Part 1

Core PyTorch

Welcome to the first part of this book. This is where we'll take our first steps with PyTorch, gaining the fundamental skills needed to understand its anatomy and work out the mechanics of a PyTorch project.

In chapter 1, we'll make our first contact with PyTorch, understand what it is and what problems it solves, and how it relates to other deep learning frameworks. Chapter 2 will take us on a tour, giving us a chance to play with models that have been pretrained on fun tasks. Chapter 3 gets a bit more serious and teaches the basic data structure used in PyTorch programs: the tensor. Chapter 4 will take us on another tour, this time across ways to represent data from different domains as PyTorch tensors. Chapter 5 unveils how a program can learn from examples and how PyTorch supports this process. Chapter 6 provides the fundamentals of what a neural network is and how to build a neural network with PyTorch. Chapter 7 tackles a simple image classification problem with a neural network architecture. Finally, chapter 8 shows how the same problem can be cracked in a much smarter way using a convolutional neural network.

By the end of part 1, we'll have what it takes to tackle a real-world problem with PyTorch in part 2.

1 Introducing deep learning and the PyTorch Library

This chapter covers

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

The poorly defined term *artificial intelligence* covers a set of disciplines that have been subjected to a tremendous amount of research, scrutiny, confusion, fantastical hype, and sci-fi fearmongering. Reality is, of course, far more sanguine. It would be disingenuous to assert that today's machines are learning to "think" in any human sense of the word. Rather, we've discovered a general class of algorithms

that are able to approximate complicated, nonlinear processes very, very effectively, which we can use to automate tasks that were previously limited to humans.

For example, at https://talktotransformer.com, a language model called GPT-2 can generate coherent paragraphs of text one word at a time. When we fed it this very paragraph, it produced the following:

> *Next we're going to feed in a list of phrases from a corpus of email addresses, and see if the program can parse the lists as sentences. Again, this is much more complicated and far more complex than the search at the beginning of this post, but hopefully helps you understand the basics of constructing sentence structures in various programming languages.*

That's remarkably coherent for a machine, even if there isn't a well-defined thesis behind the rambling.

Even more impressively, the ability to perform these formerly human-only tasks is acquired *through examples*, rather than encoded by a human as a set of handcrafted rules. In a way, we're learning that intelligence is a notion we often conflate with self-awareness, and self-awareness is definitely not required to successfully carry out these kinds of tasks. In the end, the question of computer intelligence might not even be important. Edsger W. Dijkstra found that the question of whether machines could think was "about as relevant as the question of whether Submarines Can Swim."[1]

That general class of algorithms we're talking about falls under the AI subcategory of *deep learning*, which deals with training mathematical entities named *deep neural networks* by presenting instructive examples. Deep learning uses large amounts of data to approximate complex functions 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 with functionality that was, until very recently, exclusively the domain of human beings.

1.1 The deep learning revolution

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

[1] Edsger W. Dijkstra, "The Threats to Computing Science," http://mng.bz/nPJ5.

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

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

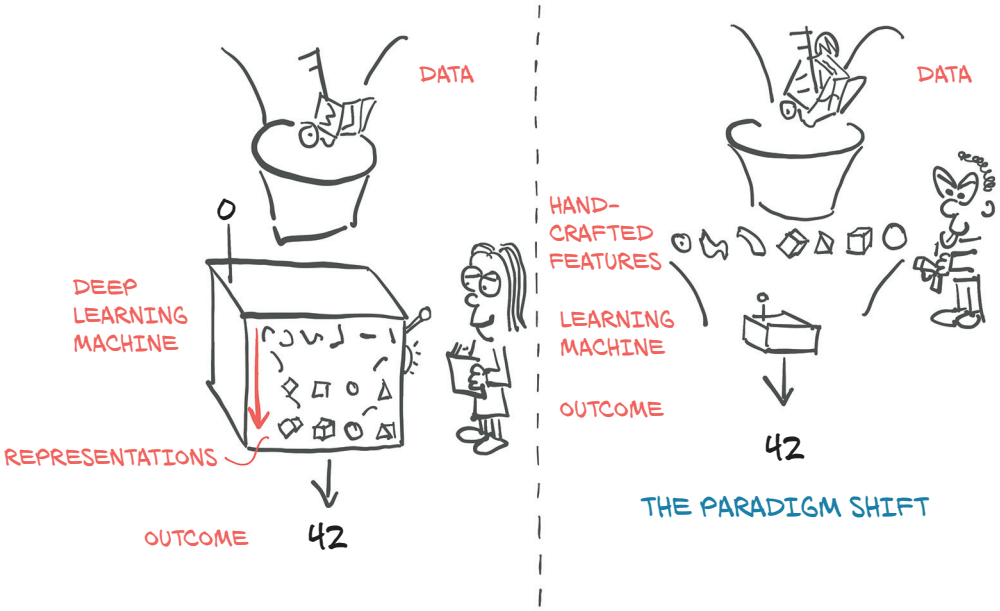

Figure 1.1 Deep learning exchanges the need to handcraft features for an increase in data and computational requirements.

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

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

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

1.2 PyTorch for deep learning

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

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

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

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

take a data source and build out a deep learning project with it, supported by the excellent official documentation.

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

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

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

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

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

1.3 Why PyTorch?

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

[2] We also recommend www.arxiv-sanity.com to help organize research papers of interest.

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

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

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

A design driver for PyTorch is expressivity, allowing a developer to implement complicated models without undue complexity being imposed by the 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 at international conferences.[3]

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

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

1.3.1 The deep learning competitive landscape

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

[3] At the International Conference on Learning Representations (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.

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

In the roughly two years that followed, the landscape changed drastically. The community largely consolidated behind either PyTorch or TensorFlow, with the adoption of other libraries dwindling, except for those filling specific niches. In a nutshell:

- Theano, one of the first deep learning frameworks, has ceased active development.
- TensorFlow:
  - Consumed Keras entirely, promoting it to a first-class API
  - Provided an immediate-execution "eager mode" that is somewhat similar to how PyTorch approaches computation
  - Released TF 2.0 with eager mode by default
- JAX, a library by Google that was developed independently from TensorFlow, has started gaining traction as a NumPy equivalent with GPU, autograd and JIT capabilities.
- 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
  - Replaced CNTK and Chainer as the framework of choice by their respective corporate sponsors

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 communities, thanks to its ease of use, and has picked up momentum since, as researchers and graduates train students and move to industry. It has also built up steam in terms of production solutions. Interestingly, with the advent of TorchScript and eager mode, both PyTorch and TensorFlow have seen their feature sets start to converge with the other's, though the presentation of these features and the overall experience is still quite different between the two.

1.4 An overview of how PyTorch supports deep learning projects

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

First, PyTorch has the "Py" as in Python, but there's a lot of non-Python code in it. Actually, for performance reasons, most of PyTorch is written in C++ and CUDA (www.geforce.com/hardware/technology/cuda), a C++-like language from NVIDIA that can be compiled to run with massive parallelism on GPUs. There are ways to run PyTorch directly from C++, and we'll look into those in chapter 15. One of the 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.

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.

As we already touched on, at its core, PyTorch is a library that provides *multidimensional arrays*, or *tensors* in PyTorch parlance (we'll go into details on those in chapter 3), and an extensive library of operations on them, provided by the `torch` module. Both tensors and the operations on them can be used on the CPU or the GPU. Moving computations from the CPU to the GPU in PyTorch doesn't require more than an additional function call or two. The second core thing that PyTorch provides is the ability of tensors to keep track of the operations performed on them and to analytically compute derivatives of an output of a computation with respect to any of its inputs. This is used for numerical optimization, and it is provided natively by tensors by virtue of dispatching through PyTorch's *autograd* engine under the hood.

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

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

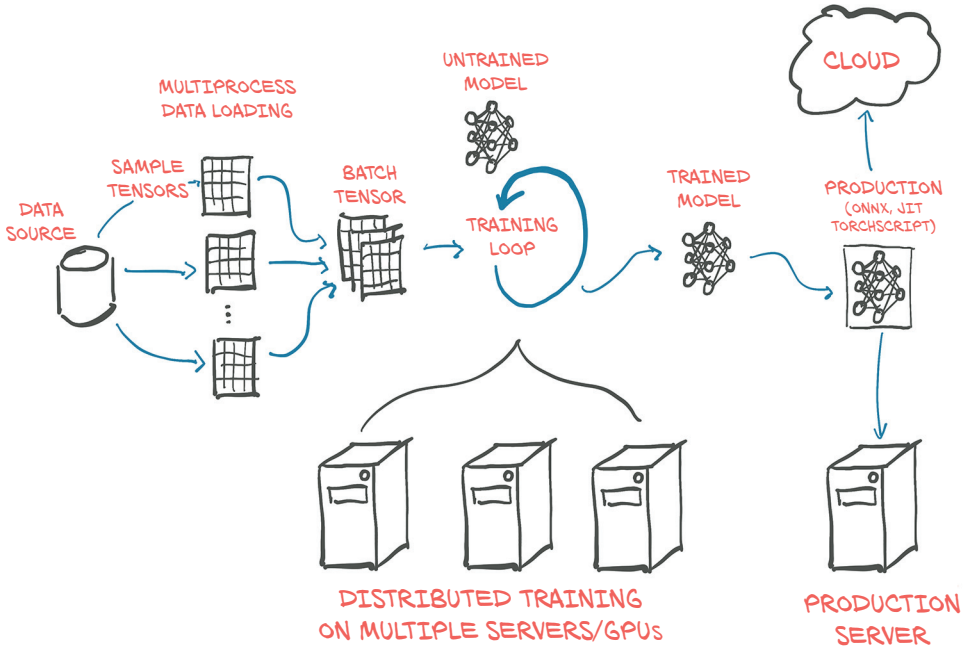

Figure 1.2 Basic, high-level structure of a PyTorch project, with data loading, training, and deployment to production

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

As data storage is often slow, in particular due to access latency, we want to parallelize data loading. But as the many things Python is well loved for do not include easy, efficient, parallel processing, we will need multiple processes to load our data, in order to assemble them into *batches*: tensors that encompass several samples. This is rather elaborate; but as it is also relatively generic, PyTorch readily provides all that magic in the `DataLoader` class. Its instances can spawn child processes to load data from a dataset in the background so that it's ready and waiting for the training loop as soon as the loop can use it. We will meet and use `Dataset` and `DataLoader` in chapter 7.

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

With the mechanism for getting batches of samples in place, we can turn to the training loop itself at the center of figure 1.2. Typically, the training loop is implemented as a standard Python `for` loop. In the simplest case, the model runs the required calculations on the local CPU or a single GPU, and once the training loop has the data, computation can start immediately. Chances are this will be your basic setup, too, and it's the one we'll assume in this book.

At each step in the training loop, we evaluate our model on the samples we got from the data loader. We then compare the outputs of our model to the desired output (the targets) using some *criterion* or *loss function*. Just as it offers the components from which to build our model, PyTorch also has a variety of loss functions at our disposal. They, too, are provided in `torch.nn`. After we have compared our actual outputs to the ideal with the loss functions, we need to push the model a little to move its outputs to better resemble the target. As mentioned earlier, this is where the PyTorch autograd engine comes in; but we also need an *optimizer* doing the updates, and that is what PyTorch offers us in `torch.optim`. We will start looking at training loops with loss functions and optimizers in chapter 5 and then hone our skills in chapters 6 through 8 before embarking on our big project in part 2.

It's increasingly common to use more elaborate hardware like multiple GPUs or multiple machines that contribute their resources to training a large model, as seen in the bottom center of figure 1.2. In those cases, `torch.nn.parallel.DistributedDataParallel` and the `torch.distributed` submodule can be employed to use the additional hardware.

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

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

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

the basis of the production deployment capabilities of PyTorch. We'll cover this in chapter 15.

1.5 Hardware and software requirements

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

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

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

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

The last consideration is the operating system (OS). PyTorch has supported Linux and macOS from its first release, and it gained Windows support in 2018. Since current Apple laptops do not include GPUs that support CUDA, the precompiled macOS packages for PyTorch are CPU-only. Throughout the book, we will try to avoid assuming you are running a particular OS, although some of the scripts in part 2 are shown as if running from a Bash prompt under Linux. Those scripts' command lines should convert to a Windows-compatible form readily. For convenience, code will be listed as if running from a Jupyter Notebook when possible.

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

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

1.5.1 Using Jupyter Notebooks

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

At this point, you need to start the notebook server from the root directory of the code checkout from GitHub. How exactly starting the server looks depends on the details of your OS and how and where you installed Jupyter. If you have questions, feel free to ask on the book's forum.[5] Once started, your default browser will pop up, showing a list of local notebook files.

[5] https://forums.manning.com/forums/deep-learning-with-pytorch

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

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

1.6 Exercises

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

1.7 Summary

- Deep learning 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 allows us to precompile models and invoke them not only from Python but also from C++ programs and on mobile devices.
- 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.