the `logMetrics`, `logImages`, and `doValidation` calls. The model won't be changing, so the only consideration is what order we want the data to be made available in. Since we're going to make our image logging faster than `doValidation`, we can produce the images first and satisfy our curiosity early.

Listing 13.16 p2ch13/train_seg.py, line 191: class LunaTrainingApp.main

```
def main(self):
    # ... line 199
    best_score = 0.0
    for epoch_ndx in range(1, self.cli_args.epochs + 1):
        # ... line 210
        trnMetrics_t = self.doTraining(epoch_ndx, train_dl)
        self.logMetrics(epoch_ndx, 'trn', trnMetrics_t)
        self.logImages(epoch_ndx, 'trn', train_dl)
        self.logImages(epoch_ndx, 'val', val_dl)
```

There isn't a single right way to structure our image logging. We are going to grab a handful of CTs from both the training and validation sets. For each CT, we are going to select six evenly spaced slices, end-to-end, and show both the ground truth, as well as our model's output. We chose six slices only because Tensorboard will show 12 images at a time, and we can arrange the browser window to have a row of label images over the model output. Arranging things this way makes it easy to visually compare the two.

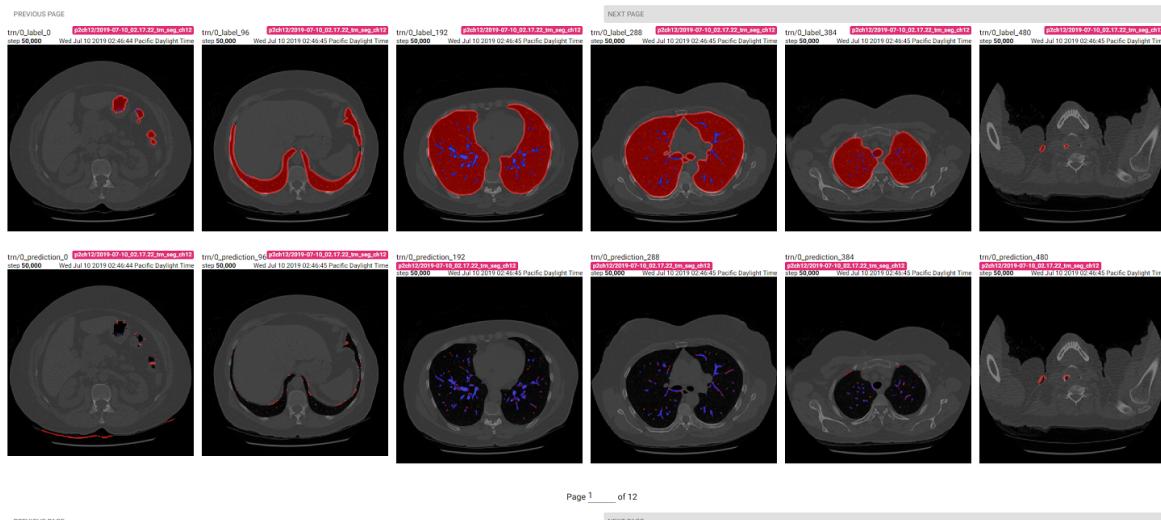


Figure 13.9 Top row: label data for training (the lung mask is used as input); bottom row: output from the segmentation model.

The setup for this is:

Listing 13.17 p2ch13/train_seg.py, line 334: class LunaTrainingApp.logImages

```
def logImages(self, epoch_ndx, mode_str, dl):
    images_iter = sorted(dl.dataset.series_list)[:12]
    for series_ndx, series_uid in enumerate(images_iter):
        ct = getCT(series_uid)

        for slice_ndx in range(6):
            ct_ndx = slice_ndx * ct.hu_a.shape[0] // 5
            ct_ndx = min(ct_ndx, ct.hu_a.shape[0] - 1)
            sample_tup = dl.dataset[(series_uid, ct_ndx, False)]

            ct_t, nodule_t, _, ben_t, mal_t, _, _ = sample_tup
```

After that, we take our `ct_tensor` and feed it into the model. This looks very much like what we see in `computeBatchLoss`; see `p2ch13/train_seg.py` for details if desired.

Once we have our `prediction_ary`, we need to build an `image_ary` that will hold RGB values to display. We're using `np.float32` values, which expect a range from 0 to 1. Our approach is going to cheat a little bit by adding together various images and masks to get data in the range 0 to 2, and then multiplying the entire array by 0.5 to get it back into the right range.

```
[source,python,indent=0]
.p2ch13/train_seg.py, line 358: class LunaTrainingApp.logImages

ctSlice_a = ct_t[dl.dataset.contextSlices_count].numpy()

image_a = np.zeros((512, 512, 3), dtype=np.float32)
image_a[:, :, :] = ctSlice_a.reshape((512, 512, 1)) ❶
image_a[:, :, 0] += prediction_a[0] * (1 - label_a[0]) ❷
image_a[:, :, 1] += prediction_a[0] * mal_a[0] ❸
image_a[:, :, 2] += prediction_a[0] * ben_a[0] ❹
image_a *= 0.5
image_a[image_a < 0] = 0
image_a[image_a > 1] = 1
```

- ❶ CT intensity to all RGB channels
- ❷ Red
- ❸ Green
- ❹ Blue
- ❺ Noise might cause sparkles

Our goal is to have a grayscale CT at half intensity, overlaid with predicted-nodule pixels in various colors. We're going to use red for pixels that are incorrect. `1 - label_ary` inverts the label, and that multiplied by the `prediction_ary` gives us only the predicted pixels that aren't in a nodule. Green gets set to every correctly predicted pixel inside of a malignant nodule. Similarly, blue is used for correctly predicted pixels in benign nodules.

After that, we renormalize our data, and clamp it (in case we start displaying augmented data here, which would cause speckles when the noise was outside our expected CT range). All that remains is to save the data to Tensorboard.

Listing 13.18 p2ch13/train_seg.py, line 369: class LunaTrainingApp.logImages

```
writer = getattr(self, mode_str + '_writer')
writer.add_image(
    '{} / {}_prediction_{}'.format(
        mode_str,
        series_ndx,
        slice_ndx,
    ),
    image_a,
    self.totalTrainingSamples_count,
    dataformats='HWC',
)
```

This looks very similar to the `writer.add_scalar` calls we've seen before. The `dataformats='HWC'` argument tells Tensorboard the order of axes in our image have our RGB channels as the third axis. As we recall, our network layers often specify outputs that are `BxCxHxW`, and we could put that data directly into tensorboard as well.

We also want to save the ground truth that we're using to train with as well:

Listing 13.19 p2ch13/train_seg.py, line 383: class LunaTrainingApp.logImages

```
if epoch_ndx == 1:
    image_a = np.zeros((512, 512, 3), dtype=np.float32)
    image_a[:, :, :] = ctSlice_a.reshape((512, 512, 1))
    image_a[:, :, 0] += (1 - label_a[0]) * ct_t[-1].numpy() # Red
    image_a[:, :, 1] += mal_a[0] # Green
    image_a[:, :, 2] += ben_a[0] # Blue
    # ... line 393
writer.add_image(
    '{} / {}_label_{}'.format(
        mode_str,
        series_ndx,
        slice_ndx,
    ),
    image_a,
    self.totalTrainingSamples_count,
    dataformats='HWC',
)
```

Here, we are using the `ct_tensor[-1]` for the red channel, which we set to the `lung_mask` in our `Dataset`. We multiply that by the inverse of the lung mask again, so that we don't mix our red channel with our green malignant nodules and blue benign nodules. After that, we normalize and save this new label image as we did with the prediction image.

13.3.2 Dice loss

The Sørensen–Dice coefficient¹⁴⁸, also known as the Dice loss, is a common loss metric for segmentation tasks. One advantage using Dice loss has over mean squared error is that Dice handles the case where only a small portion of the overall image is flagged as positive. As we recall from chapter 11 in [Why is this happening?](#), unbalanced training data can be problematic. That's exactly the situation we have here — most of a CT scan isn't a nodule. Luckily, with Dice, that won't pose a problem.

The Sørensen–Dice coefficient is based on the ratio of correctly segmented pixels to the sum of the predicted and actual pixels. Those ratios are laid out in 13.10. That might sound familiar; it's the same ratio that we saw in chapter 12, 12.2.4. We're basically going to be using a per-pixel F1 score!

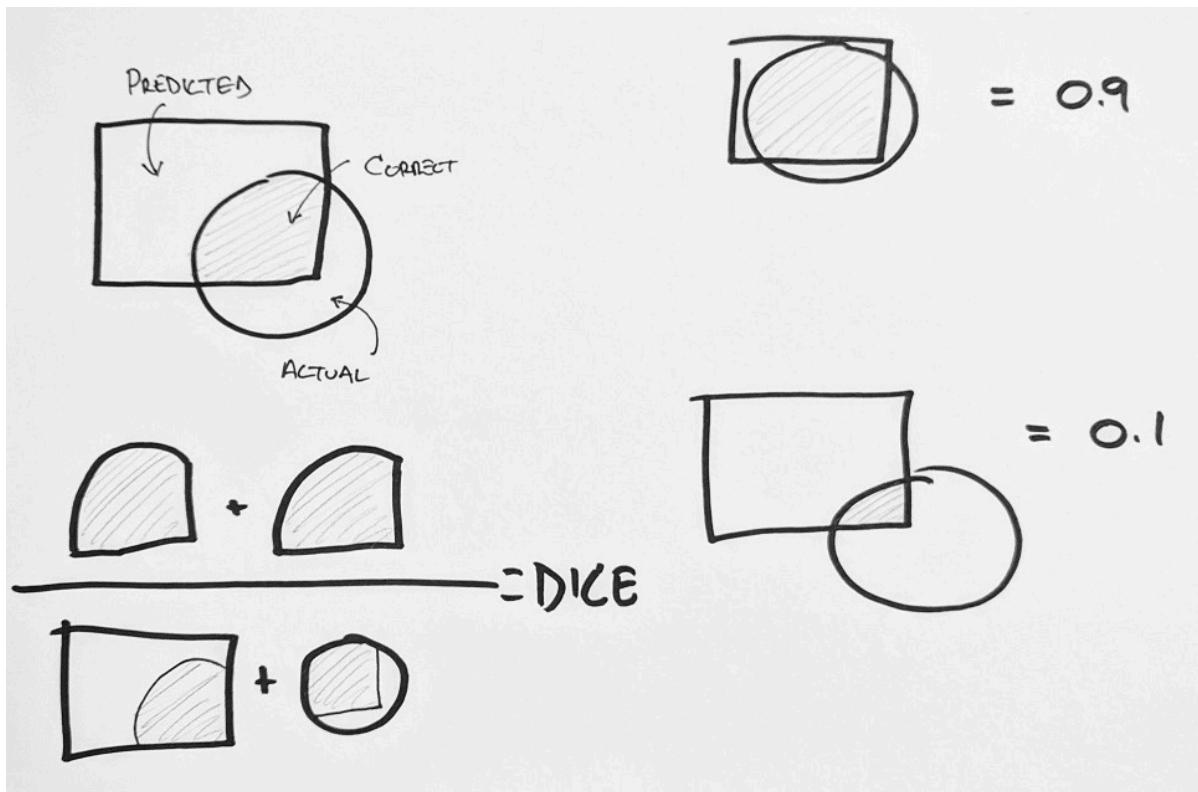


Figure 13.10 DICE ratios

There's one tiny complication. Since we're wanting a loss to minimize, we're going to take our ratio and subtract it from one. Doing so will invert the slope of our loss function, so that in the high overlap case our loss is low, and in the low overlap case, it's high. Here's what that looks like in code:

Listing 13.20 p2ch13/train_seg.py, line 308: class LunaTrainingApp.diceLoss

```
def diceLoss(self, label_g, prediction_g, epsilon=1, p=False):
    sum_dim1 = lambda t: t.view(t.size(0), -1).sum(dim=1)

    diceLabel_g = sum_dim1(label_g)
    dicePrediction_g = sum_dim1(prediction_g)
    diceCorrect_g = sum_dim1(prediction_g * label_g)

    epsilon_g = torch.ones_like(diceCorrect_g) * epsilon
    diceLoss_g = 1 - (2 * diceCorrect_g + epsilon_g) \
        / (dicePrediction_g + diceLabel_g + epsilon_g)

    return diceLoss_g
```

Note that we aren't treating the `prediction_devtensor` as a boolean here. Wishy-washy pixel predictions in the 0.4 to 0.6 range will contribute roughly the same amount to our gradient updates, no matter which side of 0.5 they might happen to fall on.

We're going to update our `computeBatchLoss` function to call `self.diceLoss`:

Listing 13.21 p2ch13/train_seg.py, line 263: class LunaTrainingApp.computeBatchLoss

```
def computeBatchLoss(self, batch_ndx, batch_tup, batch_size, metrics_g):
    input_t, label_t, label_list, ben_t, mal_t, _, _ = batch_tup
    # ... line 275
    prediction_g = self.model(input_g)
    diceLoss_g = self.diceLoss(label_g, prediction_g)

    with torch.no_grad():
        malLoss_g = self.diceLoss(mal_g, prediction_g * mal_g, p=True)
        predictionBool_g = (prediction_g > 0.5).to(torch.float32)

        metrics_g[METRICS_LABEL_NDX, start_ndx:end_ndx] = label_list
        metrics_g[METRICS_LOSS_NDX, start_ndx:end_ndx] = diceLoss_g
        metrics_g[METRICS_MAL_LOSS_NDX, start_ndx:end_ndx] = malLoss_g
    # ... line 305
    return diceLoss_g.mean()
```

We are also going to be doing something a little odd with our malignant predictions. It's vital that our segmentation model correctly identify malignant nodules as nodules so that the classifier has the opportunity to identify them. We're going to recalculate the Dice loss using the malignant mask as our ground truth. The model is being trained to identify considerably more pixels than just the malignant nodules, however, so we cannot compare the malignant mask to the predictions directly. Instead, we will multiply the predictions by the malignant mask, which results in all false positives being eliminated. False negatives can still increase our loss, which is exactly what we're looking to track.

Listing 13.22 p2ch13/train_seg.py, line 286: class LunaTrainingApp.computeBatchLoss

```
malPred_g = predictionBool_g * mal_g
tp = intersectionSum( mal_g, malPred_g)
fn = intersectionSum( mal_g, 1 - malPred_g)

metrics_g[METRICS_MTP_NDX, start_ndx:end_ndx] = tp
metrics_g[METRICS_MFN_NDX, start_ndx:end_ndx] = fn

del malPred_g, tp, fn
```

We also compute similar values for the overall segmentation results. These true positive, etc. metrics were previously computed in `logMetrics`, but due to the size of the result data (recall that each single CT slice is a quarter-million pixels!), we need to compute these summary stats live.

Listing 13.23 p2ch13/train_seg.py, line 295: class LunaTrainingApp.computeBatchLoss

```
tp = intersectionSum( label_g, predictionBool_g)
fn = intersectionSum( label_g, 1 - predictionBool_g)
fp = intersectionSum(1 - label_g, predictionBool_g)

metrics_g[METRICS_ATP_NDX, start_ndx:end_ndx] = tp
metrics_g[METRICS_AFN_NDX, start_ndx:end_ndx] = fn
metrics_g[METRICS_AFP_NDX, start_ndx:end_ndx] = fp

del tp, fn, fp
```

Now that we have these fancy new statistics, let's display them.

13.3.3 Updating our metrics logging

new metrics include returning a "score" based on f1, with recall as tiebreaker

Listing 13.24 p2ch13/train_seg.py, line 445: class LunaTrainingApp.logMetrics

```
metrics_dict['percent_mal/tp'] = sum_a[METRICS_MTP_NDX] / (malLabel_count or 1) * 100
metrics_dict['percent_mal/fn'] = sum_a[METRICS_MFN_NDX] / (malLabel_count or 1) * 100
```

Similar entries will be computed for `percent_all`, and the logging lines will include these new values. Nothing here will be particularly surprising.

We are going to start scoring our models as a way to determine if a particular training run is the best that we've seen so far. The thinking is that our malignant loss needs to be low (close to zero, hopefully), while our F1 score should be close to one. Our recall and overall loss are used as tiebreakers.

Listing 13.25 p2ch13/train_seg.py, line 405: class LunaTrainingApp.logMetrics

```
def logMetrics(self,
    epoch_ndx,
    mode_str,
    metrics_t,
):
    # ... line 453
    metrics_dict['pr/f1_score'] = 2 * (precision * recall) \
        / ((precision + recall) or 1)
    # ... line 498
    score = 1 \
        - metrics_dict['loss/mal'] \
        + metrics_dict['pr/f1_score'] \
        - metrics_dict['pr/recall'] * 0.01 \
        - metrics_dict['loss/all'] * 0.0001

    return score
```

Back in the main training loop, we'll keep track of the `best_score` we've seen so far this training run. When we save our model, we'll include a flag that indicates if this is the best score we've seen so far or not.

Listing 13.26 p2ch13/train_seg.py, line 191: class LunaTrainingApp.main

```
def main(self):
    # ... line 199
    best_score = 0.0
    for epoch_ndx in range(1, self.cli_args.epochs + 1):
        # ... line 216
        valMetrics_t = self.doValidation(epoch_ndx, val_dl)
        score = self.logMetrics(epoch_ndx, 'val', valMetrics_t)
        best_score = max(score, best_score)

        self.saveModel('seg', epoch_ndx, score == best_score)
```

Let's take a look at how we persist our model to disk.

13.3.4 Saving our model

PyTorch makes it pretty easy to save our model to disk. Under the hood, `torch.save` uses the standard Python `pickle` library, which means that we could pass our model instance in directly and it would save properly. That's not considered the ideal way to persist our model, however, since we lose some flexibility.

Instead, we will save only the *parameters* of our model. Doing this allows us to load those parameters into any model that expects parameters of the same shape, even if the class doesn't match the model those parameters were saved under. The save-parameters-only approach allows us to reuse and remix our models in more ways than saving the entire model.

We can get at our model's parameters using the `model.state_dict()` function.

Listing 13.27 p2ch13/train_seg.py, line 529: class LunaTrainingApp.saveModel

```
def saveModel(self, type_str, epoch_ndx, isBest=False):
    # ... line 545
    model = self.model
    if hasattr(model, 'module'):
        model = model.module ①

    state = {
        'model_state': model.state_dict(), ②
        'model_name': type(model).__name__,
        'optimizer_state' : self.optimizer.state_dict(), ③
        'optimizer_name': type(self.optimizer).__name__,
        'epoch': epoch_ndx,
        'totalTrainingSamples_count': self.totalTrainingSamples_count,
    }
    torch.save(state, file_path)
```

- ① This gets rid of `DataParallel`, if it exists.
- ② This is the important part.
- ③ Preserves momentum, etc.

We set our `file_path` to something like `data-unversioned/part2/models/p2ch13/seg_2019-07-10_02.17.22_ch12.50000.state`. The `.50000.` part is the number of training samples we've presented to the model so far, while the other parts of the path are obvious.

By saving the optimizer state as well, we could resume training seamlessly. While we don't provide an implementation of this, it could be useful if your access to computing resources is likely to be interrupted. Details on loading a model and optimizer to restart training can be found in the official documentation¹⁴⁹.

If the current model has the best score we've seen so far, we save a second copy of the `state` with a `.best.state` filename. This might get overwritten later by another, higher-score version of the model. By focusing only on this best file, we can divorce customers of our trained model from the details of how each epoch of training went (assuming, of course, that our score metric is high-quality).

Listing 13.28 p2ch13/train_seg.py, line 561: class LunaTrainingApp.saveModel

```
if isBest:  
    file_path = os.path.join(  
        'data-unversioned',  
        'part2',  
        'models',  
        self.cli_args.tb_prefix,  
        '{}_{}_{}.{}.state'.format(  
            type_str,  
            self.time_str,  
            self.cli_args.comment,  
            'best',  
        )  
    )  
    torch.save(state, file_path)
```

We have also updated `train_cls.py` with a similar routine for saving the classification model. In order to diagnose a CT, we'll need to have both models.

13.4 Conclusion

13.5 Exercises

13.6 Summary

- Segmentation is flagging individual pixels or voxels for membership in a class. This is in contrast to classification, which operates at the level of the entire image.
- We used segmentation to flag suspicious voxels of lung tissue, and fed the results into our classifier. This allowed us to separate out concerns between finding anatomical structures of interest, and determining malignancy.
- Providing extra masks or channels of information can help the segmentation model stay focused on the areas of interest. Providing a body mask makes it more obvious that air outside the body isn't actually lung.
- Naive approaches to 3D segmentation can quickly use too much RAM for current generation GPUs. Carefully limiting the scope of what is presented to the model can help limit RAM usage.
- Model parameters can be saved to disk and loaded back to reconstitute a model that had been saved earlier. The exact model implementation can change, as long as there is a 1:1 mapping between the old parameters and the new.

Deploying PyTorch Models



15

Deploying to production

This chapter covers:

- Various options to deploy PyTorch models.
- Deploying a server for our models.
- Exporting our models.
- Making good use of the PyTorch JIT with all of this.
- Running exported models from C++.
- Natively implementing models in C++.
- Running our models on mobile.

In Part I we learned a lot about models and Part II left us with a detailed path towards good models for a particular problem. Now that we have these great models, we need to take them where they can be useful. Maintaining infrastructure for executing inference of deep learning models at scale can be impactful from an architectural as well as cost standpoint. While PyTorch started off as a framework focused on research, starting with the 1.0 release a set of production-oriented features were added that today make PyTorch an ideal end-to-end platform from research to large-scale production.

What deploying to production means to use will vary with the use case:

- Perhaps the most natural deployment for the models we developed in Part II is that we might set up a network service providing access to our models. We do this in two versions using lightweight Python web frameworks, Flask [1: <http://flask.pocoo.org>] and Sanic [2: <https://sanicframework.org>]. The first is arguably one of the most popular of these frameworks and the latter is similar in spirit but leverages Python's new `async/await`

support for asynchronous operations for efficiency.

- We might export our model to a well-standardized format that allows us to ship it using optimized model processors, specialized hardware, or cloud services. For PyTorch models, the ONNX format fills this role.
- We may wish to integrate our models into larger applications. For this it would be handy if we were not limited to Python. Thus we will explore using PyTorch models from C++ with the idea that this also is a stepping stone to any language.
- Finally, for some things like the Zebraification, it may be nice to run our model on Mobile devices. While it is unlikely that you will have a CT module for your mobile, other medical applications like do-it-yourself skin screenings can be more natural and the user might prefer running on the device over having their skin sent to a cloud service. Luckily for us, PyTorch has gained mobile support recently and we explore that.

As we learn how to implement these use cases, we will use the classifier from [Chapter 13] as our first example to work for serving and then switch to the Zebraification model for the other bits of deployment.

15.1 Serving PyTorch models

So let us now turn to what it takes to put our model on a server. Staying true to our hands-on approach, we start out with the simplest possible server. Once we have something basic that works we take look at its shortfalls and take a stab at resolving them. In the end we'll look a bit into what is, at the time of the writing, the future.

So let's get something that listens on the network. [3: To play safe, do not do this on an untrusted network.]

15.1.1 Our model behind a Flask server

Flask is one of the most widely used Python modules, it can be installed using pip [4: Or pip3 for Python3, you might also want to run it as root or with sudo.]

```
pip install Flask
```

The API can be created by decorating functions, such as:

Listing 15. 1. flask_hello_world.py

```
from flask import Flask
app = Flask(__name__)

@app.route("/hello")
def hello():
    return "Hello World!"

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8000)
```

When started, the application will run at port 8000 and expose one route, `/hello`, that returns the "Hello World" string. At this point, we can augment our Flask server by loading a previously saved model and exposing it through a `POST` route. We will use the nodule classifier from [Chapter 13] as an example.

Instead of `/hello` we will now expose a `/predict` route that takes binary blob (the pixel content of the series) and the related meta data (a JSON object containing a dictionary with `shape` as a key) as input files provided with a POST request, and returns a JSON response with the predicted diagnosis. More precisely, our server takes one sample (rather than a batch) and returns the probability that it is malignant.

We use Flask's (somewhat curiously imported) `request` to get our data. More precisely, `request.files` contains a dictionary of file objects associated indexed by field names. We use JSON parsing of the input and return a JSON string using flask's `jsonify` helper.

The actual handling of the model is just like in [Chapter 13]: We instantiate the `LunaModel` from [Chapter 13], load the weights we got from our training, and put it in `eval` mode. As we are not training anything, we tell PyTorch that we will not want gradients when running the model by running in a `with torch.no_grad()` block.

In order to get to the data, we first need to decode the JSON to binary which we can then decode into a one-dimensional array with `numpy.frombuffer`. We convert this to a `Tensor` with `torch.from_numpy` and view it to its actual shape.

Listing 15. 2. flask_server.py

```
import numpy as np
import sys
import os
import torch
from flask import Flask, request, jsonify
import json

from p2ch13.model_cls import LunaModel

app = Flask(__name__)

model = LunaModel() # ①
model.load_state_dict(torch.load(sys.argv[1], map_location='cpu')['model_state'])
model.eval()

def run_inference(in_tensor):
    with torch.no_grad(): # ②
        # LunaModel takes a batch and outputs a tuple (scores, probs)
        out_tensor = model(in_tensor.unsqueeze(0))[1].squeeze(0)
        probs = out_tensor.tolist()
        out = {'prob_malignant': probs[1]}
        return out

@app.route("/predict", methods=["POST"])
def predict():


```

```

meta = json.load(request.files['meta']) # ③
blob = request.files['blob'].read()
in_tensor = torch.from_numpy(np.frombuffer(blob, dtype=np.float32)) # ④
in_tensor = in_tensor.view(*meta['shape'])
out = run_inference(in_tensor)
return jsonify(out)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8000)
    print(sys.argv[1])

```

- ① Here we set up our model.
- ② No autograd for us.
- ③ Our request will have one file called meta.
- ④ Here we get our data from binary blob to torch.

Run the server as

```
python3 -m p3ch15.flask_server data/part2/models/cls_2019-10-19_15.48.24_final_cls.best.state
```

We prepared a trivial client at `cls_client.py` that sends a single example. From the code directory, you can run it as

```
python3 p3ch15/cls_client.py
```

It should tell you that the nodule is very unlikely to be malignant. Clearly, our server takes inputs, runs them through our model and returns the outputs. So are we done? Not quite. Let us look at what could be better in the next section.

15.1.2 What we wish from Deployment

There are some desiderata for serving models. [5: One of the earliest public talks discussing the inadequacy of Flask serving of PyTorch models is Christian Perone's *PyTorch under the Hood* <https://speakerdeck.com/player/50a597cc8edf42d799bb239d86be3dea>]

The first, we want to support *modern protocols and their features*. Old-school HTTP is deeply serial, which means that when a client wants to send several requests in the same connection, the next requests will only be sent after the previous has been answered. Not very efficient if you want to send a batch of things. We will partially deliver here - our upgrade to Sanic certainly moves us to a framework that has the ambition to be very efficient here.

When using GPUs it is often much more efficient to *batch requests* than process them one-by-one or firing them in parallel. So we have the task to collect requests from several connections, assemble them into a batch to run on the GPU and then get the results back to the respective requestors. This sounds elaborate and (again, when we write this) seems to not be done too often in simple tutorials. That is reason enough for us to do it properly here!

Note, though, that until latency induced by the duration of a model run is an issue (in that waiting for our own run is OK but waiting for the batch running when the request arrives to finish and then for our run to give results is prohibitive), there is little reason to run multiple

batches on one GPU at a given time. Increasing the maximum batch size will generally be more efficient.

We want to serve several things in *parallel*. Even with asynchronous serving, we need our model to run efficiently on a second thread — this means we want to escape the (in)famous Python Global Interpreter Lock (GIL) with our model.

We want to do as *little copying* as possible. Both from a memory consumption and a time perspective, copying things over and over is bad. Many HTTP things are encoded in base64 (a format restricting to 6 bits per byte to encode binary in more or less alphanumeric strings) and, say for images, decoding that to binary and then again to a tensor and then to the batch is clearly relatively expensive. We will partially deliver on this - we use streaming PUT requests to not allocate base64 strings and to avoid growing strings by successively appending to them (which is terrible for performance for strings as much as Tensor s). We say that we do not deliver completely because we are not truly minimizing the copying though.

The last desired thing for serving is *safety*. Ideally we would have a safe decoding. We want to guard both against overflows and the like and resource exhaustion. Once we have a fixed-size input tensor, we should be mostly good, as it is hard to crash PyTorch starting from fixed-sized inputs. The stretch to get there, decoding images and the like, is likely more of a headache and we make no guarantees here. Internet security is a large enough field that we will not cover it at all. We should note, that neural networks are known to be susceptible to manipulation of the inputs to generate desired but wrong or unforeseen outputs [6: Known as *adversarial examples*.], but this isn't extremely pertinent to our application, so we skip it here.

Enough talk, let us improve on our server.

15.1.3 Request Batching

We will do our second example server using the *Sanic* framework (installed via the Python package of the same name). This will give us the ability to serve many requests in parallel using asynchronous processing so we tick off that from our list. While we are at it, we also implement request batching.

Now asynchronous programming can sound scary and usually comes with lots of terminology. But what we are doing here is just to allow functions to non-blockingly wait for results of computation or events (fancy people then call them co-routines).

In order to do request batching, we have to decouple the request handling from the model running. We do so by writing two functions. A Model Runner function will be started at the beginning and run forever.

Whenever we need to run the model, it assembles a batch of inputs, runs the model in a second thread (so other things can happen) and returns the result. The Request Processing then will decode the request, enqueue inputs, wait for the processing to be completed, and return the output with the results. In order to appreciate what *asynchronous* means here, think of the model

runner as a movie on television. The plot will drag on and we are captivated, but in the commercial breaks, we can get up and do something useful, like getting something to drink, er, enqueueing new requests and sending out the answers with the results. Figure 15. 1 has our two functions shown in the blocks we execute uninterrupted before handing back to the event loop.

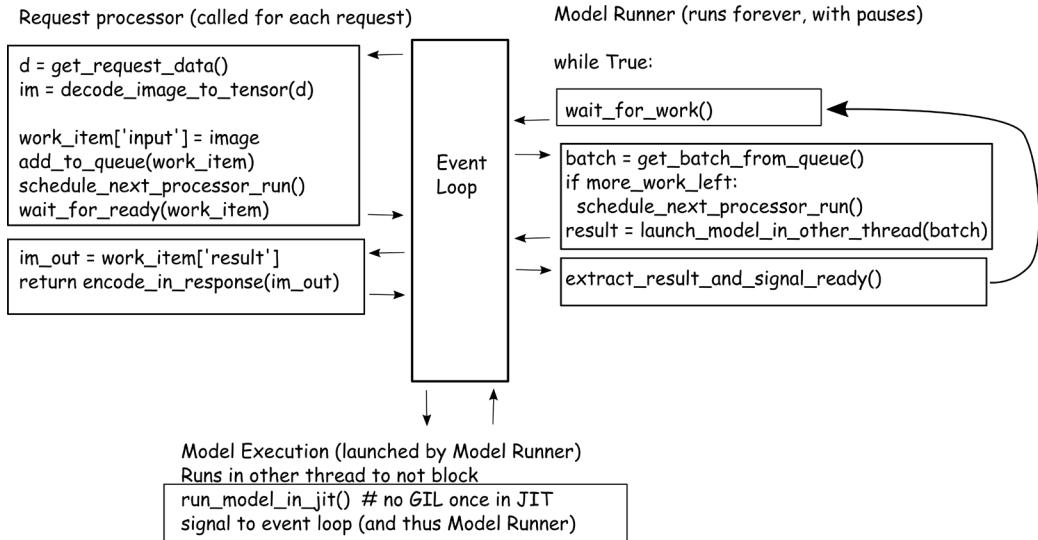


Figure 15.1. Our asynchronous server consists of three blocks - the Request Processor, the Model Runner and the Model Execution. These blocks are a bit like functions, but the first two will yield to the event loop in between.

A slight complication relative to this picture is that we have two occasions at which we need to process events: If we have accumulated a full batch, we start right away, and when the oldest request reaches the maximum wait time, we also want to go. We solve this by setting a timer for the latter. [7: An alternative might be to forego the timer and just run whenever the queue is not empty. This will potentially run smaller "first" batches, but the overall performance impact might not be so large for most applications.]

All our interesting code is in a `ModelRunner` class. First load our model and take care of some administrative things.

In addition to the model, we also need a few additional ingredients. We enter our requests into a queue. This is just a Python list in which we add work items at the back and remove them in the front.

When we modify the `queue`, we want to prevent other tasks from changing the queue from below our feet. To this effect we introduce a `queue_lock` that will be a `asyncio.Lock` provided by the `asyncio` module. As all `asyncio` objects we use here need to know the event loop which

is only available after we initialize the application, we temporarily set it to `None` in the instantiation. While locking like this may not be strictly necessary with our methods not handing back to the event loop while holding the lock and operations on the queue are atomic thanks to the GIL, it does explicitly encode our underlying assumption. If we had multiple workers, we would need to look at locking. One caveat: Python's `async` locks are not threadsafe. (Sigh.)

The `ModelRunner` will wait when it has nothing to do and we will need to signal it from the `RequestProcessor` that it should stop slacking off and get to work. This is done via an `asyncio.Event` called `needs_processing`. The `ModelRunner` will use the `wait()` method to wait for it. The `RequestProcessor` then uses `set()` to signal and the `ModelRunner` wakes up and `clear()`s the event.

Finally, we do need a timer to guarantee a maximal wait time. This timer is created when we need it by using `app.loop.call_at` and sets the `needs_processing` event, we just reserve a slot now. So actually, sometimes the event will be set directly because a batch is complete or when the timer goes off. When we process a batch before the timer goes off, we will clear it to not do too much work.

Listing 15. 3. batching_server.py

```
class ModelRunner:  
    def __init__(self, model_name):  
        self.model_name = model_name  
        self.queue = [] # ①  
        self.queue_lock = None # ②  
        self.model = get_pretrained_model(self.model_name, map_location=device) # ③  
        self.needs_processing = None # ④  
        self.needs_processing_timer = None # ⑤
```

① The queue.

② This will become our lock.

③ Here we load and instantiate the model. This is the (only) thing we will need to change for switching to the JIT. For now, we import the CycleGAN (with the slight modification of standardizing to 0..1 input and output) from `p3ch15/cyclegan.py`.

④ Our signal to run the model.

⑤ Finally, the timer.

Next we need to be able to enqueue requests, the core of the first `RequestProcessor` in Figure 15. 1 (without the decoding and re-encoding). We do this in our first `async` method `process_input`.

We set up a little Python dictionary to hold our task's information: The `input` of course, the `time` it was queued, and an `done_event` to be set when the task has been processed. The processing will add an `output`.

Holding the queue lock (conveniently done in an `async with` block), we add our task to the queue and schedule processing if needed. As a precaution, we error out if the queue has become too large. Then all we have to do is wait for our task to have been processed and return it.

Note that it is important to use the loop time (typically a monotonic clock) which may be different from the `time.time()`. Else we might end up with events being processed before they have been queued or worse.

```
async def process_input(self, input):
    our_task = {"done_event": asyncio.Event(loop=app.loop), # ①
                "input": input,
                "time": app.loop.time()}
    async with self.queue_lock: # ②
        if len(self.queue) >= MAX_QUEUE_SIZE:
            raise HandlingError("I'm too busy", code=503)
        self.queue.append(our_task)
        logger.debug("enqueued task. new queue size {}".format(len(self.queue))) # bookskip
        self.schedule_processing_if_needed() # ③
    await our_task["done_event"].wait() # ④
    return our_task["output"]
```

- ① Here we set up the task data.
- ② With the lock, we add our task and...
- ③ ...schedule processing. Processing will either just set `needs_processing` if we have a full batch. If we don't and no timer is set, it will set one to when the max wait time is up.
- ④ We wait (and hand back to the loop using `await`) for the processing to be done.

This is all we need for the request processing (except decoding and encoding). Let us look at the `model_runner` function on the right hand side of Figure 15.1 doing the model invocation.

As indicated in the figure, `model_runner` is doing some setup and then infinitely looping (but yielding to the event loop in between). It is invoked when the app is instantiated, so it can set up the `queue_lock` and the `needs_processing` event we discussed above. Then it goes into the loop, `await-ing` the `needs_processing` event.

When an event comes, first we check if there is a time set, it clears that, because we'll be processing things now, anyway. Then it grabs a batch from the queue and - if needed - schedules the processing of the next batch. It then assembles the batch from the individual tasks and launches a new thread evaluating the model using `asyncio`'s `app.loop.run_in_executor`. It then adds the outputs to the tasks and sets the `done_event`.

```
async def model_runner(self):
    self.queue_lock = asyncio.Lock(loop=app.loop)
    self.needs_processing = asyncio.Event(loop=app.loop)
    logger.info("started model runner for {}".format(self.model_name)) # bookskip
    while True:
        await self.needs_processing.wait() # ①
        self.needs_processing.clear()
        if self.needs_processing_timer is not None: # ②
            self.needs_processing_timer.cancel()
            self.needs_processing_timer = None
        async with self.queue_lock:
            to_process = self.queue[:MAX_BATCH_SIZE] # ③
            del self.queue[:len(to_process)]
            self.schedule_processing_if_needed()
        # so here we copy, it would be neater to avoid this # bookskip
```

```

batch = torch.stack([t["input"] for t in to_process], dim=0)
# we could delete inputs here...

result = await app.loop.run_in_executor(None, functools.partial(self.run_model, batch)) # ④
for t, r in zip(to_process, result): # ⑤
    t["output"] = r
    t["done_event"].set()
del to_process

```

- ① We wait here until there is something to do.
- ② We cancel the timer if it is set.
- ③ We grab a batch and the schedule the running of the next batch if needed.
- ④ Here we run the model in a separate thread. It will move data to the device and then hand over to the model. We continue processing after it is done.
- ⑤ Finally, we add the results to the work-item and set the ready event.

And that's basically it. The web framework — roughly looking like Flask with `async` and `await` sprinkled in — needs a little wrapper. And we need to start the `model_runner` function on the event loop.

As mentioned above, locking the queue would not be necessary if we do not have multiple runners taking from the queue and potentially interrupting themselves, but knowing our code will be adapted to other projects, we stay on the safe side of losing requests.

We start our server with

```
python3 -m p3ch15.request_batching_server data/p1ch2/horse2zebra_0.4.0.pth
```

Now we can test by uploading the image `data/p1ch2/horse.jpg` and saving the the result:

```
curl -T data/p1ch2/horse.jpg http://localhost:8000/image --output /tmp/res.jpg
```

Note that this server does get a few things right - it batches requests for the GPU, runs asynchronously - but we still use the Python mode and so the GIL hampers running our model in parallel to the request serving in the main thread.

It will not be safe for potentially hostile environments like the internet. In particular, the decoding of request data seems neither optimal in speed nor completely safe.

In general, it would be nicer if we could have decoding where we pass the request stream to a function along with a preallocated memory chunk and the function would decode the image from the stream to us. But we do not know of a library that does things this way.

15.2 Exporting Models

So far, we used PyTorch from the Python interpreter. But this is not always desirable — the GIL is still potentially blocking our improved web-server, we might want to run on embedded where Python is too expensive or unavailable. This is when we export our model. There are several in which we can play this. We might go away from PyTorch entirely and move to more specialized frameworks. Or we might stay within the PyTorch ecosystem and use the JIT. Even when we

then run the JITed model in Python, we might be after two of its advantages: sometimes the JIT enables nifty optimizations or — as in the case of our webserver — just want to escape the GIL, which JITed models do. Finally, but we take some time to get there, we might run our model under the `libtorch`, the C++ library PyTorch offers, or with the derived Torch Mobile.

15.2.1 Interoperability beyond PyTorch with ONNX

Sometimes we want to leave the PyTorch ecosystem with our model in hand — for example to run on embedded hardware with a specialized model deployment pipeline. For this purpose, ONNX (Open Neural Network Exchange) provides an interoperation format for neural networks and machine learning models [8: <https://onnx.ai>]. Once exported, the model can be executed using any ONNX-compatible runtime, such as ONNXRuntime [9: The code lives at <https://github.com/microsoft/onnxruntime>], but be sure to read their privacy statement! Currently, building ONNXRuntime yourself will get you a package that does not send things to the mothership.], provided that the operations in use in our model are supported by the ONNX standard and the target runtime. It is, for example, quite a bit faster on the Raspberry Pi than running PyTorch directly. Beyond traditional hardware, a lot of specialized AI accelerator hardware supports ONNX footnote[<https://onnx.ai/supported-tools.html#deployModel>].

In a way, a deep learning model is a program with a very specific instruction set, made of granular operations like matrix multiplication, convolution, relu, tanh, etc. As such, if we can serialize the computation, we can re-execute it in another runtime that understands its low-level operations. ONNX is a standardization of a format describing those operations and their parameters.

Most of the modern deep learning framework support serialization of their computations to ONNX, some of them can load an ONNX file and execute it (although this is not the case for PyTorch). Some low footprint ("edge") devices accept an ONNX file in input and generate low-level instructions for the specific device. Some cloud computing provides make it now possible to upload an ONNX file and see it exposed through a REST endpoint.

In order to export a model to ONNX, we need to run a model with a dummy input, i.e. the values of the input tensors don't really matter, what matters is that they are of the correct shape and type. By invoking the `torch.onnx.export` function, PyTorch will *trace* the computations performed by the model and serialize them into an ONNX file with the provided name.

```
torch.onnx.export(seg_model, dummy_input, "seg_model.onnx")
```

The resulting ONNX file can now be run in a runtime, compiled to an edge device or uploaded to a cloud service. Using it from Python could be done after installing `onnxruntime` or `onnxruntime-gpu` and getting a batch as a numpy array.

```
import onnxruntime  
  
sess = onnxruntime.InferenceSession("seg_model.onnx") # ①  
input_name = sess.get_inputs()[0].name
```

```
pred_onnx, = sess.run(None, {input_name: batch})
```

- ① The API with sessions and calling run with a set of named inputs might remind you of other deep learning frameworks. But then, we are firmly in static graph land here.

Not all operators of TorchScript can be represented as standardized ONNX operators. If we export operations foreign to ONNX, we will get errors about unknown `aten` operators when trying to use the runtime.

15.2.2 PyTorch's own export — tracing

When interoperability is not the key, but we need to get escape the Python GIL or otherwise export our network, we can use PyTorch's own representation, called the TorchScript graph. We will see what that is and how the JIT that generates works in the next section. But let us give it a spin right here and now.

The simplest way to make a TorchScript model is to trace it. This looks exactly like the ONNX exporting. This isn't surprising, because that is what the ONNX model uses under the hood, too. Here we just feed dummy inputs into the model using the `torch.jit.trace` function. We import the `UNetWrapper` from [Chapter 13], load the trained parameters, set the model to evaluation.

Before we trace the model, there is one additional caveat: We want to make all the parameters to not require gradients. Using the `torch.no_grad()` context manager in tracing will not cause the traced model to not require gradients in the execution. If we look at [four_ways], we see why: When after the model has been traced, PyTorch executes the traced model, it will have parameters requiring gradients, execute the recorded operations, and they will make everything require gradients. To escape that, we would have to run the traced model in a `torch.no_grad` context. To spare us this — from experience, it is easy enough to forget and be surprised by the lack of performance — we loop through the model parameters and set all of them to not require gradients.

But then it is just calling `torch.jit.trace`. [10: Strictly speaking, this trace the model as a function. Recently, PyTorch gained the ability to preserve more of the module structure using `torch.jit.trace_module`, but for us, the plain tracing is sufficient.]

```
import torch
from p2ch13.model_seg import UNetWrapper

seg_dict = torch.load('data-unversioned/part2/models/p2ch13/seg_2019-10-20_15.57.21_none.best.state',
map_location='cpu')
seg_model = UNetWrapper(in_channels=8, n_classes=1, depth=4, wf=3, padding=True, batch_norm=True,
up_mode='upconv')
seg_model.load_state_dict(seg_dict['model_state'])
seg_model.eval()
for p in seg_model.parameters(): # ①
    p.requires_grad_(False)

dummy_input = torch.randn(1, 8, 512, 512)
traced_seg_model = torch.jit.trace(seg_model, dummy_input) # ②
```

- ① We set the parameters to not require gradient here.
- ② The tracing.

The tracing will give us a warning

TracerWarning: Converting a tensor to a Python index might cause the trace to be incorrect. We can't record the data flow of Python values, so this value will be treated as a constant in the future. This means that the trace might not generalize to other inputs!

```
return layer[:, :, diff_y:(diff_y + target_size[0]), diff_x:(diff_x + target_size[1])]
```

this stems from cropping we do in the U-Net, but as long as we only ever plan to feed images of size 512x512 into the model, we will be OK. We'll take a closer look at what causes the warning and how to get around the limitation it highlights if we need to in the next section. It will also be important when we want to convert models that are more complex than CNNs and also UNets to TorchScript.

We can save the traced model

```
torch.jit.save(traced_seg_model, 'traced_seg_model.pt')
```

and load it back without needed anything but the saved file and call it

```
loaded_model = torch.jit.load('traced_seg_model.pt')
prediction = loaded_model(batch)
```

The PyTorch JIT will keep the training / evaluation state and that our parameters do not require gradients. If we had not taken care about it beforehand, we would need to use `with torch.no_grad()` in the execution.

TIP The above exports a JITed PyTorch model and you can run it without keeping the source. However, we always want to establish a workflow where we automatically go from source model to installed JITed model for deployment. If we do not, we will find ourselves in a situation where we would like to tweak something with the model, but have lost the ability to modify and re-generate. Always keep the source, Luke!

15.2.3 Our server with a traced model

Now is a good time to iterate our webserver to what is our final version here.

We can export the CycleGAN traced model with

```
python3 p3ch15/cyclegan.py data/p1ch2/horse2zebra_0.4.0.pth data/p3ch15/traced_zebra_model.pt
```

Now we just need to replace the call to `get_pretrained_model` with `torch.jit.load` in our server (and drop the now unneeded `import of get_pretrained_model`). This also means that our model now runs independent of the GIL and this is what we wanted our server to achieve here.

For your convenience we have put the small modifications in `request_batching_jit_server.py`, do run it with the traced model file path as command line argument.

But now that we have had a taste of what the JIT can do for us, let us dive into the details!

15.3 Interacting with the PyTorch JIT

Debuting in PyTorch 1.0, the PyTorch JIT is at the center of quite a few recent innovations around PyTorch, not least providing a rich set of deployment options.

15.3.1 What to expect from moving beyond classic Python/PyTorch

Quite often, Python is attributed with a lack of speed. While there is some truth to it, the tensor operations we use in PyTorch usually are in themselves large enough that the Python slowness between them is not a large issue. For small devices like smartphones, the memory overhead that Python brings might be more important. So one thing to keep in mind is that quite often, the speedup of taking Python out of the computation might just be 10% or below.

Another immediate speedup of not running the model in Python only appears in multi-threaded environments, but then it can be significant: As the intermediates are not Python objects, the computation is not affected by the menace of all Python parallelization, the GIL. This is what we had in mind above and realized when using a traced model in our server above.

Moving off the classic PyTorch way of executing one operation before looking at the next does give PyTorch a holistic view on the calculation, i.e. it can consider the calculation in its entirety. This opens the door to crucial optimizations and higher-level transformations. Some of those apply mostly to inference while others can also provide a significant speedup in training.

Let us look at a quick example to give you a taste of why looking at several operations at once can be beneficial. When PyTorch runs a sequence of operations on the GPU, it will call a sub-program (kernel in CUDA parlance) for each of them. Every one of these kernels read the input from GPU memory, compute the result, and then store the result. Thus most of the time is typically not spent computing things, but reading from and writing to memory. But this can be improved on by reading only once, computing several operations, and then writing at the very end. This is precisely what the PyTorch JIT Fuser does. This simple trick allows it to significantly narrow the gap between the speed of LSTM and generalized LSTM cells flexibly defined in PyTorch and the rigid, but highly optimized LSTM implementation provided by libraries like CuDNN. To give you a flavour of this Figure 15. 2 shows the pointwise computation taking place in an LSTM cell, a popular building block for recurrent networks. The details are not important to us here, but there are 5 inputs, two outputs and 7 intermediate results. By computing them in one go, the JIT reduces the number of memory reads from 12 to 7 and the number of writes from 9 to 2. These are the large gains the JIT gets us, for LSTMs, it can reduce the time to train an LSTM network by a factor of four.

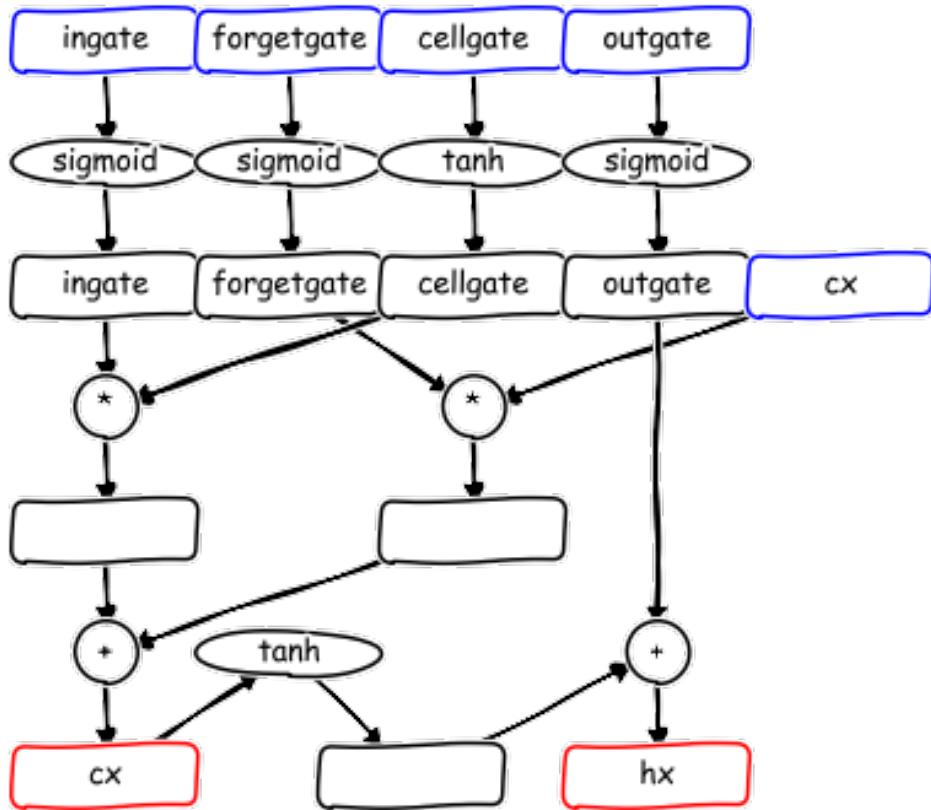


Figure 15. 2. LSTM cell pointwise operations. From five inputs, the four gates at the top and the cx on the right, this block computes two outputs, the hx and cx at the bottom. The boxes in between are intermediate results that vanilla PyTorch will store in memory, but the JIT fuser will just keep in registers.

In summary, the speedup of using the JIT just to escape Python is more modest than we might naively expect when we have been told that Python is awfully slow, but avoiding the GIL is a significant win for multithreaded applications. The large speedups in JITed models come from special optimizations that the JIT enables but which are more elaborate than just avoiding Python overhead.

15.3.2 The dual nature of PyTorch as Interface and Backend

To understand how moving beyond Python works, it is beneficial to mentally separate PyTorch into several parts. We have seen a first glimpse into this in 1.4. Our PyTorch `torch.nn` modules - which we first saw in Chapter 6 and which have been our main tool in modelling ever since - hold the parameters of our network and are implemented using the functional interface - functions taking and returning Tensors. These are implemented as a C++ extension, hand over

to the C++-Level Autograd-Enabled layer. (This then hands the actual computation to an internal library called ATen, performing the computation or relying on backends to do so, but this is not important.)

Given that the C++ functions are already there, the PyTorch developers made them into an official API. This is the nucleus of LibTorch, which allows us to write C++ tensor operations looking almost like their Python counterparts. As the `torch.nn` modules are Python-only by nature, the C++ API mirrors them in a namespace `torch::nn` that is designed to look a lot like the Python part but is independent.

So this would allow us to re-do in C++ what we did in Python. But that is not what we want, we want to *export* the model. Happily, there is another interface to the same functions provided by PyTorch: the PyTorch JIT. The PyTorch JIT provides a "symbolic" representation of the computation. This representation is the TorchScript Intermediate Representation (TorchScript IR sometimes just TorchScript). In the next sections we will see how to get this representation of our Python models, and how they can be saved, loaded, and executed. We already mentioned TorchScript in 1.3.2 when discussing delayed computation.

Similar to what we discussed for the regular PyTorch API, the PyTorch JIT functions to load, inspect, and execute TorchScript modules can also be accessed both from Python and from C++.

In summary we have four ways of calling PyTorch functions — from both C++ and Python we can either call functions directly or have the JIT as an intermediary. All of these eventually call the C++ LibTorch functions and from there ATen and the computational backend.

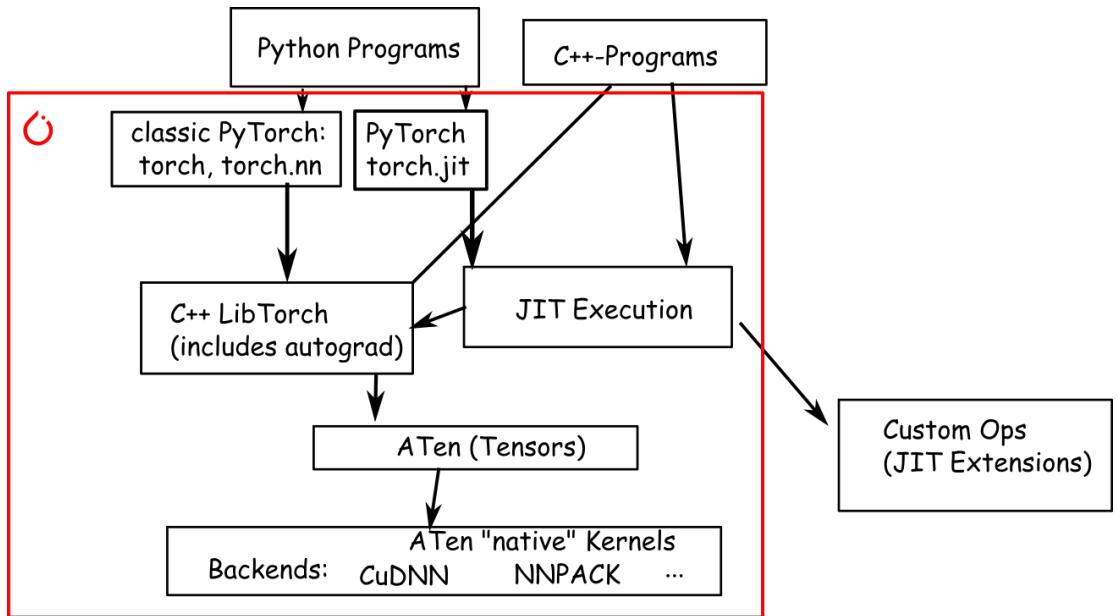


Figure 15. 3. Many ways of calling into PyTorch

15.3.3 TorchScript

TorchScript is at the center of the deployment options envisioned by PyTorch. As such, it is worth taking a close look at how it works.

There are two straightforward ways to create a TorchScript model: Tracing and Scripting. We will look at each of them in the next sections. At a very high level the two work as following:

- In tracing, which we used above, we execute our usual PyTorch model using sample (random) inputs. The PyTorch JIT has hooks (in the C++ autograd interface) for every function that allows it to record the computation. In a way it is like saying *watch me how I compute the outputs — now you can do the same*. Given that the JIT only comes into play when PyTorch functions (and also `nn.Module`s) are called, you can run any Python code while tracing, but the JIT will only notice those bits (and notably no control flow). When we use Tensor shapes — usually a tuple of integers --, the JIT tries to follow what's going on, but may have to give up. This is what gave us the warning when tracing the UNet.
- In scripting, the PyTorch JIT looks at the actual Python code of our computation and compiles it into the TorchScript IR. This means that while we can be sure that every aspect of our program is captured by the JIT, we are restricted to those parts understood by the compiler. This is like saying *I am telling you how to do it — now you do the same*. Sound like programming, really.

But we are not here for theory, let us try that with a very simple function that is just inefficiently adding over the first dimension.

```
# In[2]:  
def myfn(x):\n    y = x[0]\n    for i in range(1 x.size(0)):\n        y = y + x[i]\n    return y
```

We can trace it:

```
# In[3]:  
inp = torch.randn(5,5)  
traced_fn = torch.jit.trace(myfn, inp)  
print(traced_fn.code)  
  
# Out[3]:  
def myfn(x: Tensor) -> Tensor:\n    y = torch.select(x 0 0)\n    y0 = torch.add(y torch.select(x 0 1) alpha=1)\n    y1 = torch.add(y0 torch.select(x 0 2) alpha=1)\n    y2 = torch.add(y1 torch.select(x 0 3) alpha=1)\n    _0 = torch.add(y2 torch.select(x 0 4) alpha=1)\n    return _0
```

/usr/lib/python3/dist-packages/ipykernel_launcher.py:3: TracerWarning: Converting a tensor to a Python index might cause the trace to be incorrect. We can't record the data flow of Python values, so this value will be treated as a constant in the future.

This means that the trace might not generalize to other inputs! This is separate from the ipykernel package so we can avoid doing imports until

We see the big warning and indeed, the code just has the fixed indexing and additions for five rows and would not deal as intended with four or six rows.

This is where scripting helps:

```
# In[4]:  
scripted_fn = torch.jit.script(myfn)  
print(scripted_fn.code)  
  
# Out[4]:  
def myfn(x: Tensor) -> Tensor:\n    y = torch.select(x 0 0)\n    _0 = torch._range_length(1 torch.size(x 0) 1)\n    y0 = y\n    for _1 in range(_0):\n        i = torch._derive_index(_1 1 1)\n        y0 = torch.add(y0 torch.select(x 0 i) alpha=1)\n    return y0
```

We can also print the scripted graph, which is more closely to the internal representation of TorchScript.

```
# In[5]:
```

```
xprint(scripted_fn.graph)

# Out[5]:
graph(%x.1 : Tensor):
    %10 : bool = prim::Constant[value=1]()
    %2 : int = prim::Constant[value=0]()
    %5 : int = prim::Constant[value=1]()
    %y.1 : Tensor = aten::select(%x.1 %2 %2)
    %7 : int = aten::size(%x.1 %2)
    %9 : int = aten::__range_length(%5 %7 %5)
    %y : Tensor = prim::Loop(%9 %10 %y.1)
    block0(%11 : int %y.6 : Tensor):
        %6.1 : int = aten::__derive_index(%11 %5 %5)
        %18 : Tensor = aten::select(%x.1 %2 %6.1)
        %y.3 : Tensor = aten::add(%y.6 %18 %5)
        -> (%10 %y.3)
    return (%y)
```

In practice, you would most often use `torch.jit.script` in the form of a decorator

```
@torch.jit.script
def myfn(x):
...
```

while you could also do this with a custom `trace` decorator taking care of the inputs, this has not caught on.

Although TorchScript (the language) looks like a subset of Python, there are quite fundamental differences. If we look very closely, we see that PyTorch has added type specifications to the code.

This hints at an important difference: TorchScript is statically typed - every value (variable) in the program has one and only one type. Also the types are limited to that the TorchScript IR has a representation for. Within the program, the JIT will usually infer the type automatically, but you need to annotate any non-Tensor arguments of scripted functions with their types. This is in stark contrast to Python, where we can assign anything to any variable.

So far we traced functions to get scripted functions. But we graduated from just using functions in [Chapter 5] to using Module's a long time ago. Sure enough, we can also trace or script models. These will then behave roughly like the `Module's we know and love. For both tracing and scripting, pass an instance of the `Module` to `torch.jit.trace` (with sample inputs) or `torch.jit.script` (without sample inputs), respectively. This will give us the `forward` method we are used to. If we want to expose other methods (this only works in scripting) to be called from the outside, we decorate them with `@torch.jit.export` in the class definition.

TODO: Example?

When we said that the JITed Module's work like they did in Python, this includes that we can use them for training, too. On the flip side, this means that we need to set them up for inference

(e.g. using the `torch.no_grad()` context) just like our traditional models, too, to make them do the right thing.

With algorithmically relatively simple models - like the CycleGAN, classification models and the U-Net-based segmentation - we can just trace the model as we did above. For more complex models, a nifty property is that you can use scripted or traced functions from other scripted or traced code and that you can use scripted or traced submodules when constructing and tracing or scripting a module.

We can also trace functions calling `nn.Models`, but then we need to set all parameters to not require gradients, as the parameters will be constants for the traced model.

As we have seen tracing already, let us look at a practical example of scripting in more detail.

15.3.4 Scripting the gaps of traceability

In more complex models, for example those from the FastRCNN family for detection or recurrent networks used in natural language processing, the bits with control flow like `for` loops need to be scripted. Similarly, if we needed the flexibility, we would find the code bit the tracer warned about:

Listing 15.4. From utils/unet.py

```
class UNetUpBlock(nn.Module):
    ...
    def center_crop(self, layer, target_size):
        _, _, layer_height, layer_width = layer.size()
        diff_y = (layer_height - target_size[0]) // 2
        diff_x = (layer_width - target_size[1]) // 2
        return layer[:, :, diff_y:(diff_y + target_size[0]), diff_x:(diff_x + target_size[1])] # ①

    def forward(self, x, bridge):
        ...
        crop1 = self.center_crop(bridge, up.shape[2:])
        ...

```

① The tracer warns here.

What happens is that the JIT magically replaces the shape tuple `up.shape` with a 1d integer tensor with the same information. Now the slicing `[2:]` and the calculation of `diff_x` and `diff_y` are all traceable tensor operations. That, however, does not save us, because the slicing then wants Python int's and there the reach of the JIT ends, giving us the warning.

But we can solve it in a straightforward way: We script `center_crop`. We slightly change the cut between caller and callee by passing `up` to the scripted `center_crop` and extracting the sizes there. Other than that, all we need is to put the `@torch.jit.script` decorator. We get the following code that makes the UNet model traceable without warnings:

```
@torch.jit.script
def center_crop(layer, target):
```

```

    ... , layer_height, layer_width = layer.size()
    ... , target_height, target_width = target.size()
    diff_y = (layer_height - target_height) // 2
    diff_x = (layer_width - target_width) // 2
    return layer[:, :, diff_y:(diff_y + target_height), diff_x:(diff_x + target_width)]
```

```

class UNetUpBlock(nn.Module):
    ...
    def forward(self, x, bridge):
        ...
        crop1 = center_crop(bridge, up)
        ...

```

Another option we could take - but which we will not take here - is to move un-scriptable things into custom operators implemented in C++. The TorchVision library does that for some speciality operations in the MaskRCNN models.

15.4 LibTorch – PyTorch in C++

So we have seen various way to export our models, but so far, we have been with Python. We now look at how we can forego python and work with C++ directly.

Let us go back to the horse to zebra CycleGAN example. We will now take the JITed model from Section 1.2.3 and run it from a C++ program.

15.4.1 Running JITed models from C++

The hardest part about deploying PyTorch vision models in C++ is choosing an image library to choose the data [11: But TorchVision might develop a convenience function for it]. Here, we go with the very lightweight libary CImg [12: <http://cimg.eu/>]. If you are very familiar with OpenCV, do adapt the code to use that instead, we just felt that CImg is easiest for our exposition.

Running a JITed model is very simple, we first show the image handling.

As the image handling is not really what we are after, we will do this very quickly. For the PyTorch side, we will include a C++ header `torch/script.h`. Then we need to set up and include the `CImg` library.

In the `main` function, we load an image from a file given on the command line and resize it (in CImg). So we now have a 227x227 image in the `CImg<float>` variable `image`.

At the end of the program, we create an `out_img` of the same type from our `(1, 3, 277, 277)`-shaped tensor and save it.

Don't worry about these bits. They are not the PyTorch C++ we want to learn so we can just take them as is.

Listing 15. 5. cyclegan_jit.cpp

```
#include "torch/script.h" // #①
#define cimg_use_jpeg
#include "CImg.h"
using namespace cimg_library;

int main(int argc, char **argv) {
    CImg<float> image(argv[2]); // #②
    image = image.resize(227, 227); // #③
    // ...here we need to produce an output tensor from input
    CImg<float> out_img(output.data_ptr<float>(), output.size(2), // #④
                         output.size(3), 1, output.size(1));
    out_img.save(argv[3]); // #⑤
    return 0;
}
```

- ① We include the PyTorch script header and CImg with native JPEG support.
- ② We load and decode the image into a float array.
- ③ This resizes to a smaller size.
- ④ The method `data_ptr<float>()` gives us a pointer to the tensor storage. With it and the shape information, we can construct the output image.
- ⑤ Finally, we save the image.

The actual computation is straightforward, too. We need to make an input tensor from the image, load our model, and run the input tensor through it.

So here is our plan. Recall from [Chapter 3] that PyTorch keeps the values of a tensor in a large chunk of memory in a particular order. So does `CImg`, and we can get a (`float`) pointer to the this memory chunk using `image.data()` and the number of elements using `image.size()`. With these two, we can create a somewhat smarter reference, a `torch::ArrayRef` (which is just a shorthand for pointer plus size, PyTorch uses those at the C++ level for data but also for returning sizes without copying). And that we can just parse into the `torch::tensor` constructor, just like we would with a list. Sometimes, you might want to use the similar-working `torch::from_blob`. The difference is that `tensor` will copy the data and you do not need to take care the underpinning memory is available during the lifetime of the tensor.

Now our tensor is only 1d, so we need to reshape it. Conveniently, CImg uses the same ordering (channel, rows, columns) just as PyTorch does. If not we would need to adapt the reshaping and permute the axes like we did in [Chapter 4]. As CImg uses a range of 0...255 and we made our model to use 0...1, we divide here and multiply below. This could, of course, be absorbed into the model, but we wanted to re-use our traced model.

NOTE When switching libraries, it is easy to forget these conversion steps. They are non-obvious unless we look up the memory layout and scaling convention of PyTorch and the image processing library you use. If we forget, we will be disappointed by not getting the results we anticipate. Here, the model would go wild because it gets extremely large inputs. However, at the end, the outputs of our model would be in the 0..1 range by the final squashing. This would look all black. Other frameworks have other conventions, e.g. OpenCV likes to store

images as BGR instead of RGB, requiring us to flip the channel dimension. We always want to make sure that the input we feed to the model in the deployment is the same as the one we fed into it in Python.

Loading the traced model is very straightforward using `torch::jit::load`.

Next we have to deal with an abstraction PyTorch introduces to bridge between Python and C++: We need to wrap our input in an `IValue` (or more), the generic datatype for any value. A function in the jit is passed a vector of `IValue`'s, so we declare that and then `push_back` our input tensor. This will automatically wrap our tensor into an `IValue`. We feed a vector of them to the forward and get a single one back. We can then unpack the Tensor in the result `IValue` with `.toTensor`.

Here we see a bit of `IValue`'s: They do have a type (here `Tensor`) but they could also be holding `int64_t` or `double`s or a list of `Tensor`'s. E.g. had we multiple outputs, we would get an `IValue` holding a list of `Tensor`'s, which ultimately stems from the Python calling conventions. When we unpack a `Tensor` from an `IValue` using `.toTensor`, the `IValue` will transfer ownership, i.e. become invalid. But let's not worry about it, we got a `Tensor` back. Because sometimes the model might return non-contiguous data (with gaps in the storage of [Chapter 3]) but `CImg` reasonably requires us to provide it with a contiguous block, we call `contiguous`. It is important that we assign this contiguous `Tensor` to a variable that is in scope until we are done working with the underlying memory. — Just like in Python, PyTorch will free memory if it sees no tensors using it anymore.

Listing 15. 6. cyclegan_jit.cpp

```
auto input_ = torch::tensor(torch::ArrayRef<float>(image.data(),
    image.size())); // # ①
auto input = input_.reshape({1, 3, image.height(), image.width()}).div_(255); // # ②
auto module = torch::jit::load(argv[1]); // # ③
std::vector<torch::jit::IValue> inputs; // # ④
inputs.push_back(input);
auto output_ = module.forward(inputs).toTensor(); // # ⑤
auto output = output_.contiguous().mul_(255); // # ⑥
```

- ① Converting to a tensor.
- ② Reshaping and re-scaling to move from `CImg` conventions to PyTorch's.
- ③ This load the JITed model or function from file.
- ④ Here we pack the input into a (one-element) vector of `IValue`s.
- ⑤ We call the module and extract the result tensor. For efficiency, the ownership is moved, so if we held on to the `IValue`, it would be empty afterwards.
- ⑥ Make sure our result is contiguous.

So let us compile this! On Debian or Ubuntu, you need to install `cimg-dev`, `libjpeg-dev`, `libx11-dev` to use `CImg`.

You can download a C library of PyTorch from the PyTorch page. But given that we already have PyTorch installed footnote:[We hope you have not been slacking off with trying out things you

read.], we might as well use that, it comes with all we need for C. We need to know where our PyTorch installation lives, so open Python and check `torch.+file+`, which might say

`/usr/local/lib/python3.7/dist-packages/torch/init.py`. This means that the CMake files we need are in `/usr/local/lib/python3.7/dist-packages/torch/share/cmake/`.

While using CMake seems overkill for a single source file project, linking to PyTorch is a bit complex and so we just use the following as a boilerplate cmake file. [13: The code directory actually has a bit longer version to work around Windows issues.]

Listing 15. 7. CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0 FATAL_ERROR)
project(cyclegan-jit) # ①

find_package(Torch REQUIRED) # ②
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${TORCH_CXX_FLAGS}")

add_executable(cyclegan-jit cyclegan_jit.cpp) # ③
target_link_libraries(cyclegan-jit pthread jpeg X11) # ④
target_link_libraries(cyclegan-jit "${TORCH_LIBRARIES}")
set_property(TARGET cyclegan-jit PROPERTY CXX_STANDARD 14)
```

① The project name, replace with your own, but also do so below.

② We need Torch.

③ This is our source file.

④ And here we link to the bits required for Clmg. Clmg itself is all-include, so it does not appear here.

It is best to make a `build` dir as a subdirectory of and then in it run CMake as `CMAKE_PREFIX_PATH=/usr/local/lib/python3.7/dist-packages/torch/share/cmake/cmake ...` and finally `make`. This will build our `cyclegan-jit` program which we can then run as

```
./cyclegan-jit ..//traced_zebra_model.pt      ..//data/p1ch2/horse.jpg /tmp/z.jpg
```

So we just ran our PyTorch model without Python. Awesome! If you want to ship your application, you likely want to copy the libraries from `/usr/local/lib/python3.7/dist-packages/torch/lib` into where your executable lies so that they will always be found.

15.4.2 C++ from the start – the C++ API

In order to get a taste of the C++ modular API — which is intended to feel a lot like the Python one — we will translate the CycleGAN generator into a model natively defined in C++ — without the JIT. We do, however, need the pretrained weights, and so we save a traced version of the model (and here it is important not to trace a function but really the model!).

We first import the one-stop `torch/torch.h` header and Clmg. As spelling out `torch::Tensor` can be tedious, we import the name into the main namespace.

Listing 15. 8. cyclegan_cpp_api.cpp

```
#include <torch/torch.h>
```

```
#define cimg_use_jpeg
#include <Clmg.h>
using torch::Tensor;
```

When we look at the source code in the file, we find that `ConvTransposed2d` is ad-hoc defined when ideally they should be taken from the standard library. The matter here is that the C++ modular API is still under development and with PyTorch 1.4 the pre-made `ConvTranspose2d` module cannot be used in `Sequential` because it takes an optional second argument. [14: This is a great improvement over PyTorch 1.3 where we needed to implement custom modules for `ReLU`, `InstanceNorm2d`, and others.] Usually we could just leave `Sequential` — as we did for Python — but we want to stay our model to have the same structure as the Python CycleGAN generator from [Chapter 2].

Let us look at the residual block. Just like we would in Python, we register a subclass of `torch::nn::Module`. Our residual block has a sequential `conv_block` submodule.

TODO: Diagram of the network?

Just like we did in Python, we need to initialize our submodules, notably the `Sequential`. We do so using the C++ initialization statement. This is similar to how we construct them in Python in the `__init__` constructor. Unlike Python, C++ does not have the introspection and hooking capabilities enabling redirection of `setattr` to combine assignment to a member and registration.

Since the lack of keyword arguments makes the parameter specification a bit awkward with default arguments, modules (like tensor factory functions) typically take an *options* argument. Optional keyword arguments in Python correspond to methods of the options object that we can chain. For example, the Python module `nn.Conv2d(in_channels, out_channels, kernel_size, stride=2, padding=1)` we need to convert below translates to `torch::nn::Conv2d(torch::nn::Conv2dOptions(in_channels, out_channels, kernel_size).stride(2).padding(1))`. A bit more tedious, but if you're reading this because you love C++ and aren't deterred by the hoops it makes you jump through.

We should always take care that registration and assignment to members is in sync or things will not work as expected: for example, loading and updating parameters during training will happen to the registered module but the actual module being called is a member. This synchronization was done behind the scenes by the Python `nn.Module` class, but it is not automatic in C++. Failing to do so will cause us many headaches.

In contrast to how we did (and should!) in Python, we need to call `m->forward(...)` for our modules. It seems some modules can also be called directly, but for `Sequential`, this is not the case.

A final comment on calling conventions is in order: Depending on whether you modify tensor provided to functions [15: This is a bit blurry because you can create a new tensor sharing memory and in-place that, but it's best to avoid that if possible.], tensor arguments should

always be passed as `const Tensor&` for tensors that are left unchanged or `Tensor` if they are changed. Tensors should be returned as `Tensor`. Wrong argument types like non-const references (`Tensor&`) will lead to unparseable compiler errors.

```
struct ResNetBlock : torch::nn::Module {
    torch::nn::Sequential conv_block;
    ResNetBlock(int64_t dim)
        : conv_block( // # ①
            torch::nn::ReflectionPad2d(1),
            torch::nn::Conv2d(torch::nn::Conv2dOptions(dim, dim, 3)),
            torch::nn::InstanceNorm2d(torch::nn::InstanceNorm2dOptions(dim)),
            torch::nn::ReLU(/*inplace=*/true), torch::nn::ReflectionPad2d(1),
            torch::nn::Conv2d(torch::nn::Conv2dOptions(dim, dim, 3)),
            torch::nn::InstanceNorm2d(torch::nn::InstanceNorm2dOptions(dim))) {
        register_module("conv_block", conv_block); // # ②
    }

    Tensor forward(const Tensor &inp) {
        return inp + conv_block->forward(inp); // # ③
    }
};
```

- ① Here we initialize `Sequential` including its submodules.
- ② We always want to remember to register the modules we assigned or bad things will happen!
- ③ As might be expected, our `forward` function is pretty simple.

In the main generator class, we follow a typical pattern in the C++ API more closely by naming our class `ResNetGeneratorImpl` and promoting it to a torch module `ResNetGenerator` using the `TORCH_MODULE` macro. The background for this is that we want to mostly handle modules as references or shared pointers. The wrapped class accomplishes this.

```
struct ResNetGeneratorImpl : torch::nn::Module {
    torch::nn::Sequential model;
    ResNetGeneratorImpl(int64_t input_nc = 3, int64_t output_nc = 3,
        int64_t ngf = 64, int64_t n_blocks = 9) {
        TORCH_CHECK(n_blocks >= 0);
        model->push_back(torch::nn::ReflectionPad2d(3)); // # ①
        ... // # ②
        model->push_back(torch::nn::Conv2d(
            torch::nn::Conv2dOptions(ngf * mult, ngf * mult * 2, 3)
                .stride(2)
                .padding(1))); // # ③
        ...
        register_module("model", model);
    }
    Tensor forward(const Tensor &inp) { return model->forward(inp); }
};

TORCH_MODULE(ResNetGenerator); // # ④
```

- ① Here we add modules to the `Sequential` container in the constructor. This allows us to add a variable number of modules in a for loop.
- ② We are sparing us to reproduce some tedious things here.
- ③ Here we have an example of `Options` in action.

- ④ This creates a wrapper `ResNetGenerator` around our `ResNetGeneratorImpl` class. As archaic as it seems, the matching names are important here.

That's it, we defined the perfect C++ analogous of the Python `ResNetGenerator` model. Now we only need a bit of a main function to load parameters and run our model. The loading of the image with `CImg` and the conversion from image to tensor and tensor back to image will be the same as in the previous section. To include some variation, we display the image instead of writing it to disk.

The interesting changes are in how we create and run the model. Just as one would expect, we instantiate the model by declaring a variable of the model type. We load the model using `torch::load` (here it is important that we wrapped the model above). While this looks very similar to PyTorch practitioners, note that it will work on JIT-saved files rather than Python-serialized state dictionaries.

When running the model, we need the equivalent of `with torch.no_grad():`. This is provided by instantiating a variable of type `NoGradGuard` and keeping it in scope for as long as we do not want gradients. Just like in Python, we set the model into evaluation mode calling `model->eval()`. This time around, we call the `model->forward` with our input `Tensor` and get a `Tensor` as a result - no JIT is involved, so we do not need `IValue` packing and unpacking.

```
ResNetGenerator model; // # ①
...
torch::load(model, argv[1]); // # ②
...
cimg_library::CImg<float> image(argv[2]);
image.resize(400, 400);
auto input_ =
    torch::tensor(torch::ArrayRef<float>(image.data(), image.size()));
auto input_ = input_.reshape({1, 3, image.height(), image.width()});
torch::NoGradGuard no_grad; // # ③
model->eval(); // # ④
auto output = model->forward(input); // # ⑤
...
cimg_library::CImg<float> out_img(output.data_ptr<float>(), output.size(3),
                                     output.size(2), 1, output.size(1));
cimg_library::CImgDisplay disp(out_img, "See a C++ API zebra!"); // # ⑥
while (!disp.is_closed()) {
    disp.wait();
}
```

- ① We instantiate our model.
- ② Here we load the parameters.
- ③ Declaring a guard variable is the equivalent of the `torch.no_grad()` context. We could put it in a `{ ... }` block if you need to limit how long you turn off gradients.
- ④ As in Python, `eval` mode is turned on (for our model it would not be strictly relevant).
- ⑤ Again, we call forward rather than the model.
- ⑥ Displaying the image, we need to wait for a key rather than immediately exiting our program.

Phew. Writing this in C++ was a lot of work for the Python fans that we are. We are glad that we only promised to do inference here, but of course LibTorch also offers optimizers, dataloaders and much more. The main reason to use the API is, of course, when you want to create models and neither the JIT nor Python are a good fit.

For your convenience, the `CMakeLists.txt` contains also the instructions for building `cyclegan-cpp-api`, so building is just like in the previous section.

Run as

```
./cyclegan_cpp_api ..//traced_zebra_model.pt ..//data/p1ch2/horse.jpg
```

But we knew what the model would be doing, didn't we?

15.5 Emerging Technology: Enterprise serving of PyTorch models

Now, we may ask ourselves whether all of the deployment things we discussed so far should involve as much coding as they do. Sure it is a common enough thing for someone to have coded all that. As of very early 2020, while we are busy with the finishing touches to the book, we have great expectations for the near future, but at the same time we feel that the deployment landscape will significantly change by the summer.

Currently, there is RedisAI [16: <https://github.com/RedisAI/redisai-py>] (that one of the authors is involved with) waiting to apply Redis goodness to our models. Work is underway on PyTorch Serving [17: Development discussion is at <https://github.com/pytorch/pytorch/issues/27610>]. Similarly, MLFlow [18: <https://mlflow.org/>] is building out more and more support. For the more specific tasks of information retrieval, there also is EuclidesDB [19: <https://euclidesdb.readthedocs.io/en/latest/>] to do AI-based feature databases.

Exiting times, but unfortunately, they did not sync with our writing schedule. We might hope to have more to tell in the second edition or second volume...

15.6 Going mobile

As the last variant of deploying a model, we will consider deployment to mobile devices. When we want to bring our models to mobile, we are typically looking at Android and / or iOS. Here, we focus on Android.

The C++ parts of PyTorch, LibTorch, can be compiled for Android and we could access that from an app written in Java using the Android Java Native Interface (JNI). But we really only need a handful of functions from PyTorch - loading a JITed model, making inputs into `Tensor's and `IValue's, running them through the model, and getting results back. To save us the trouble of using the JNI, the PyTorch developers wrapped these functions into a small library as PyTorch Mobile.

Now, the stock way of developing apps in Android is in the AndroidStudio IDE and we will be using it, too. But this means that there are a few dozen files of administrativa — which also happen to change from one Android version to the next. As such, we focus on the bits that turn one of the Android Studio templates (Java App with empty Activity) into an app taking a picture and running it through our Zebra-CycleGAN and display the result. Sticking with the theme of the book, we will be efficient with the Android bits (and they can be painful compared with writing PyTorch code) in the example app.

To infuse life into the template, we need to do three things:

We need to define a UI. To keep things as simple as we can, we have two elements: the first is a `TextView` named `headline` which we can click to take and transform a picture. Secondly, we need an `ImageView` to show our picture, which we call `image_view`. We will leave the picture-taking to the camera app, something which you would likely avoid in your app for a smoother user experience, because dealing with the camera directly would blur our focus on deploying PyTorch models. [20: We are very proud of the topical metaphor.]

Then we need to include PyTorch as a dependency. This is done by editing our app's `build.gradle` file and adding `pytorch_android` and `pytorch_android_torchvision`.

Listing 15.9. Additions to build.gradle

```
dependencies { // # ①
    ...
    implementation 'org.pytorch:pytorch_android:1.4.0' // # ②
    implementation 'org.pytorch:pytorch_android_torchvision:1.4.0' // # ③
}
```

① The `dependencies` section is very likely already there, if not add it at the bottom.

② The `pytorch_android` library will get us the core things mentioned above and

③ the helper library `pytorch_android_torchvision` — perhaps a bit immodestly named when compared to its larger TorchVision sibling — contains a few utilities to convert `Bitmap` objects to `Tensor`s, but at the time of writing not much more.

We need to add our traced model as an asset.

Finally we can get to the meat of our shiny app, the Java class derived from activity that contains our main code. We just discuss an excerpt here. It starts with imports and model setup.

We need some imports from the `org.pytorch` namespace. In the typical style that is a hallmark of Java, we import `IValue`, `Module`, `Tensor` which do what we might expect and the class `org.pytorch.torchvision.TensorImageUtils` holding utility functions to convert between 'Tensor's and 'Image's.

First, of course, we need to declare a variable holding our model. Then, when our app is started — namely in `onCreate` of our activity — we load the module using the `Model.load` method from the location given as an argument. There is a slight complication though: apps have their data provided by the supplier as `assets` which are not easily accessible from the

filesystem. For this reason a utility method `assetFilePath` (taken from the PyTorch Android examples) copies the asset to a location in the filesystem.

Finally, in Java we need to catch exceptions our code may throw unless we want (and are able to) declare the method we are coding as throwing them in turn.

In code this looks like

Listing 15. 10. MainActivity.java part 1

```
...
import org.pytorch.IValue; // # ①
import org.pytorch.Module;
import org.pytorch.Tensor;
import org.pytorch.torchvision.TensorImageUtils;
...
public class MainActivity extends AppCompatActivity {
    private org.pytorch.Module model; // # ②

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        try { // # ③
            model = Module.load(assetFilePath(this, "traced_zebra_model.pt")); // # ④
        } catch (IOException e) {
            Log.e("Zebraify", "Error reading assets", e);
            finish();
        }
        ...
    }
    ...
}
```

① Don't you love imports?

② this will hold our JITed model.

③ In Java we have to catch the exceptions.

④ This loads the module from a file.

When we get an image from the camera app using Android's `Intent` mechanism, we need to run it through our model and display it. This happens in the `onActivityResult` event handler.

Converting the bitmap we get from Android to a tensor is handled by the `TensorImageUtils.bitmapToFloat32Tensor` function (static method) which takes two float arrays of `means` and `stds` in addition to the `bitmap`. Here we specify the mean and std of our input data(set) which will then be mapped to have zero mean and unit standard deviation just like TorchVision's `Normalize` transform. Android already gives us the images in the 0..1 range that we need to feed into our model, so we specify mean 0 and standard deviation 1 to make the normalization not change our image.

Around the actual call to `model.forward` we then do the same `IValue` wrapping and unwrapping dance as we did when using the JIT in C++ except now our `forward` takes a single `IValue` rather than a vector of them.

Finally we need to get back to a `Bitmap`. Here PyTorch will not help us, so we need to define our own `tensorToBitmap` (and submit that pull request to PyTorch). We spare you the details here, as it is tedious and full of copying (from the `Tensor` to a `float[]` array to a `int[]` array containing ARGB values to the bitmap), but so it is. It is designed to be the inverse of `bitmapToFloat32Tensor`.

Listing 15. 11. MainActivity.java part 2

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == REQUEST_IMAGE_CAPTURE && resultCode == RESULT_OK) {
        // this gets called when the camera app got a picture
        Bitmap bitmap = (Bitmap) data.getExtras().get("data");

        final float[] means = {0.0f, 0.0f, 0.0f}; // # ①
        final float[] stds = {1.0f, 1.0f, 1.0f};

        final Tensor inputTensor = TensorImageUtils.bitmapToFloat32Tensor(bitmap, // # ②
            means, stds);

        final Tensor outputTensor = model.forward(IValue.from(inputTensor)).toTensor(); // # ③
        Bitmap output_bitmap = tensorToBitmap(outputTensor, means, stds, Bitmap.Config.RGB_565); // # ④
        image_view.setImageBitmap(output_bitmap);
    }
}
```

- ① Here we do normalization, but the default is images in the range of 0...1 so we do not need to transform, i.e. have 0 shift and a scaling divisor of 1.
- ② This gets us a Tensor from a bitmap, combining steps like TorchVision's `ToTensor` (converting to a float Tensor with entries between 0 and 1) and `Normalize`.
- ③ This looks almost like what we did in C++.
- ④ This `tensorToBitmap` is our own invention.

And that's all we need to do to get PyTorch into Android. Using the minimal additions to the code we left out here to request a picture, we have a Zebraify Android app. Well done! [21: At the time of writing this, PyTorch Mobile is still relatively young and you might hit rough edges. On Pytorch 1.3, the colors were off on an actual 32 bit ARM phone while working in the emulator. The reason is likely a bug in one of the computational backend functions that are only used on ARM. With PyTorch 1.4 and a newer phone (64 bit ARM), it seemed to work better.]



Figure 15. 4. Our CycleGAN Zebra App

We should note that the above gets us a full version of PyTorch with all ops on Android. This will, in general, also include operations you will not need for a given task, leading to the question if we would save some space by leaving them out. It turns out that starting with PyTorch 1.4 you can by building a customized version of the PyTorch library to use. [22: See <https://pytorch.org/mobile/android/#custom-build>]

15.6.1 Improving efficiency: Model design and quantization

If we wanted to explore mobile in more detail, our next step would be to try and make our models faster.

When we wish to reduce the memory and compute footprint of our models the first thing to look at are streamlining model itself, i.e. computing the same or very similar mappings from inputs to outputs with fewer parameters and operations. This is often called distillation. The details of distillation vary - sometimes we try to shrink each weight by eliminating small or irrelevant weights [23: Lottery Ticket and WaveRNN], in other examples, we combine several layers of a net into one [24: DistilBERT] or even train a fully different, simpler model to reproduce the larger model's outputs [25: OpenNMT's original CTranslate]. We mention this because these modifications are likely to be the first step to getting models to run faster.

Another way to is to reduce the footprint of each parameter and operation: instead of expending the usual 32 bit per parameter in form of a float, we convert our model to work with integers, a typical choice is 8 bit. This is quantization. [26: In contrast to quantization, (partially) moving to 16 bit floating-point for training which usually called reduced or (if some bits stay 32 bit) mixed- precision training.]

PyTorch does offer us quantized tensors for this purpose. It is exposed as a set of scalar types similar to `torch.float`, `torch.double`, `torch.long` (compare [Ch 3 Section Data types]). The most common quantized tensor scalar types are `torch.quint8` and `torch.qint8`, representing numbers in unsigned and signed 8 bit integers, respectively. PyTorch uses a separate scalar type here in order to use the dispatch mechanism we briefly looked at in [Ch 3 Dispatch].

It might seem surprising that using 8 bit integers instead of 32 bit floating points even works at all and typically there is only a slight degradation in results, but not much. There are two things that seem to contribute: If we consider rounding errors as essentially random and convolutions and

linear layers as weighted averages, we may expect that rounding errors typically cancel. [27: The fancy people would refer to the Central Limit Theorem here. And indeed care must be taken that the independence (in the statistical sense) of rounding errors is preserved. For example, one usually wants that 0 (a prominent output of ReLU) is exactly representable. Because otherwise all these zeros would be changed by the exact same quantity in rounding, leading to errors adding up rather than cancelling.] This seems to allow reducing the relative precision from more than twenty bits in 32 bit floating points to the 7 bits that signed integers offer. The other thing quantization does (in contrast to training with 16 bit floating point) is to move from floating point to fixed precision (per tensor or channel). This means that the largest values are resolved to 7 bit precision and the values that are one eighth of them only to $7-3 = 4$ bits. But if things like L1-Regularization (briefly mentioned in [Chapter 8]) work, we might hope similar effects allow us to afford less precision for the smaller values in our weights when quantizing. Apparently they do.

Quantization debuted with PyTorch 1.3 and is still a bit rough in terms of supported operations in PyTorch 1.4. It is rapidly maturing out though, and we recommend to check it out if you are serious about computationally efficient deployment.

15.7 Conclusion

This concludes our short tour of how to get our models out to where we want to apply them. While the ready-made torch serving has been not quite there yet when we wrote this, when it arrives we will likely want to export our model through the JIT, so it's good we looked at it. In the meantime, you now know how to deploy your model to a network service, in a C++ application, or on mobile.

We look forward to see what you will build with it!

15.8 Exercises

As we close out Deep Learning with PyTorch, we have one final exercise for you:

Pick a project that sounds exciting to you. Kaggle is a great place to start looking.

Dive in.

You have acquired the skills and learned the tools you need to succeed. We can't wait to hear what you do next; drop us a line on the book's forum and let us know!

15.9 Summary

- We can serve PyTorch models by wrapping them in a Python webserver framework such as flask.
- By using JITed models, we can avoid the GIL even when calling them from Python, which is a good idea for serving.
- Request batching and asynchronous processing helps use resources efficiently, in particular when inference is on the GPU.
- For beyond-PyTorch exporting of models ONNX is a great format, ONNXRuntime provides a backend for many purposes, including the Raspberry Pi.
- The JIT allows you to export and run arbitrary Pytorch code in C++ or on mobile with little effort.
- Tracing is the easiest way to get JITed models, you might need to use scripting for some particularly dynamic parts.
- There also is good C++ (and an increasing number of other languages) support.

Notes

1. We also recommended www.arxiv-sanity.com/ to help organize research papers of interest.
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).
3. 5.1
4. [PyTorch's Autograd: Back-propagate all things](#)
5. 5.1.5
6. www.geforce.com/hardware/technology/cuda
7. dawn.cs.stanford.edu/benchmark/index.html
8. pytorch.org/get-started/locally/
9. jupyter.org
10. forums.manning.com/forums/deep-learning-with-pytorch
11. github.com/deep-learning-with-pytorch/dlwpt-code
12. imagenet.stanford.edu
13. wordnet.princeton.edu
14. github.com/pytorch/vision
15. papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf
16. arxiv.org/pdf/1512.03385.pdf
17. arxiv.org/pdf/1512.00567.pdf
18. pillow.readthedocs.io/en/stable/
19. pytorch.org/docs/stable/torch.nn.html#torch.nn.functional.softmax

A relevant example is described here:

20. www.vox.com/2018/4/18/17252410/jordan-peele-obama-deepfake-buzzfeed (warning: coarse language)

21. We maintain a clone of the code at github.com/deep-learning-with-pytorch/ImageCaptioning.pytorch

Andrej Karpathy and Li Fei-Fei, Deep Visual-Semantic Alignments for Generating Image Descriptions,

22. cs.stanford.edu/people/karpathy/cvpr2015.pdf

23. Contact the publisher for franchise opportunities!

24. www.scipy.org/

25. scikit-learn.org/stable/

26. pandas.pydata.org/

As perception is not trivial to norm, there are many weights people have come up with, see e.g. Wikipedia:

27. Luma (Video)

28. Tim Rocktäschel's blog rockt.github.io/2018/04/30/einsum gives a good overview

29. Sasha Rush's Tensor Considered Harmful nlp.seas.harvard.edu/NamedTensor

pytorch.org/tutorials/intermediate/named_tensor_tutorial.html

and

30. pytorch.org/docs/stable/named_tensor.html

31. And signed-ness, in the case of uint8.

32. pytorch.org/docs/

Future PyTorch releases might make Storage not directly accessible, but what we show here still provides a

33. good mental picture of how tensors work under the hood.

34. docs.python.org/3/c-api/buffer.html

35. rocm.github.io/

Before the regular build process, you need to run `tools/amd_build/build_amd.py` to translate the GPU

36. code.

37. github.com/pytorch/xla

38. colab.research.google.com/
39. www.hdfgroup.org/solutions/hdf5/
40. www.h5py.org/
41. en.wikipedia.org/wiki/Convolutional_neural_network#History
42. Something of an understatement: en.wikipedia.org/wiki/Color_model

For many purposes, using TorchVision is a great default choice to deal with image and video data. We go
43. with `imageio` here for somewhat lighter exploration.

44. wiki.cancerimagingarchive.net/display/Public/CPTAC-LSCC#dd4a08a246524596add33b9f8f00f288
45. As a starting point for a more indepth discussion, refer to en.wikipedia.org/wiki/Level_of_measurement
46. archive.ics.uci.edu/ml/datasets/bike+sharing+dataset

This could also be a case where it is useful to go beyond the main path. Speculatively, we could also try to reflect *like categorical, but with order* more directly by generalizing one-hot encodings to mapping the i-th of our four categories here to a vector that has ones in the positions 0...i and zeros beyond that. Or - similar to the embeddings we discuss in 4.5 - we could take partial sums of embeddings, in which case it might make sense to make those positive. Like with many things you encounter in practical work, this could be a
47. place where *trying what works for others* and then experimenting in a systematic fashion is a good idea.

- Nadkarni et al. Natural language processing: an introduction. JAMIA
48. www.ncbi.nlm.nih.gov/pmc/articles/PMC3168328/

49. Wikipedia entry for Natural Language Processing: en.wikipedia.org/wiki/Natural-language_processing
50. www.gutenberg.org/
51. www.english-corpora.org/
52. www.gutenberg.org/files/1342/1342-0.txt

Most commonly implemented by the subword-nmt and SentencePiece libraries, the conceptual drawback is
53. that the representation of a sequence of characters is not unique anymore.

54. This is from a SentencePiece tokenizer trained on a machine translation dataset.

Actually, with out 1d view of color, this is not possible, as sunflower's yellow and brown will average to white... But you get the idea and it does work better in higher dimensions.

56. One example is code.google.com/archive/p/word2vec/

57. This goes under the name of finetuning.

58. github.com/deep-learning-with-pytorch/dlwpt-code/tree/master/p1ch4

59. Or download some from the internet, if a camera isn't available.

60. As recounted by Michael Fowler; galileoandeinstein.physics.virginia.edu/1995/lectures/morekepl.html

Understanding the details of Kepler's laws is not needed for understanding the chapter, but more information can be found at en.wikipedia.org/wiki/Kepler%27s_laws_of_planetary_motion

62. plus.maths.org/content/origins-proof-ii-keplers-proofs

63. Unless you're a theoretical physicist ;)

64. Luca is Italian, so please forgive him for using sensible units. --Eli

65. github.com/deep-learning-with-pytorch/dlwpt-code/blob/master/p1ch5/1_parameter_estimation.ipynb

66. Or maybe it is; we won't judge how you spend your weekend!

67. Bummer! What are we going to do on Saturdays, now?

68. In reality, it will track that something changed params using an inplace operation.

69. www.umontreal.ca/en/artificialintelligence/

There are particular circumstances (involving views as discussed in 3.6) in which `requires_grad` is not set to `False` even when they are created in a `no_grad` context. It is best to use the `detach` function if we need to be sure.

71. psycnet.apa.org/doiLanding?doi=10.1037%2Fh0042519

72. pytorch.org/docs/stable/nn.html#hardtanh but note that the default range is -1 to +1.

Of course, even these statements aren't *always* true:

73. openai.com/blog/nonlinear-computation-in-linear-networks/

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.

75. pytorch.org/docs/stable/nn.html#torch.nn.Module.add_module

76. Aren't rhetorical questions great?

That's probably not going to translate to print well; you'll have to take our word for it, or check it out in the Jupyter notebook.

Not being able to do this kind of operation inside of `nn.Sequential` was an explicit design choice by the PyTorch authors; see the linked comments from @soumith at github.com/pytorch/pytorch/issues/2486

79. I presume; I haven't timed it myself. — Eli

80. colab.research.google.com/

81. github.com/deep-learning-with-pytorch/dlwpt-code

82. commons.wikimedia.org/wiki/File:Image_of_3D_volumetric_QCT_scan.jpg

83. creativecommons.org/licenses/by-sa/3.0/deed.en

84. en.wikipedia.org/wiki/CT_scan#Process

The `series_uid` of this sample is 1.3.6.1.4.1.14519.5.2.1.6279.6001.126264578931778258890371755354, which can be useful if you'd like to look at it in detail later.

86. Retina-UNet: arxiv.org/pdf/1811.08661.pdf

F i s h N e t :

87. papers.nips.cc/paper/7356-fishnet-a-versatile-backbone-for-image-region-and-pixel-level-prediction.pdf

88. www.mayoclinic.org/diseases-conditions/lung-cancer/expert-answers/lung-nodules/faq-2005844

89. Not if we want decent results, at least

90. www.cancer.gov/publications/dictionaries/cancer-terms/def/nodule

91. luna16.grand-challenge.org/Description/

92. luna16.grand-challenge.org/download/

93. Ubuntu provides this via the `p7zip-full` package.

94. www.7-zip.org/

To the rare researcher who has all of their data well-prepared for them in advance: Lucky you! The rest of us
95. will be over here writing parsing and loading code.

96. The name comes from the DICOM nomenclature.

97. www.dicomstandard.org/

98. itk.org/Wiki/MetaIO/Documentation#Quick_Start

99. docs.python.org/3.6/library/uuid.html

100. en.wikipedia.org/wiki/Hounsfield_scale

101. There are exceptions, but they're not relevant right now.

102. Have you found the single misspelled word in this book yet?

103. github.com/deep-learning-with-pytorch/dlwpt-code

104. To something simpler, actually, but the point is we've got options here.

105. The larger `nodule_t` output isn't particularly readable, so we elide most of it in the listing.

106. pypi.python.org/pypi/diskcache/

107. matplotlib.org/

108. Any shell, really, but if you're using a non-Bash shell, you already knew that.

109. docs.python.org/3/library/argparse.html

The only way wrapping the model isn't fully transparent is that custom attributes on your model won't be mirrored on the `DataParallel` instance. We'll revisit this in chapter 13.

111. pytorch.org/tutorials/intermediate/ddp_tutorial.html

112. pytorch.org/docs/stable/optim.html#torch.optim.SGD

113. pytorch.org/docs/stable/optim.html#algorithms

114. pytorch.org/docs/stable/nn.html#conv3d

115. Number of samples, Channels per sample, Depth, Height, Width

116. Which is why there's an exercise to experiment with both in the next chapter!

There are numerical stability benefits for doing so; propagating gradients accurately through an exponential function calculated using 32-bit floating point numbers can be problematic.

118. github.com/pytorch/pytorch/issues/18182

119. The seminal paper on the topic is proceedings.mlr.press/v9/glorot10a/glorot10a.pdf.

120. pytorch.org/docs/stable/nn.html#torch.nn.CrossEntropyLoss

121. docs.scipy.org/doc/numpy/user/basics.indexing.html#boolean-or-mask-index-arrays

122. livebook.manning.com/book/deep-learning-with-pytorch/chapter-11/

If getting dinner in France doesn't involve an airport, feel free to substitute "Paris, Texas" to make the joke work.

124. [https://en.wikipedia.org/wiki/Paris_\(disambiguation\)](https://en.wikipedia.org/wiki/Paris_(disambiguation))

125. github.com/deep-learning-with-pytorch/dlwpt-code/blob/master/util/util.py#L163

126. pypi.org/project/tensorflow/

If you're running training on a different computer from your browser, you'll need to replace `localhost` with the appropriate hostname or IP address.

128. github.com/pytorch/pytorch/blob/v1.2.0/torch/utils/tensorboard/writer.py#L267

129. No one actually says this.

130. en.wikipedia.org/wiki/F1_score

131. en.wikipedia.org/wiki/Arithmetic_mean

132. en.wikipedia.org/wiki/Harmonic_mean

133. en.wikipedia.org/wiki/Geometric_mean

Keep in mind that these images are just a representation of the classification space, and do not represent
134. ground truth.

135. It's not clear if this is actually true, but it's plausible, and the loss *was* getting better...

136. 495958 to be exact.

137. www.cs.toronto.edu/~kriz/cifar.html

138. pytorch.org/docs/0.3.1/data.html#torch.utils.data.sampler.WeightedRandomSampler

And remember that this is after only 100k training samples presented, not the 500k+ of the unbalanced
139. dataset.

140. en.wikipedia.org/wiki/Transformation_matrix#Affine_transformations

141. pytorch.org/docs/stable/nn.html#affine-grid

142. pytorch.org/docs/stable/nn.html#torch.nn.functional.grid_sample

143. Yep, that's a hint it's not the F1 score!

144. arxiv.org/pdf/1505.04597.pdf

145. github.com/jvanvugt/pytorch-unet

146. Copyright (c) 2018 Joris

147. See `util/unet.py`, line 123.

148. en.wikipedia.org/wiki/S%C3%B8rensen%E2%80%93Dice_coefficient

149. pytorch.org/tutorials/beginner/saving_loading_models.html