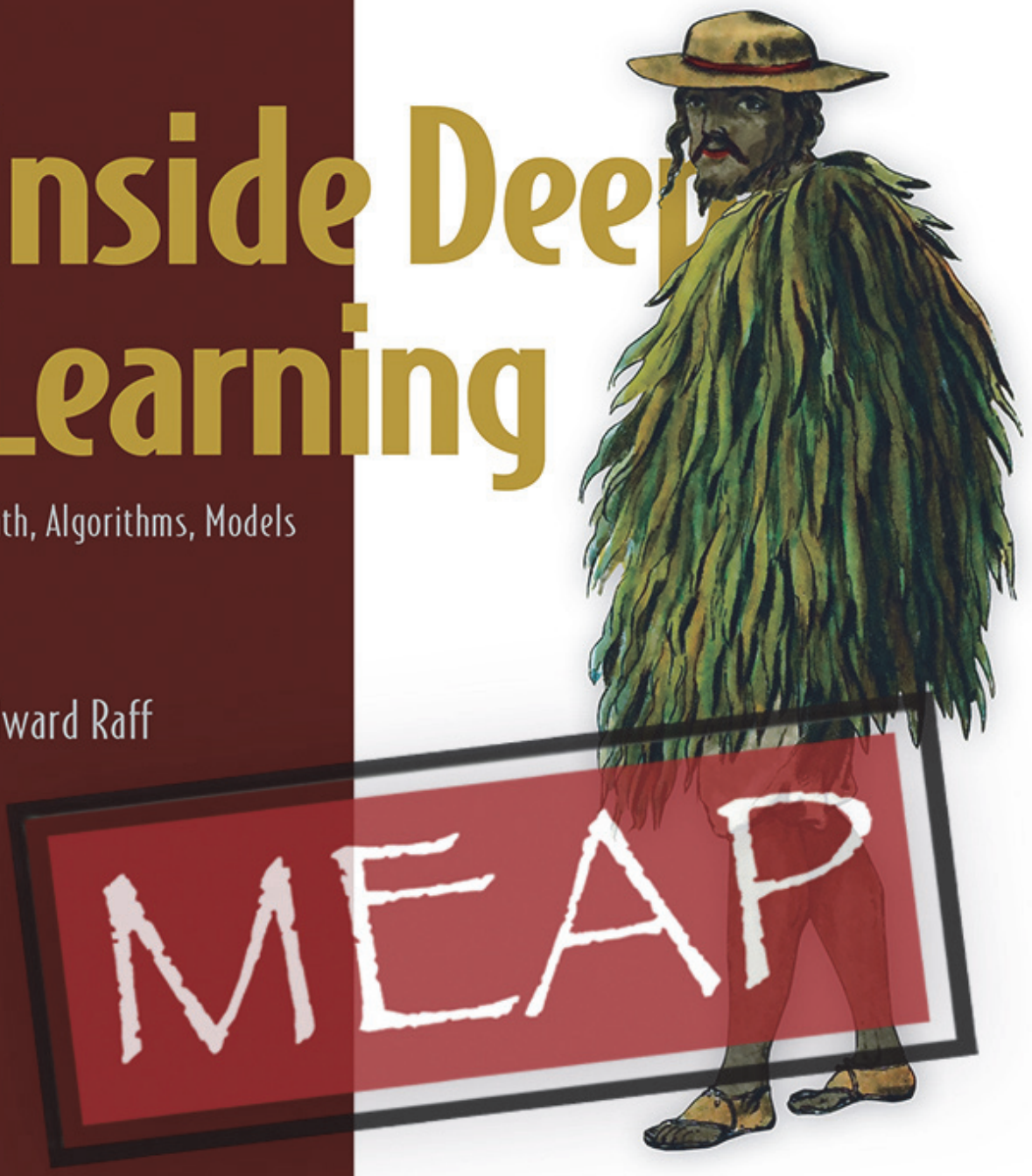
Using PyTorch and Google Colab

Inside Deep Learning

Math, Algorithms, Models

Edward Raff



MANNING

MEAP Edition
Manning Early Access Program
Inside Deep Learning
Math, Algorithms, Models
Version 1

For more information on this and other Manning titles go to
manning.com

welcome

I'm delighted that you've purchased the MEAP for my book *Inside Deep Learning: Math, Algorithms, Models*. This book is geared toward those with a firm programing background who have some machine learning (ML) experience but want to go deeper. You are in good shape to reach through this book if you are comfortable with the basics of calculus, linear algebra, and statistics that go into machine learning.

If you can record the inputs and correct outputs for a task, deep learning can help you translate that task from human time and effort to an automated process. The input could be an image, and the output "cat", or "dog" for example, would describe the images' content. The input could be an English sentence, and the output a French sentence with the same meaning. That's machine translation. The input could be the description "cat" and the output be an actual image of a cat! This input/output nature makes deep learning widely applicable to almost any domain you can think of. It's why I got into machine learning in the first place, giving me a chance to have an impact and contribute to solving real problems in almost any domain.

Deep learning (also called neural networks) gives us tools to help improve the quality of life in big and small ways, and my sincere desire is to pass that along to you. By the end of this book, you should understand:

- What deep learning is
- How to build and modify deep learning models
- Which "building blocks" you should look toward for a given problem

To achieve this there are two overall parts to the book. In each chapter we will not only show how to implement the code for these techniques but we'll show and explain the math behind them too.

Part 1 will cover the basics. The framework we will use to implement our deep learning models (PyTorch), and the three oldest and most common types of neural networks (they come from the 1980s and earlier!). These are the foundations from which all the other chapters will build. Once we have these foundations underfoot, we will learn the approaches invented in the past ten years that helped revive neural networks. Combined, these give us a set of "building blocks" that we can put together.

Part 2 will start building new kinds of approaches that move past the standard classification and regression problems from normal machine learning. We'll do unsupervised learning, object detection, generative modeling, and transfer learning as some major concepts.

After almost a decade as a consultant I've had the opportunity to work on or assist with a large number of topics in a variety of domains. The topics of these chapters have been chosen based on what I've found useful to solving problems over my career thus far. It's all the things

I wish I knew years ago. My students thus far have found it useful to them academically and professionally, and my goal is to enable you for a successful career in this field.

During the MEAP I'll be diligently working on the next chapters, and eagerly looking forward to your feedback in the liveBook Discussion Forum. If you find the book has taught you something new, I will be very happy and if I've failed to explain something well, I want to improve it for you as best I can.

—Edward Raff

brief contents

PART 1: FOUNDATIONAL METHODS

1 *The Mechanics of Learning*

2 *Fully Connected Networks*

3 *Convolutional Neural Networks*

4 *Recurrent Neural Networks*

5 *Modern Training Techniques*

6 *Common Design Building Blocks*

PART 2: BUILDING ADVANCED NETWORKS

7 *Embedding & Auto Encoding*

8 *Object Detection*

9 *Generative Adversarial Networks*

10 *Attention Mechanism*

11 *Alternatives to RNNs*

12 *Transfer Learning*

13 *Advanced Design Building Blocks & Training Techniques*

1

The Mechanics of Learning

This chapter covers:

- Introducing PyTorch, a tensor-based API for deep learning
- Running faster code with PyTorch's GPU acceleration
- Understanding automatic differentiation as the basis of learning
- Using the Dataset interface to prepare our data.

Deep Learning, also called neural networks, or artificial neural networks, has lead to dramatic advances in the quality, accuracy, and usability machine learning. Technology that was considered impossible just 10 years ago is now widely deployed or considered technically possible, even if work remains. Digital assistants like Cortana, Google, Alexa and Siri are ubiquitous and can react to natural spoken language. Self driving cars have been racking up millions of miles on the road as they are refined for eventual deployment. We can finally catalog and calculate just *how much* of the Internet is made of cat photos. Deep Learning has been instrumental to the success of all of these use cases, and many more.

In this book I will give you a wide breadth of exposure to some of the most common and useful techniques in deep learning today. A major focus will be not just how to use and code these networks, but to understand how and why they work at a deep level. With a deeper understanding you'll be better equipped to select the best approach for your own problems. Beyond being able to wield these tools effectively, you'll also have the understanding to keep up with advances in this rapidly progressing field. To make best use of this book, you should be familiar with programming in python, and have some passing memory of a calculus, statistics, and linear algebra course. You should also have some prior experience with ML, and it is OK if you aren't an expert. Topics from ML broadly will be quickly re-introduced, but our goal is to move into new and interesting details about deep learning.

Let's get a clearer idea of what Deep Learning is, and how this book will teach it. Deep Learning is itself a sub-domain of Machine Learning, which is also a sub-domain of Artificial Intelligence. Broadly 1, we could describe AI as getting computers to make decisions that *look* smart. I say *look* because it is very hard to define what "smart" or "intelligence" truly is. These are deep

philosophical questions that we are not going to tackle. But the point is that AI should be making decisions that we think are reasonable and what a "smart" person would do. Your GPS telling you how to get home actually uses some old[2] AI techniques to work, and taking the fastest route home is a "smart" decision. Getting computers to play many video games has been done with purely AI based approaches, where only the rules of the game are encoded - and one does not *need* to actually show the AI how to play a single game of Chess. Figure 1.1 shows AI as the outer most layer of these fields.

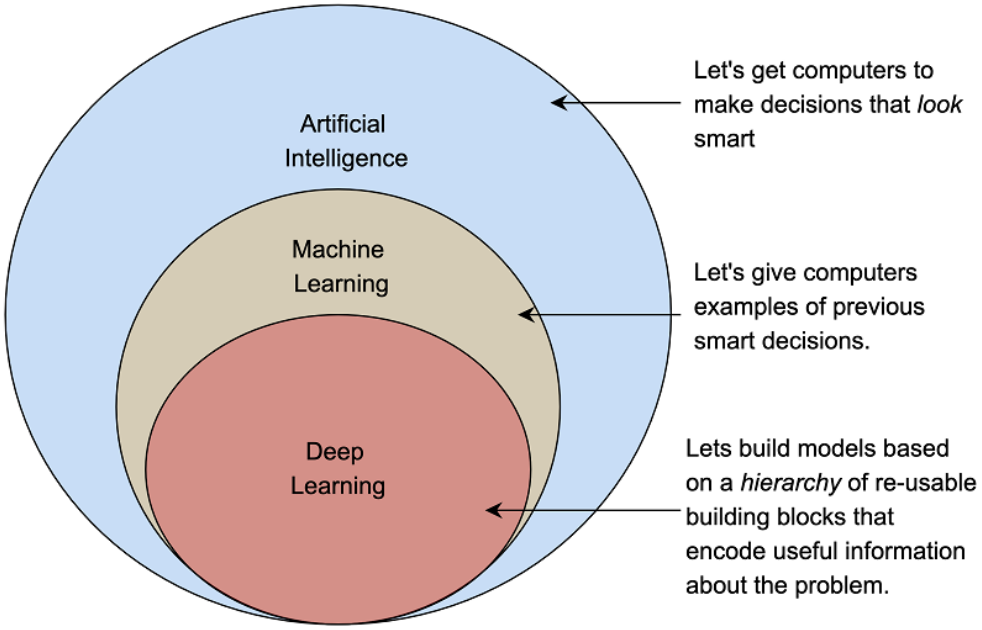


Figure 1.1: A (simplified) hierarchy of AI, ML, and Deep Learning. Each layer has a short description of how we might characterized the added item that makes it different from the preceding layer. So a deep learning approach will generally have the properties of ML and AI.

Machine Learning (ML) is when we start to give our AI *examples* of previous smart, and not so smart, decisions. Telling it explicitly "this is what should have happened". For example, we can improve our chess playing AI by giving it example games from chess grand masters (which has a winner and looser, so both a smart and not-as-smart set of decisions) to learn from as well.

Deep Learning in turn is not one "algorithm" in particular, but a whole slew of *hundreds* of small "algorithms" that act like building blocks. Part of being a good practitioner is known the building blocks you have available, and which ones to stick them together to create one larger model for *your* problem. Each building block is designed to work well for certain kinds of problems, giving the model valuable information. Figure 1.2 shows how we might compose different blocks together to tackle three different situations. One of the goals in this book is to cover a wide breadth of the different kinds of building blocks, so that you know and understand how they can be used for different kinds of problems. Some of the blocks a very generic ("Data is a Sequence" could be used for literally *any* kind of sequence) while others are more specific ("Data is an Image" applies to only images), which will impact when and how you want to use them.

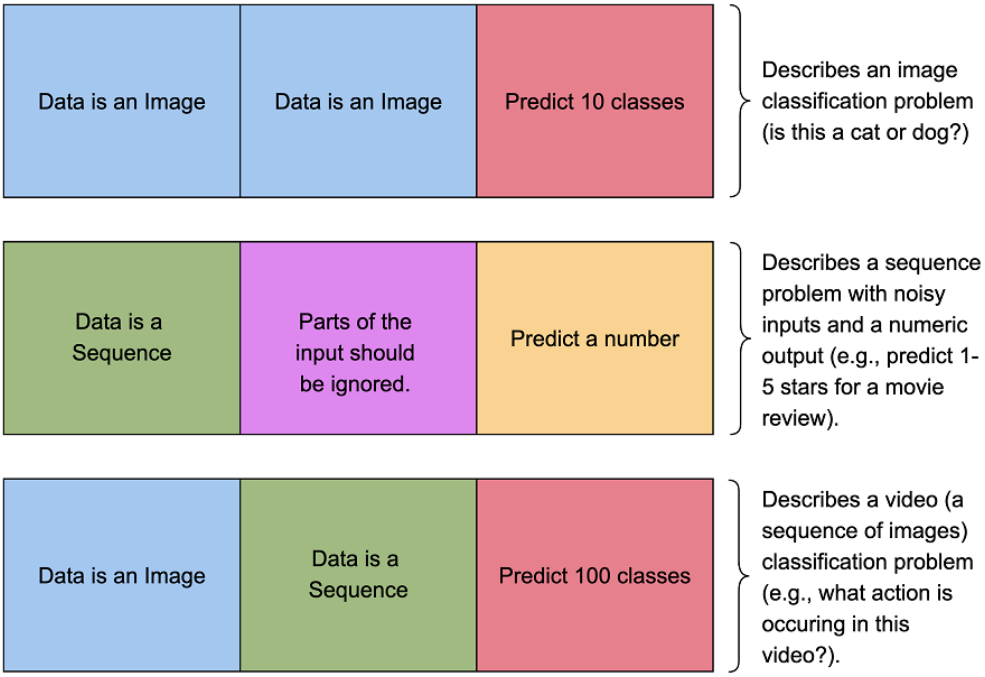


Figure 1.2: One of the defining characteristics of deep learning is the use of re-usable "blocks" that we can build models from. Different blocks are useful for different kinds of data, and can be mixed-and-matched to deal with different problems. Here we can see how five different blocks can be re-used to describe models for different types of problems.

The first row shows two "image" blocks, in order to create a *deep* model. Applying blocks repeatedly is where the "deep" in "deep learning" comes from. Adding depth makes a model capable of solving more complex problems. This depth is often obtained by repeatedly stacking the same kind of block multiple times. The second row in the figure shows a case for a sequence problem. For example, text can be represented as a sequence of words. But not *all* words are meaningful, and so we may want to give the model a block that helps it learn to *ignore* certain words. The third row shows how we can describe new problems using the blocks we know about. If we want our AI to watch a video and predict what is happening (e.g., "running", "tennis", or "adorable puppy attack") we can use the "image" and "sequence" block to create a "sequence of images", which is a verbose way of saying "a video".

These different building blocks define our model, but like all of machine learning we also need data and some mechanism for learning. When we say "learning", we are not talking about the way that humans learn. In machine (and deep) learning, the "learning" is the mechanical process of getting the model to make the "smart looking" predictions on our data. This happens via a process called "optimization" or "function minimization". At the start, before we have seen any data, our model is simply going to return random outputs because all of the parameters that control the model are initialized to random values. By *optimizing* the blocks over the data, we make our models learn. This gives us the larger picture in Figure 1.3.

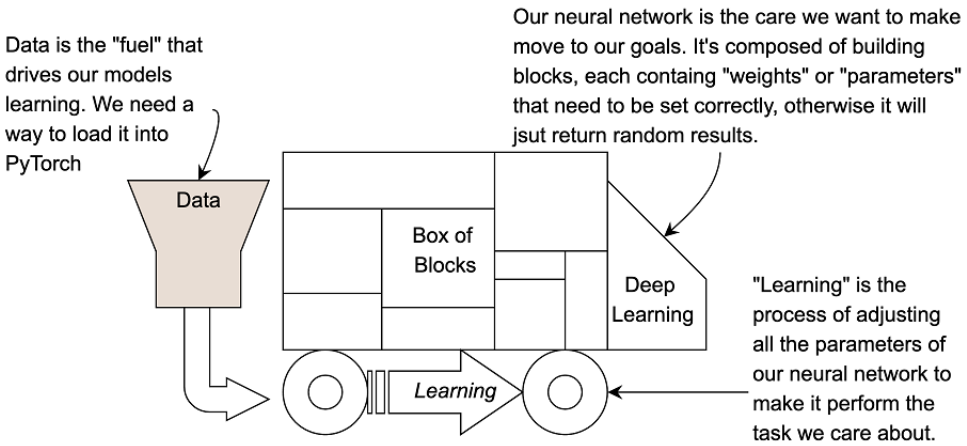


Figure 1.3: The `car` of deep learning. The car is built from many different building blocks that we will learn about, and we use different building locks to build cars for different tasks. But we need fuel and wheels to make the car go. Our wheels are the task of *learning* which is done via a process called *optimization*, and the fuel is our data.

In most chapters of this book you will learn about new building blocks that you can use to build deep learning models for different applications. You can think of each block as a kind of (potentially very simple) algorithm. We will learn about the uses of each block, some explanation about how or why they work, and how to combine them in code to create a new model. Thanks to the nature of building blocks on top of each other, we will be able to ramp up from simple tasks (e.g., simple prediction problems you could have tackled with your favorite non-deep ML algorithm) to complex examples like machine translation (e.g., having a computer translate from English to French). As we do this we will start with the basic approaches and methods that have been in use since the *1960s* for training and building neural networks, but using a modern framework [3]. As we progress through this book well build upon what we've learned in previous chapters, introducing new blocks of using what we've learned to extend old blocks or build new blocks from existing ones.

That said, this book is *not* a cook book of code snippets to just throw at any new problem. Rather, the goal is to get you comfortable with the language that deep learning researchers use to describe new and improved blocks they are creating, so that you can have some familiarity and recognize when a new block may be useful to you. Math can often be used to express complex changes very succinctly, so I will be sharing the math behind the building blocks.

We won't *do* a lot of math, but we are going to *show* the math. When I say we won't *do* math that does not mean the book has none, but it means we are not going to *derive* or *prove* the math that will be shown. Instead I'm going to present you with the final equations, explain what they mean and do, and try to attach useful intuition to the different equations we will see through the book. I'm calling it intuition because we will go through the bare minimum math needed to explain the high level idea of what is happening, and showing exactly "why" a result is the way it is requires more math than I'm asking you to have. As we show the math, we will interweave it with PyTorch code whenever possible. This way you can start to build a mental map between equations and deep learning code.

For for this chapter, we are first going to get introduced to our compute environment Google Colab. Next, we can start talking about PyTorch and tensors, which is how we represent information inside of PyTorch. After that well dive into the use of Graphics Processing Units (GPUs) that make PyTorch fast, and *automatic differentiation* which is the "mechanics" PyTorch leverages to make our neural network models learn. Finally, we will quickly implement a dataset object which is what PyTorch will need to feed our data into the model for the learning process. This gives us the fuel and wheels to get our deep learning car *moving* starting in chapter 2. From there on we can focus on *just* the deep learning.

1.1 Getting Started with Colab

We will be using Graphics Processing Units (GPUs) for *everything* we do with deep learning. It is unfortunately a computationally demanding practice, and GPUs are essentially a *requirement* for getting started, and especially when you start to work on larger applications. I use deep learning all the time as part of my day-job, and regularly kick off jobs that take a few *days* to train on multiple GPUs. Some of my research experiments take as much as *a month* of compute for each run.

Unfortunately, GPUs are also decently expensive. The best option currently for most people who want to get started with deep learning is to spend\$600-\$1200 on one of the higher end Nvidia GTX or Titan GPUs. That is *if* you have a computer that can be expanded/upgraded with a high end GPU. If not, you are probably looking at at least \$1500-\$2500 to build a nice workstation to put those GPUs in. Thats a pretty steep cost just to *learn* about deep learning.

Google's Colab (https://colab.research.google.com) provides a GPU for free, for a limited amount of time. I've designed every example in this book to run within Colab's time limits. Appendix A contains the instructions for setting up Colab. Once you have it setup, the common data science and machine learning tools like `seaborn`, `matplotlib`, `tqdm`, and `pandas` are all built in and ready to go. Colab operates as a familiar Jupyter notebook, where you run code in "cells" which produce an output directly below. This book is in fact a Jupyter notebook, so you can run the code blocks (like the next one) to get the same results. I'll let you know if a code cell isn't meant to be run.

```
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
from tqdm.autonotebook import tqdm
import pandas as pd
```

As we progress through this book, we are not going to repeatedly show all of the imports, as that is mostly a waste of paper. Instead they will be available on-line as part of the downloadable copies of the code, which can be found at https://github.com/EdwardRaff/Inside-Deep-Learning.

1.2 The World as Tensors

Deep learning has been used on spreadsheets, audio, images, and text, but deep learning frameworks don't use different classes or objects to distinguish between these many different kinds of data. Instead they have one data type that they work with, and we the users must convert our data into this format. For PyTorch this singular "view" of the world is through a *tensor* object. Tensors are used to represent both data, the inputs/outputs to any deep learning block, and the parameters that control the behavior of our networks. Two important features are built into these tensor objects, the ability to do fast parallel computation with Graphics Processing

Units (GPUs) and the ability to do some calculus (derivatives) for us automatically. With your prior ML experience in Python you should have prior experience with Numpy which also uses the tensor concept. In this section we are going to quickly review the tensor concept, and note the ways that tensors in PyTorch differ from Numpy that form the foundation for our deep learning building blocks.

We will begin by importing the torch library, and discussing *tensors*, which are also called "n-dimensional arrays". Both Numpy and PyTorch allow us to create n-dimensional arrays. A 0-dimensional array is called a *scalar*, and is any single number (e.g., "3.4123"). A 1-dimensional array is a *vector* (e.g., [1.9, 2.6, 3.1, 4.0, 5.5]), and a 2-dimensional array is a *matrix*. Scalars, vectors, and matrices are all tensors. In fact, any value of n is still a tensor. The word tensor is simply referring to the overall concept of an n-dimensional array.

We care about tensors because they are a convenient way for us to organize much of our data, and our algorithms. This is the first foundation that PyTorch provides us, and we will often convert Numpy tensors to PyTorch tensors. Figure 1.4 shows four tensors, their matching the shapes, and the mathematical way to express the shape. Extending the pattern a 4-dimensional tensor could be written as (B, C, W, H) or as $\mathbb{R}^{B, C, W, H}$.

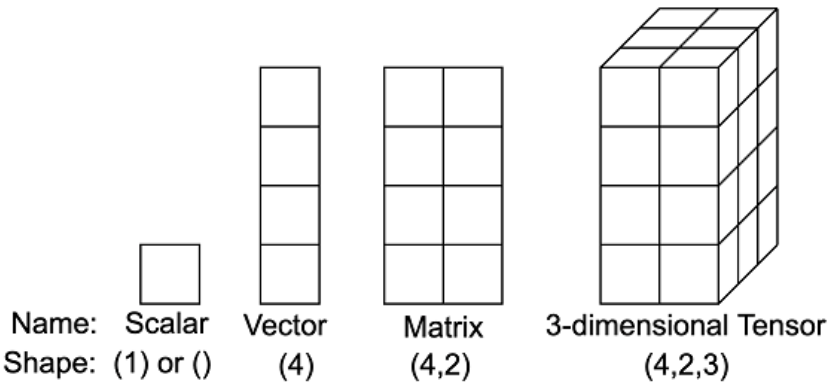


Figure 1.4: Examples of different kinds of tensors you might see, with more dimensions or "axes" as we move from left to right. A scalar represents a single value. A vector a list of values and is how we often think about one datapoint. A matrix is a grid of values and will often be used for a dataset. A 3-dimensional tensor can be used to represent a dataset of sequences.

We will use some common notation to associate math symbols with tensors of a specific shape. We will use a capitol letter like X or Q to represent a tensor with two or more dimensions. If we are talking about a vector, we will use a lower case bold letter like $\boldsymbol{x}$ or $\boldsymbol{h}$. Last, we will use a lower case non-bold letter like x or h for scalars.

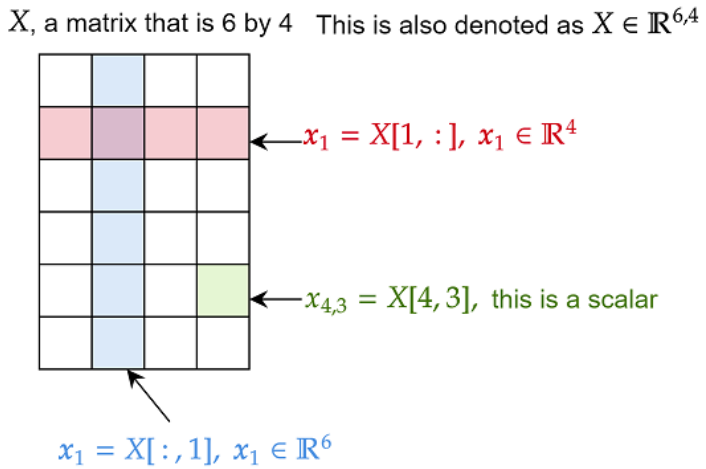


Figure 1.5: A tensor can be `sliced` to grab sub-tensors from a larger one. For example in red we can grab a row-vector from the larger matrix, and in blue we grab a column-vector from the matrix. Depending on what our tensor represents, this can allow us to manipulate different parts of the data.

In talking about and implementing neural networks, we often want to refer to a row within a larger matrix, or a scalar within a larger vector. This is shown in Figure 1.5 and is often called "slicing." So if we have a matrix X, we could use $\boldsymbol{x}_i$ to reference the i'th row of X. In code, that would be `x_i = X[i,:]`. If we wanted the i'th row and j'th column, that would become $x_{i,j}$, which is not bold because it is reference a single value - making it a scalar. Again, the code version of that would be `x_ij = X[i,j]`.

To use PyTorch we will need to import it as the `torch` package. With it, we can immediately start creating some tensors. Every time you nest a list within another list, you are creating a new dimension to the tensor that PyTorch will produce.

```
import torch
```

```
torch_scalar = torch.tensor(3.14)
torch_vector = torch.tensor([1,2,3,4])
torch_matrix = torch.tensor([[1,2,],
                             [3,4,],
                             [5,6,],
                             [7,8,]])
#You don't have to format it like I did, thats just for clarity
torch_tensor3d = torch.tensor([
                              [
                              [1, 2, 3],
                              [4, 5, 6],
                              ],
                              [
                              [7, 8, 9],
                              [10,11,12],
                              ],
                              [
                              [13,14,15],
                              [16,17,18],
                              ],
                              [
```

```
                                [19,20,21],
                                [22,23,24],
                                ]
                                ])
```

And if we print the shapes of these tensors out, you should see that you get the same shapes listed above. Again, while scalars, vectors, and matrices are different things, they are all unified under the larger umbrella of "tensors". We care about this because we will use tensors of difference shapes to represent different types of data. We will get to those details later, for now are are going to simply focus on the mechanics PyTorch provides to work with tensors.

```
print(torch_scalar.shape)
print(torch_vector.shape)
print(torch_matrix.shape)
print(torch_tensor3d.shape)

torch.Size([])
torch.Size([4])
torch.Size([4,2])
torch.Size([4,2,3])
```

If you have done any machine learning or scientific computing in python, you have probably used the NumPy library. As you would expect, PyTorch supports converting these NumPy objects into their PyTorch counterparts. Since both of them represent data as tensors, this is a very painless process. The next two code blocks show how we can create a random matrix in NumPy, and then convert it into a PyTorch `Tensor` object.

```
x_np = np.random.random((4,4))
print(x_np)


[[0.75801813 0.35488001 0.98752776 0.24777024]
 [0.59312147 0.54179572 0.107195   0.40178697]
 [0.02499667 0.16483829 0.13334598 0.14996395]
 [0.02202227 0.23933321 0.79426645 0.88604378]]


x_pt = torch.tensor(x_np)
print(x_pt)


tensor([[0.7580, 0.3549, 0.9875, 0.2478],
        [0.5931, 0.5418, 0.1072, 0.4018],
        [0.0250, 0.1648, 0.1333, 0.1500],
        [0.0220, 0.2393, 0.7943, 0.8860]], dtype=torch.float64)
```

Both NumPy and torch support multiple different data types. By default, NumPy will use 64-bit floats, and PyTorch will default to 32-bit floats. However, if you create a PyTorch tensor from a NumPy one, it will use the same type as the given NumPy tensor. You can see that above where PyTorch suddenly felt the need to let us know that `dtype=torch.float64`, since it is not the default choice.

For deep learning, the most common types we will care about are 32-bit floats, 64-bit integers (Longs), and booleans (i.e., binary "True"/"False"). Most operations will leave the type of a tensor unchanged, unless we explicitly create or cast it to a new version. To avoid issues with types, you can always specify explicitly what type of tensor you want to create when calling a function. The below code shows how we can check what type of data is contained in our tensor using the `dtype` attribute.

```
print(x_np.dtype, x_pt.dtype)
```

```
float64 torch.float64
```

```
#Lets force them to be 32-bit floats
x_np = np.asarray(x_np, dtype=np.float32)
x_pt = torch.tensor(x_np, dtype=torch.float32)
print(x_np.dtype, x_pt.dtype)
```

```
float32 torch.float32
```

The main exception to this is logic operations, which we can use to quickly create *binary masks*. A mask is a tensor used to tell us which portions of another tensor are valid to use or not. This is something that we will eventually use in some of our more complex neural networks. So lets say we wanted to find every value in a tensor that was greater than 0.5. Both PyTorch and NumPy would allow us to use the standard logic operators to check for things like this.

```
b_np = (x_np > 0.5)
print(b_np)
print(b_np.dtype)
```

```
[[ True False  True False]
 [ True  True False False]
 [False False False False]
 [False False  True  True]]
bool
```

```
b_pt = (x_pt > 0.5)
print(b_pt)
print(b_pt.dtype)
```

```
tensor([[ True, False,  True, False],
        [ True,  True, False, False],
        [False, False, False, False],
        [False, False,  True,  True]])
torch.bool
```

While the APIs between NumPy and PyTorch are not identical, they share many functions with the same names, behaviors, and characteristics.

```
    np.sum(x_np)
```

```
[13]: 6.406906
```

```
    torch.sum(x_pt)
```

```
[14]: tensor(6.4069)
```

While many are the same, some are not *quite* identical. There may be slight differences in behavior, or in the arguments required. These discrepancies are usually because the PyTorch version has made some changes that are particular to how these methods will be used for neural network design and execution. Below is an example of the transpose function, where PyTorch requires us to specific which two dimensions are to be transposed. NumPy simply takes the two dimensions and transposes them without complaint.

```
      np.transpose(x_np)
```

```
[15]: array([[0.75801814, 0.59312147, 0.02499667, 0.02202227],
              [0.35488   , 0.54179573, 0.1648383 , 0.23933321],
              [0.9875277 , 0.107195  , 0.13334598, 0.79426646],
              [0.24777023, 0.40178698, 0.14996396, 0.8860438 ]], dtype=float32)
     torch.transpose(x_pt,0,1)
```

```
[16]: tensor([[0.7580, 0.5931, 0.0250, 0.0220],
              [0.3549, 0.5418, 0.1648, 0.2393],
              [0.9875, 0.1072, 0.1333, 0.7943],
              [0.2478, 0.4018, 0.1500, 0.8860]])
```

In this case, PyTorch does this because we often want to transpose different dimensions of the tensor for deep learning applications, where NumPy tries to stay with more general expectations. See below how we can "transpose" two of the dimensions in our `torch_tensor3d` from the start of the chapter. Originally it had a shape of $(4, 2, 3)$. If we transpose the first and third dimensions, we will get a shape of $(3, 2, 4)$.

```
print(torch.transpose(torch_tensor3d,0,2).shape)
```

```
torch.Size([3,2,4])
```

Because such changes exist, you should always double check the PyTorch documentation at https://pytorch.org/docs/stable/index.html if you attempt to use a function you are familiar with, but suddenly find it is not behaving as expected. Its also a good tool to generally have open when using PyTorch. There are a lot of different functions that can help you within PyTorch, and we simply can not review them all.

1.2.1 PyTorch GPU Acceleration

The first important functionality that PyTorch gives us beyond what NumPy can do is use a *Graphics Processing Unit* (GPU) to accelerate mathematical calculations. What are GPUs? They are hardware in your computer that is specifically designed for doing 2D and 3D graphics, mainly to accelerate videos (watching an HD movie) or play video games. So what does that have to do with neural networks? Well, a lot of the math involved in making 2D and 3D graphics fast is tensor based, or at least related. For this reason, GPUs have been getting good at doing a lot of the things we want very quickly. As graphics, and thus GPUs, got better and more powerful, people realized they could also be used for a lot of scientific computing and machine learning.

So at a high level, you can think of GPUs as giant tensor calculators. You should almost always use a GPU when doing anything with neural networks. Its a good pair up since neural networks are very compute intensive, and GPUs are fast at the exact type of computations we need to perform. If you want to do deep learning in a professional context, you should invest in a computer with a powerful Nvidia GPU. But for now, we can get by for *free* using Colab.

The trick to using GPUs effectively is to avoid computing on a *small* amount of data. This is because your computer's CPU must first move data to the GPU, then ask the GPU to perform it's math, wait for the GPU to finish, and then copy the results back off the GPU. The steps surrounding this process are fairly slow, and take longer than it would for the CPU to do the math itself if we are only calculating a few things.

What exactly counts as 'too small'? Well, that depends on your CPU, GPU, and the math you are doing. If you are worried about this problem, you can do some benchmarking to see if the CPU is coming out faster. If so, you are probably working on too little data.

Lets test some of that out right now with a matrix multiplication. This is a pretty basic linear algebra operation that is very common in neural networks. Whats the math for this? We have a matrix $X^{n,\ m}$ and $Y^{m,\ p}$, and we can compute a resulting matrix $C^{n,\ p} = X^{n,\ m} Y^{m,\ p}$. Note that C has as many rows as X and as many columns as Y. When implementing neural networks, we are going to be doing lots of operations that change the *shape* of a tensor, which we can see happens when we multiply two matrices together. This will be a common source of bugs, so you should think about tensor shapes when writing code.

Well use the *timeit* library to help us, it allows us to run some code multiple times and tells us how long it took to run it. First we will make a larger matrix X, and we will compute XX several times, and see how long that takes to run.

```
import timeit
x = torch.rand(2**11,2**11)
time_cpu = timeit.timeit("x@x", globals=globals(), number=100)
```

You should see it takes a bit of time to run that code, but not too long. On my computer it took 3.49 seconds to run, which is stored in the `time_cpu` variable. Now how do we get PyTorch to use our GPU? First we need to create a `device` reference. We can ask PyTorch to give us one using the `torch.device` function. If you have an Nvidia GPU, and the CUDA drivers are installed properly, you should be able to pass in `"cuda"` as a string and get a object back representing that device.

```
print("Is CUDA available? :", torch.cuda.is_available())
device = torch.device("cuda")
```

```
Is CUDA available? : True
```

Now that we have a reference to the GPU (device) that we want to use, we just need to ask PyTorch to move that object to the given device. Luckily that can be done with a simple `to` function, and then we can use the same code as before.

```
x = x.to(device)
time_gpu = timeit.timeit("x@x", globals=globals(), number=100)
```

When I run this code, I get the time to perform 100 multiplications as 0.1318 seconds, which is an instant 25.31 × speedup. Now this was a pretty ideal case, as matrix multiplications are super efficient on GPUs, and we created a pretty big matrix. You should try making the matrix smaller and smaller and see how that impacts the speedup that you get.

Something that we do need to be aware of is that this only works if every object involved is on the same device. Say you run the below code, where the variable `x` has been moved onto the GPU, and y has not (so it is on the CPU by default).

```
x = torch.rand(128,128).to(device)
y = torch.rand(128,128)
x*y
```

You will end up getting an error message that says:

```
 RuntimeError: expected device cuda:0 but got device cpu
```

The error will tell you which device the first variable is on (`"cuda:0"`) but that the second variable was on a different device ('cpu'). If we instead wrote `y*x` you would see the error change too

"expected device cpu but got device cuda:0" instead. Whenever you see an error like this, you have a bug somewhere that kept your from moving everything to the same compute device.

The other thing that we want to be aware of is how to convert our PyTorch data back to the CPU. For example, we may want to convert a tensor back to a NumPy array so that we can pass it to Matplotlib, or save it to disk. The PyTorch `tensor` object as a `.numpy()` method that will do this for you, but if you call `x.numpy()` right now, you will get the error:

```
TypeError: can't convert CUDA tensor to numpy. Use Tensor.cpu()
to copy the tensor to host memory first.
```

Instead, you can use the handy shortcut function `.cpu()` to move an object back to the CPU, where we can interact with it in a normal way. So you will often see code that looks like `x.cpu().numpy()` when we want to access the results of our work.

The `.to` and `.cpu()` methods make it easy to write code that is suddenly GPU accelerated. Once on a GPU or similar compute device, almost *every* method that comes with PyTorch can be used and will net you a nice speedup. But sometimes we will want to store tensors and other PyTorch objects in a list, dictionary, or some other standard python collection. TO help us with that, we are going to define this `moveTo` function, which will recursevly go through the common python and PyTorch contains and move every device found onto the specified device.

```
def moveTo(obj, device):
    """
    obj: the python object to move to a device, or to move its contents to a device
    device: the compute device to move objects to
    """
    if isinstance(obj, list):
        return [moveTo(x, device) for x in obj]
    elif isinstance(obj, tuple):
        return tuple(moveTo(list(obj), device))
    elif isinstance(obj, set):
        return set(moveTo(list(obj), device))
    elif isinstance(obj, dict):
        to_ret = dict()
        for key, value in obj.items():
            to_ret[moveTo(key, device)] = moveTo(value, device)
        return to_ret
    elif hasattr(obj, "to"):
        return obj.to(device)
    else:
        return obj

some_tensors = [torch.tensor(1), torch.tensor(2)]
print(some_tensors)
print(moveTo(some_tensors, device))
```

```
[tensor(1), tensor(2)]
[tensor(1, device='cuda:0'), tensor(2, device='cuda:0')]
```

You should see above that the first time we printed the arrays we saw a `tensor(1)` and `tensor(2)`, but after using the `moveTo` function we saw `device=cuda:0` show up. We won't have to use this function often, but when we do, it will make our code easier to read and write. With that, we now have the fundamentals to write some *fast* code accelerated by GPUs.

Why do we care about GPUs?

Using a GPU is fundamentally about *speed*. It can literally be the difference between waiting *hours* or *minutes* for a neural network to train. And this is before we start considering very large networks or huge datasets. I've tried to make every individual neural network that is trained in this book take 10 minutes or less, in most cases under 5 minutes, when using a GPU. That means using toy problems and finagling them to show behavior that is representative of real-life.

Why not just learn on some real world data and problems? Because real-world neural networks can take *days* or *weeks* to train. Some of the research I have done for my day job can literally take *a month* to train using *multiple* GPUs. Now not all problems are that big and require that long, but its a realistic problem. The code we write is still perfectly good and valid for real world tasks, we will just need to wait a little longer for the results.

This long compute time also means that you need to learn skills on how to be productive while your model is training. One way to do this is to be developing new code for your next model on a spare machine or using the CPUs while your GPU is busy. You won't be able to train it, but you can push just a tiny amount of data through to make sure no errors happen. This is also why I want you to learn how to map the math used in deep learning to code. So while your model is busy training, you can be reading about the latest and greatest deep learning tools that might help you.

1.3 Automatic Differentiation

So far what we've seen is that PyTorch provides an API similar to NumPy for performing mathematical operations on tensors, with the advantage of using a GPU to perform faster math operations when available. The second major foundation that PyTorch gives us is called *automatic differentiation*. What this means is, as so long as we use PyTorch provided functions, PyTorch can compute derivatives and gradients automatically for us. In this section we are going to learn about what that means and how automatic differentiation ties into the task of minimizing a function. Then the next section we will see how to wrap it all up in a simple API provided by PyTorch.

Now your first thought might be "what is a gradient and why do I care about that?". We care because we can use the derivative of a function $f(x)$ to help us find an input x^* that is a *minimizer* of $f(x)$. This means the value of $f(x^*)$ will be smaller than $f(x^* + y)$ for whatever value we set y to. The mathy way to say this would be to state that $f(x^*) \leq f(y), \forall x \neq y$.

$$f(\underbrace{x^*}_{\text{this specific value}}) \overbrace{\leq}^{\text{is as small as or smaller}} f(y), \underbrace{\forall}_{\text{for all cases where...}} \underbrace{y \neq x}_{y \text{ is not equal to } x}$$

Another way to say this is that if I wrote the below code, I would be stuck waiting for an infinite loop.

```
while f(x_star) <= f(random.uniform(-1e100, 1e100)):
    pass
```

Why do we want to minimize a function? For all the kinds of machine learning and deep learning we will discuss in this course, we will train neural networks by defining a *loss function*. The loss function tells the network, in a numeric and quantifiable way, how *badly* it is doing at the problem right now. So if the loss is high, things are going poorly. A high loss means the network is "losing the game", and badly. If the loss is zero, the network has perfectly solved the problem. We don't usually allow the loss to go negative, because that gets confusing to think about.

When you read math about neural networks, you will often see the loss function defined as $\ell(x)$, where x are the inputs to the network, and $\ell(x)$ gives us the loss the network received. Because of this, *loss functions return scalars*. This is important, because we can compare scalars and say that one is definitively bigger or smaller than another, so it becomes unambiguous as to how "bad" a network is at the "game".

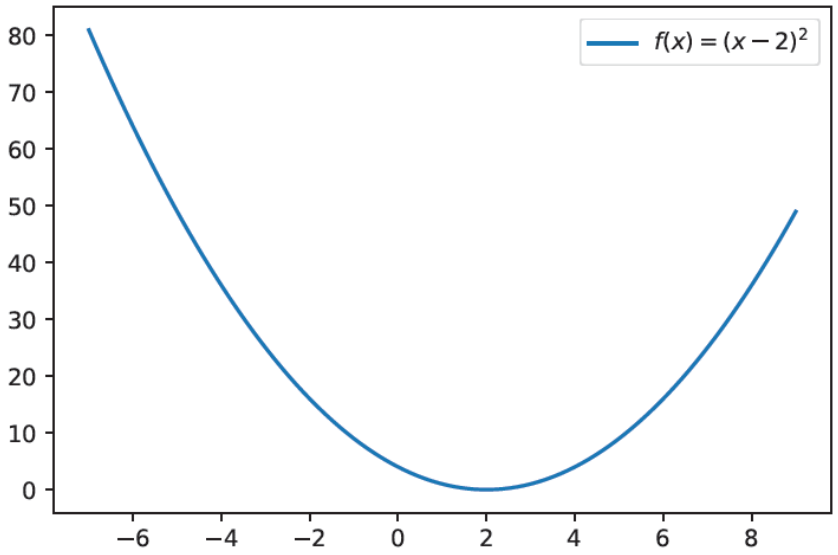
So we have stated that gradients are helpful, and perhaps you remember from a calculus class about minimizing functions using derivatives and gradients. Lets do a little bit of a math reminder about how we can find the minimum of a function using calculus.

Say we have the function $f(x) = (x-2)^2$, lets define that with some PyTorch code and plot what the function looks like.

```
def f(x):
    return torch.pow((x-2.0),2)

x_axis_vals = np.linspace(-7,9,100)
y_axis_vals = f(torch.tensor(x_axis_vals)).numpy()

sns.lineplot(x_axis_vals, y_axis_vals, label='$f(x)=(x-2)^2$')
```

```
/home/edraff/anaconda3/lib/python3.7/site-packages/seaborn/\_decorators.py:43:
 FutureWarning: Pass the following variables as keyword args: x, y. From version
0.12, the only valid positional argument will be `data`, and passing other
arguments without an explicit keyword will result in an error or
misinterpretation.
   FutureWarning
```

```
[22]:<AxesSubplot:>
```



Ok, we can clearly see that the minimum of this function is at $x = 2$, where we get the value of $f(2) = 0$. But this is an intentionally easy problem, so lets say we couldn't plot it. We can use calculus to help us find the answer.

We denote the derivative of $f(x)$ as $f'(x)$, and we can get the answer (using calculus) that $f'(x) = 2 \cdot x - 4$. The minimum of a function (x^*) exists at *critical points*, which are points where $f'(x) = 0$. So lets find them by solving for x. In our case we get:

$2 \cdot x - 4 = 0$
$2 \cdot x = 4$

(add 4 to both sides)

$x = 4/2 = 2$

(divide each side by 2)

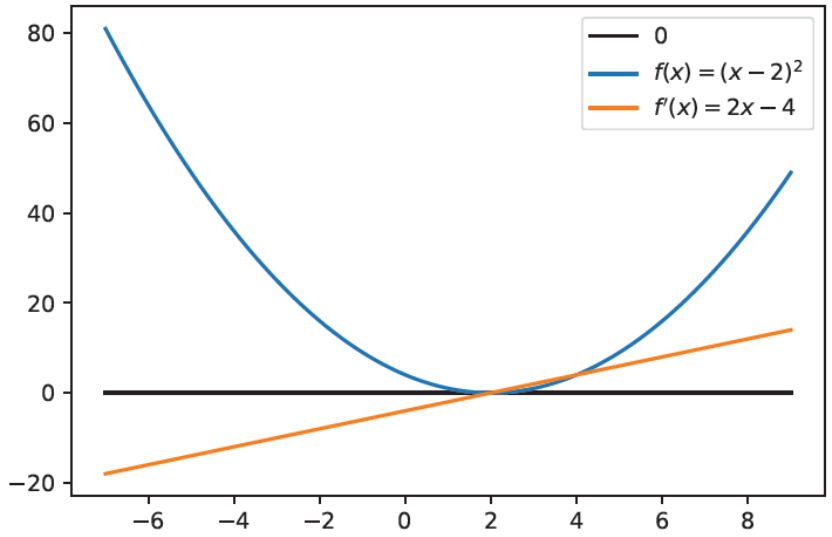
Now this required us to solve the above equation for when $f'(x) = 0$. PyTorch can't quite do that for us, because we are going to be developing more complicated functions where finding the *exact* answer is not possible. But say we have a current guess $x^?$, where we are pretty sure that it is not the minimizer. We can use $f'(x^?)$ to help us know how to adjust $x^?$ so that we move closer to a minimizer.

How is that possible? Lets plot $f(x)$ and $f'(x)$ at the same time.

```
def fP(x): #Defining the derivative of f(x) manually
    return2*x-4

y_axis_vals_p = fP(torch.tensor(x_axis_vals)).numpy()

#First, lets draw a black line at 0, so that we can easily tell if something is positive or
       negative
sns.lineplot(x_axis_vals, [0.0]*len(x_axis_vals), label="0", color='black')
sns.lineplot(x_axis_vals, y_axis_vals, label='$f(x) = (x-2)^2$')
sns.lineplot(x_axis_vals, y_axis_vals_p, label="$f'(x)=2 x - 4$")
```

```
/home/edraff/anaconda3/lib/python3.7/site-packages/seaborn/\_decorators.py:43:
 FutureWarning: Pass the following variables as keyword args: x, y. From version
0.12, the only valid positional argument will be `data`, and passing other
arguments without an explicit keyword will result in an error or
misinterpretation.
   FutureWarning
/home/edraff/anaconda3/lib/python3.7/site-packages/seaborn/\_decorators.py:43:
 FutureWarning: Pass the following variables as keyword args: x, y. From version
0.12, the only valid positional argument will be `data`, and passing other
arguments without an explicit keyword will result in an error or
misinterpretation.
   FutureWarning
/home/edraff/anaconda3/lib/python3.7/site-packages/seaborn/\_decorators.py:43:
 FutureWarning: Pass the following variables as keyword args: x, y. From version
0.12, the only valid positional argument will be `data`, and passing other
arguments without an explicit keyword will result in an error or
misinterpretation.
   FutureWarning
```

```
[23]:<AxesSubplot:>
```



Look at the orange line. You should notice that when we are too far left of the minimum ($x = 2$), we see that $f'(x^?) < 0$. When we are to the right of the minimum we instead get that $f'(x) > 0$. Only when we are at a minimum, do we see that $f'(x^?) = 0$. So if $f'(x^?) < 0$ we need to increase $x^?$ and if $f'(^?x) > 0$ we need to decrease the value of $x^?$. The *sign* of the gradient f' tells us which *direction* we should move to find a minimizer. This process of *gradient decent* is summarized in Figure 1.6.

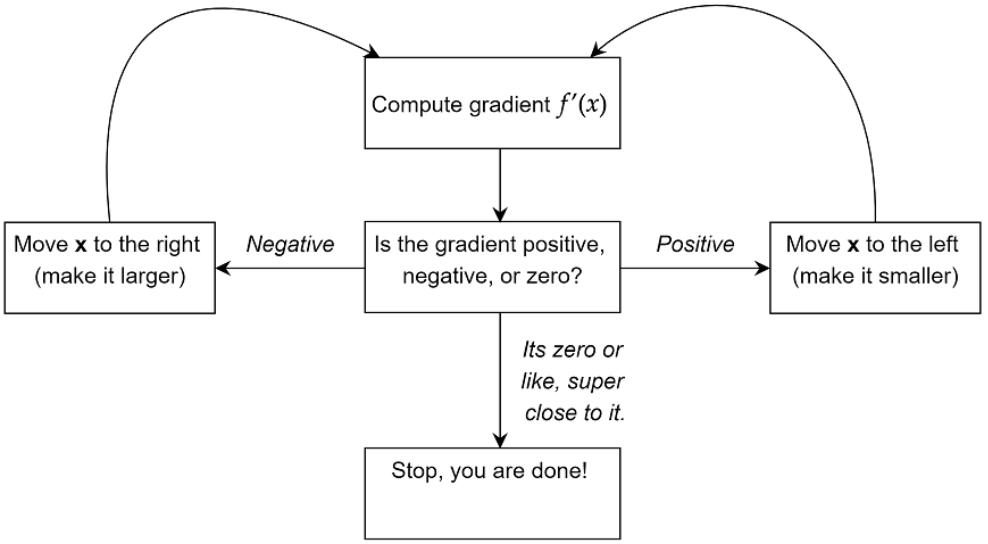


Figure 1.6: The process to minimize a function $f(x)$ using it's derivative $f'(x)$ is called *gradient descent*, and this figure shows how it is done. We iteratively compute $f'(x)$ to decide if we need to make x larger or *smaller* so that we make the value of $f(x)$ as small as possible. The process stops when we are "close enough" to the gradient being zero. You can also stop early if you have done a lot of updates, because "close enough is good enough" holds true for deep learning and we rarely need to perfectly minimize a function.

We also care about the *magnitude* of $f'(x^?)$. Because we are looking at a one dimensional function, the magnitude just means the absolute value of $f'(x^?)$, i.e., $|f'(x^?)|$. The magnitude gives us an idea about how far away we are from the minimizer. So the sign of $f'(x^?)$ (<0 or>0) tells us which *direction* we should head, and the size ($|f'(x)|$) tells us how *far* we should head.

This is not a coincidence. This will *always* be true for any function. So if we can compute a derivative, we can find a minimizer. Now, you may be thinking, "I don't remember my calculus all that well…" or complaining that I skipped the steps on how to compute $f'(x)$. This is precisely why we use PyTorch, automatic differentiation will compute the value of $f'(x)$ for us. Lets use this toy example of $f(x) = (x-2)^2$ to see how it works.

First, lets create a new variable that we want to minimize. We will do this similar to before, but we will add a new flag that tells PyTorch to start keeping track of the gradient. This gets stored in a variable called 'grad', which does not exist yet since we haven't actually computed anything.

```
x = torch.tensor([-3.5], requires_grad=True)
print(x.grad)
```

```
None
```

Ok, so we see there is no current gradient. Lets try computing $f(x)$ though and see if anything changes now that we set `requires_grad=True`.

```
value = f(x)
print(value)
```

```
tensor([30.2500], grad\_fn=<PowBackward0>)
```

Now when we print the value of the returned variable, we get a slightly different output. We see the first part where the value "30.25" is printed, which is the correct value of $f(-3.5)$. But we also see this new `grad_fn=<PowBackward0>`. Once we tell PyTorch to start calculating gradients, it will begin to keep track of *every* computation we do. It uses this information to go backwards and calculate the gradients for everything that was used and had a `requires_grad` flag set to `True`.

Once we have a single *scalar* value, we can tell PyTorch to go back and use this information to actually compute the gradients. This is done using the `.backward()` function, after which we will see there is now a gradient in our original object.

```
value.backward()
print(x.grad)
```

```
tensor([-11.])
```

That covers the mechanics of how PyTorch can compute gradients for us. Now we can use this automatic differentiation of our PyTorch function $f(x)$ to numerically find the answer that $f(2) = 0$. First we are going to describe it using a mathematically notation, and then again in actual code.

We start with our current guess $x_{cur} = -3.5$. I've chosen 3.5 arbitrarily, in real life you would usually pick a random value. We will also keep track of our previous guess using x_{prev}. Since we have not done anything yet, its fine to set the "previous" step to any large value (e.g., $x_{prev} = x_{cur} * 100$).

Next, we compare if our current and previous guesses are very similar. We do this by checking if $\|x_{cur} - x_{prev}\|_2 > \epsilon$. The function $\|z\|_2$ is called the *norm* or *2-norm*. Norms are the most common and standard ways of measuring *magnitude* for vectors and matrices. For one dimensional cases (like this one), the 2-norm is the exact same thing as the absolute value. If we do not explicitly state what kind of norm we are talking about, you should always assume the 2-norm. The value ϵ is a common mathematical notation to refer to some arbitrary small value. So the way to read this is:

$$\overbrace{\|x_{cur} - x_{prev}\|_2}^{\text{the difference between our previous and current solution}} \overbrace{>}^{\text{is greater than}} \overbrace{\epsilon}^{\text{a small value}}$$

Ok, so now we know that $\|x_{cur} - x_{prev}\|_2 > \epsilon$ is how we check if there are large ($> \epsilon$) magnitude ($\| \cdot \|_2$) changes ($x_{cur} - x_{prev}$) between our guesses. If this is false, that means $\|x_{cur} - x_{prev}\|_2 \leq \epsilon$, which means the change was small, and that we can stop. Once we stop, we accept x_{cur} as our answer to the value of x that minimized $f(x)$ If not, we need a new *better* guess.

To get this new guess, we are going to move in the *opposite* direction of the derivative. This looks like: $x_{cur} = x_{cur} - \eta \cdot f'(x_{cur})$. The value η is called the *learning rate* , and is usually a small value like $\eta = 0.1$ or $\eta = 0.01$. We do this because the gradient $f'(x)$ tells us which way to head, but only gives us a *relative* answer about how far away we are. It doesn't tell us exactly how far we should travel in that direction. Since we don't know how for to travel, we want to be conservative and go a little slower. Figure 1.7 shows why.

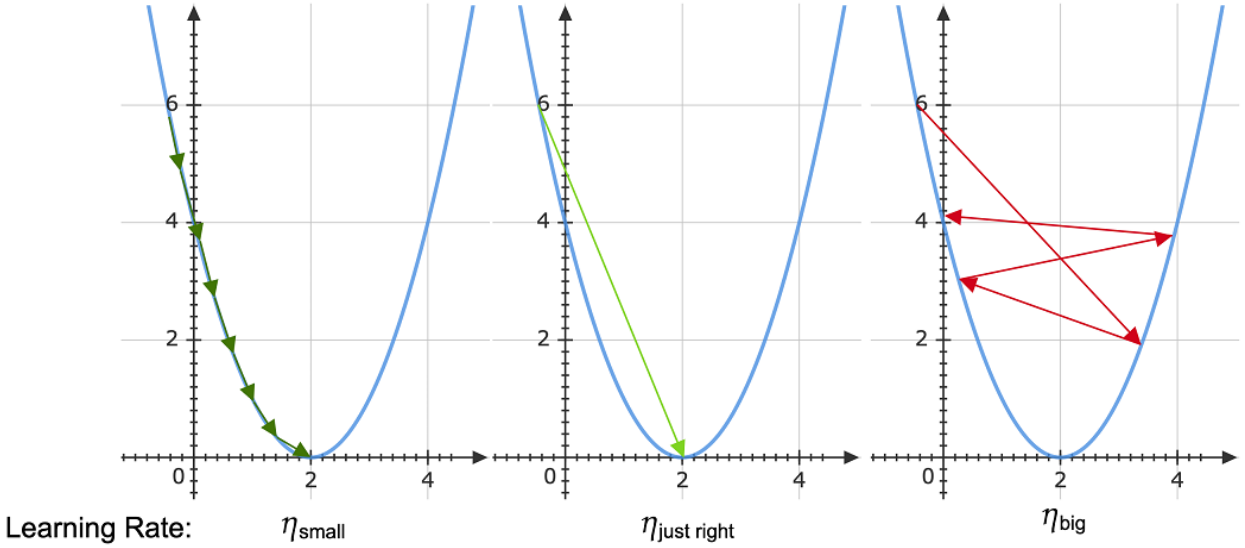


Figure 1.7: Three examples of how the learning rate η (also called "step size") impacts learning. On the left we have η being *small*er than necessary. This still reaches the minimum, but takes more steps than needed. If we knew the perfect value of η can could set it *just right*, we could take the minimum number of steps to the minimum (middle). On the right η can be set too *big*, which causes divergence. We will never reach the solution!

By taking smaller steps in the current direction we don't "drive past" the answer, and have to turn back around. Look at the above example of our function to understand how that happens. If we have the *exactly* correct best value of η (middle image), we can take one step to the minimum. But we do not know what that value is. If we are instead conservative, and choose a value that is likely smaller than we need we may take more steps to get to the answer, but we will eventually get there (left image). But if we set our learning rate too high, we can end up shooting past the solution and bouncing around it instead (right image).

That might sound like a lot of scary math, but hopefully you will feel better about it when you look at the code that does the work. Its only a few lines long. At the end of the loop, we print out the value of x_{cur} and see that it is equal to 2.0, PyTorch found the answer.

```
x = torch.tensor([-3.5], requires_grad=True)

x_cur = x.clone()
x_prev = x_cur*100 #Make the initial "previous" solution larger
epsilon = 1e-5
eta = 0.1

while torch.norm(x_cur-x_prev) > epsilon:
    x_prev = x_cur.clone() #We need to make a clone here so that x_prev and x_cur don't point to
       the same object

    #Compute our function, gradient, and update
    value = f(x)
    value.backward()
    x.data -= eta * x.grad
    x.grad.zero_() #We need to zero out the old gradient, as py-torch will not do that for us

    #What are we currently now?
    x_cur = x.data

print(x_cur)
```

```
tensor([2.0000])
```

I Keep Hearing About Backpropagation?

Many books begin their discussion about deep learning with an algorithm called *backpropagation*. This is the name of the original algorithm used to compute all the gradients in a neural network. Personally, I think backpropagation is a very intimidating place to start, as it involves a lot more math and drawing graphs, but is fully encapsulated by automatic differentiation. Go ahead and look it up online if you like, there are tons of tutorials out there. But with modern frameworks like PyTorch, I don't think you need to know about the mechanics of back backpropagation to get *started* and build good intuitions.

1.4 Optimizing Parameters

What we just did, finding the minimum of a function $f(\cdot)$, is called *optimization*. Because we will specify the goal of our network using a loss function $\ell(\cdot)$, we can optimize $f(\cdot)$ to minimize our loss. If we reach a loss $\ell(\cdot) = 0$, that means our network (appears) to have perfectly solved the problem. This why why we care about optimization. This is foundational to how most modern neural networks are trained today, and Figure 1.8 shows a simplification of how this ends up working.

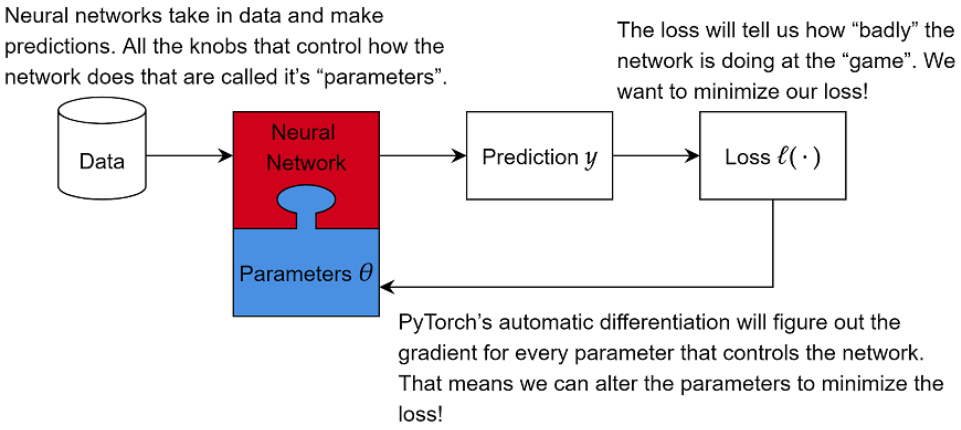


Figure 1.8: Depiction of how the loss $\ell(\cdot) = 0$ and optimization process gets used by neural networks. The neural network is controlled by it's parameters Θ. To make useful predictions on the data, we need to alter the parameters. This is done by first computing the loss $\ell(\cdot) = 0$ that tells us how badly the network is doing. Since we want to minimize the loss, we can use the gradient to alter the parameters! This gets the network to make useful predictions.

Because of how important optimization is, PyTorch includes some two additional concepts to help us: Parameters and Optimizers. A `Parameter` of a model is a value that we want to alter using an `Optimizer`, so that we can try and reduce our loss $\ell(\cdot)$. We can easily convert any tensor into a `Parameter` using the `nn.Parameter` class. So to do that, lets re-solve the previous problem of minimizing $f(x) = (x-2)^2$ with an initial guess of $x_{cur} = 3.5$. The first thing we will do is create a `Parameter` object for the value of x, since that is what we are going to alter.

```
x_param = torch.nn.Parameter(torch.tensor([-3.5]), requires_grad=True)
```

The object `x_param` is now a `nn.Parameter`, which behaves the same way as Tensors do. We can use a `Parameter` anywhere that we would use a tensor in PyTorch and the code will work just fine. But now we can create an `Optimizer` object. The simplest optimizer we will use is called `SGD`, which stands for *Stochastic Gradient Descent*. The word "gradient" is there because we are using the gradients/derivatives of functions. "Descent" because we are minimizing or "descending" to a lower value of the function that we are minimized. We will get to the "stochastic" part in the next chapter.

To use SGD, we simply need to create the associated object with a `list` of `Parameters` that we want to adjust. We can also specify the learning rate η, or accept the default. The below code we will specify eta to match the original code above.

```
optimizer = torch.optim.SGD([x_param], lr=eta)
```

Now we can re-write the previous ugly loop into something cleaner, that looks much closer to how we will train neural networks in practice. We will loop over the optimization problem a fixed number of times, which we often call *epochs*. The `zero_grad` method does the cleanup we did manually before for *every* parameter that was passed in as an input. We compute our "loss", call `.backward()` on that loss, and then ask the optimizer to perform one `.step()` of the optimization.

```
for epoch in range(60):
    optimizer.zero_grad() #x.grad.zero_()
```

```
    loss_incurred  = f(x_param)
    loss_incurred.backward()
    optimizer.step() #x.data -= eta * x.grad
print(x_param.data)
```

And now you should see the code prints out tensor(2.0000), just like before. This will make our lives easier when we have literally *millions* of parameters in our network.

You'll notice a big change in this code is that we are not optimizing until we hit a gradient of zero or until the difference between previous and current solutions is very small. Instead we are doing something dumber, just doing a fixed number of steps. In deep learning we rarely get to a loss of zero, and we would be waiting way too long for that to happen. So what most people do is pick a fixed number of epochs that they are wiling to wait for, and see what the results look like at the end. This way we get an answer faster, and usually its good enough that we are happy to use it.

1.5 Loading DataSet Objects

Now we have learned a little bit about the basic tools PyTorch provides us. We want to start training a neural network to solve our problems. But first, we need some data. Using the common notation of machine learning, we need a set of input data X and associated output labels y. In PyTorch we will represent that with a `Dataset` object. By using this interface, PyTorch provides us with efficient loaders that will automatically handle using multiple CPU cores to pre-fetch the data and only keep a limited amount of data in-memory at a time. Lets start by loading a familiar dataset from ScikitLearn, MNIST. We will convert it from a NumPy array to the form that PyTorch would like to see that data as.

PyTorch uses a `Dataset` class to represent a dataset, and it encodes the information about: 1) How many items are in the dataset? 2) How to get the n'th item in the data set. So lets go ahead and look at what that looks like.

```
from torch.utils.data import Dataset
from sklearn.datasets import fetch_openml

# Load data from https://www.openml.org/d/554
X, y = fetch_openml('mnist_784', version=1, return_X_y=True)
print(X.shape)

(70000,784)
```

We can see that we have loaded the classic MNIST dataset with a total of 70,000 rows and 784 features. Now we will crease a simple Dataset class that takes in `X, y` as input. We need to define a `__getitem__` method, which will return the data and label as a `tuple(inputs, outputs)`. The `inputs` are the objects we want to give to our model as inputs, and the `outputs` are used for the output. We also need to implement the `__len__` function that returns how large the dataset is.

```
class SimpleDataset(Dataset):

    def __init__(self, X, y):
        super(SimpleDataset, self).__init__()
        self.X = X
        self.y = y

    def __getitem__(self, index):
        inputs = torch.tensor(self.X[index,:], dtype=torch.float32)
        targets = torch.tensor(int(self.y[index]), dtype=torch.int64)
        return inputs, targets
```

```
    def __len__(self):
        return self.X.shape[0]
#Now we can make a PyTorch dataset
dataset = SimpleDataset(X, y)
```

Now we have a simple dataset object. It keeps the entire dataset in memory, which is OK for small datasets, but something we will want to fix in the future. We can confirm that the dataset still has 70,000 examples, and each example as 784 features, just as before. We can quickly confirm that the length and index functions we implemented work as expected.

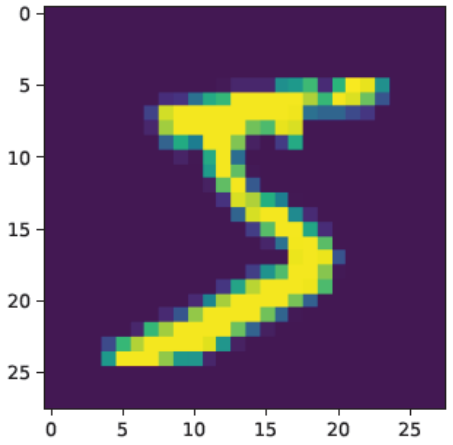
```
print("Length: ", len(dataset))
example, label = dataset[0]
print("Features: ", example.shape) #Will return 784
print("Label of index 0: ", label)
```

```
Length: 70000
Features:  torch.Size([784])
Label of index0:  tensor(5)
```

The MNIST dataset is a dataset of hand drawn numbers. We can visualize this by re-shaping the data back into an image, just to confirm our data loader is working.

```
plt.imshow(example.reshape((28,28)))
```

```
[34]: <matplotlib.image.AxesImage at  0x7feec703f850>
```



1.5.1 Creating a Training and Testing Split

Now we have*all* our data in one dataset. However, like good machine learning practitioners, we should create a training and a testing split. In some cases we have a dedicated training and testing dataset. If that is the case, you should create two separate `Dataset` objects, one for training and one for testing, from the respective data sources.

In this case, we just have one dataset. PyTorch has a simple utility that can help us break the corpus into a train and test split. Lets say we wanted 20% of the data to be used for testing, that could be done as follows using the `random_split` method.

```
train_size = int(len(dataset)*0.8)
test_size = len(dataset)-train_size
```

```
train_dataset, test_dataset = torch.utils.data.random_split(dataset, (train_size, test_size))
print(" {} examples for training and  {} for testing".format(len(train_dataset),
       len(test_dataset)))
```

```
56000 examples for training and14000 for testing
```

And now we have a train and test set. In reality, the first 60,000 points are the standard training set for MNIST and the last 10,000 the standard test set. But the point was to show you the function for creating randomized splits yourself.

With that, we have learned about all of the foundational tools that PyTorch provides.

1. A NumPy like tensor API, which supports GPU acceleration.
2. Automatic differentiation, which lets us solve optimization problems
3. An abstraction for datasets.

We will start building on this foundation, and you may notice it starts to impact how you think about neural networks in the future. They do not magically do what is asked, but they try to numerically solve a goal specified by a loss function $\ell(\cdot)$. We need to make sure we are careful in how we define or choose $\ell(\cdot)$, because that will determine what the algorithm learns.

1.6 Exercises

1. Write a series of `for` loops that compute the average value in `torch_tensor3d`
2. Write code that indexes into `torch_tensor3d` and prints out the value "13".
3. For every power of 2 (i.e., 2^i or `2**i`) up to 2^{11}, create a random matrix $X \in \mathbb{R}^{2^i, 2^i}$ (i.e., `X.shape` should give `(2**i, 2**i)`). Time how long it takes to compute XX (i.e., `X @ X`) on a CPU and on a GPU and plot the speedup. For what sized matrices is the CPU faster than the GPU?
4. We used PyTorch to find the numeric solution to $f(x) = (x-2)^2$. Write code that will find the solution to $f(x) = \sin(x\text{-}2) \cdot (x\text{+}2)2\text{+}\sqrt{|\cos(x)|}$. What answer do you get?
5. Write a new function that takes two inputs, x and y, where $f(x,y) = \exp(\sin(x)^2)/(x-y)^2 + (x-y)^4$. Use an `Optimizer` with initial parameter values of $x = 0.2$ and $y = 10$. What do they converge to?
6. Create a function `libsvm2Dataset` that takes a path to a libsvm dataset file (see https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/ for many that you can download) and creates a new dataset object. Check that is has the correct length, and that each row has the expected number of features.
7. **Challenging**: Use NumPy's `memmap` functionality to write the MNIST dataset to disk. Then create a `MemmapedSimpleDataset` that takes the mmmaped file as input, reading the matrix from disk to create PyTorch tensors in the `__getitem__` method. Why do you think this would be useful to have?

1.7 Summary

1. PyTorch represents almost everything using Tensors, which are multi-dimensional arrays.
2. Using a GPU PyTorch can accelerate any operation done using Tensors
3. PyTorch tracks what we do with tensors to perform "automatic differentiation", which means it can calculate gradients for us.

4. We can use gradients to minimize a function, the values altered by a gradient are "parameters".
5. We will use a "loss function" to quantify how well a network is doing at a task, and use gradients to minimize that loss, which results in "learning" the parameters of the network.
6. PyTorch provides a `Dataset` abstraction, so that we can let PyTorch deal with a number of nuisance tasks on minimizing memory usage.

FOOTNOTES

1. Some may take offense at how I'm categorizing these groups. It's an oversimplification.↩
2. Some like to say "Good Old Fashioned AI", or "GOFAI", for using these classic tried-and-true methods.↩
3. Back in my day, we had to write our neural networks by scratch. In my father's day, they had to walk two miles up hill to use a computer - and the walk back was still up-hill!↩

2

Fully Connected Networks

This chapter covers:

- Implementing a training loop in PyTorch
- Changing loss functions for regression and classification problems
- Implementing and training a fully connected network
- Training faster using smaller batches of data

Now that we know how PyTorch gives us Tensors to represent our data and parameters, we can progress to building our first neural networks. This starts with showing how *learning* happens inside PyTorch. As we described in the last chapter learning is based on the principle of optimization: we can compute a loss for how well we are doing and use gradients to minimize that loss. This is how neural networks the parameters of a network are "learned" from the data, and is also the basis of many different machine learning algorithms. For these reasons optimization of loss functions is the foundation PyTorch is built from. So to implement any kind of neural network in PyTorch we must phrase the problem as an optimization problem (remember that this is also called function minimization).

In this chapter, we will first learn how to set up this optimization approach to learning. It is a widely applicable concept, and the code we write will be usable for almost any neural network you run across. This process is called a *training loop*, and even works for simple bread-and-butter ML algorithms like linear and logistic regression. Since we are focusing on the mechanics of training, we will start with these two basic algorithms so we can stay focused on how training loops work in PyTorch for classification and regression.

The vector of weights that makes a logistic/linear regression is also called a *linear layer* or a *fully connected* layer. This means both can be described as a "single layer" model in PyTorch. Since neural networks can be described as a sequence of layers, we will modify our original logistic and linear models to become full fledged neural networks.In doing so you will learn the importance of a non-linear layer, and how logistic and linear regression are related to each other and to neural networks.

After mastering these concepts of a training loop, classification and regression loss functions, and defining a fully-connected neural network, we will have covered the foundational concepts of deep learning that will re-occur in almost every model you will ever train. To end out the chapter we will refactor our code into a convenient helper function, and learn the practical utility of training on small groups of data called *batches* instead of using the entire dataset.

2.1 Neural Networks as Optimization

Last chapter we used PyTorch's *automatic differentiation* capability for optimizing (read, minimizing) a function. We defined a loss function to minimize and used the `.backward()` function to compute gradients, which told us how to alter the parameters to minimize the function. If we *make the input to the loss function a neural network*, then we can use this exact same approach to train a neural network. This creates a process called a *training loop* with three major components: 1)the training data (with correct answers), 2) the model and loss function, and 3) the update via gradients. These three components are outlined in Figure 2.1.

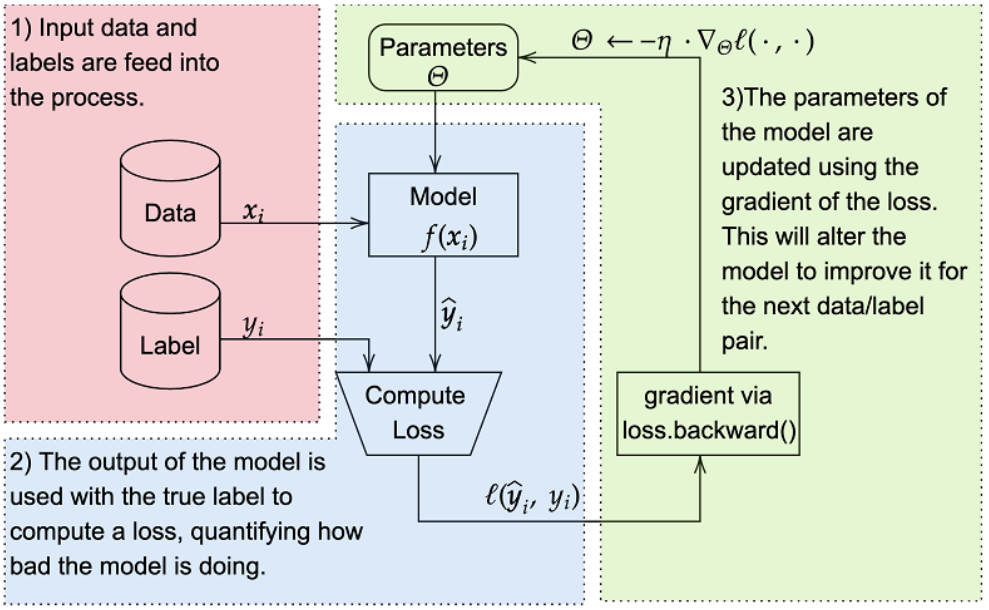


Figure 2.1: The three major steps to training a model in PyTorch. 1) The input data that drives learning. 2) The model is used to make a prediction, and a loss scores how much the prediction and true label differ. 3) PyTorch's automatic differentiation is used to update the parameters of the model, improving its predictions.

Before we start lets introduce some standard notation we will re-use throughout this book. Lets use $\boldsymbol{x}$ to denote our input features, and $f()$ to denote our neural network model. The label associated with $\boldsymbol{x}$ is denoted with y. Our model takes in $\boldsymbol{x}$, and produces a prediction $\hat{y}$. Written out, this becomes $\hat{y} = f(\boldsymbol{x})$. This notation is widely used within deep learning papers, and getting familiar with it will help you stay up-to-date as new approaches are developed in the future.

Our model needs to have some parameters to adjust. Changing the parameters allows the network to alter it's predictions to try and reduce the loss function. We will denote in abstract *all*

the parameters of our model using Θ. If we want to be explicit, we might even say that $\hat{y} = f_\Theta(\boldsymbol{x})$ to state that the model's prediction and behavior is dependent on the value of it's parameters Θ. You will also see Θ called the "state" of the model.

Now we have a notation and language for describing our model, we also need a way to frame the goal as one of function minimization. To do this, we will use the concept of a *loss function*. The loss function *quantifies* just how badly our model is doing at the goal of predicting the ground truth y. If y is our goal, and $\hat{y}$ is the prediction, then we will denote our loss function as $\ell(y, \hat{y})$.

Now we have all we need to abstractly describe learning as a function minimization problem. Lets say we have a training set with N examples, which is done with the equation $\min_\Theta \sum_{i=1}^{N} \ell(f_\Theta(\boldsymbol{x_i}), y_i)$. Lets write out in English what this equation says, and we will color-code each English description to the math that says the same thing.

Alter the parameters Θ to minimize

the error/loss of the

neural network's prediction

against the correct predictions

over the entire dataset.

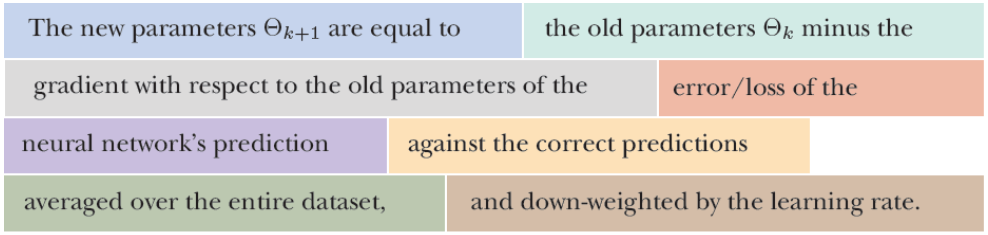
$$\min_\Theta \quad \sum_{i=1}^{N} \quad \ell(\quad f_\Theta(\boldsymbol{x}_i), \quad y_i \quad)$$

By breaking up and looking at each piece of the math we can see how this is just describing our goal, and that the math allows us to convey a long sentence in less space. As code, that might instead look like

```
def F(X, y, f, theta):
    total_loss =0
    for i in range(N):
        total_loss += loss(f(X[i,:], theta), y[i])
    return total_loss
```

The summation ($\sum_{i=1}^{N}$) is going over all N pairs of input ($\boldsymbol{x_i}$) and output (y_i), and determining just how badly ($\ell(\cdot, \cdot)$) we are doing. The big question is, how do we adjust Θ to do this?

We do this by gradient descent, which is the reason PyTorch provides automatic differentiation. Lets say that Θ_k is the *current* state of our model, which we want to improve. How do we find the next state Θ_{k+1}, that hopefully reduces the loss of our model? The math equation we want to solve is $\Theta_{k+1} = \Theta_k - \eta \cdot \frac{1}{N} \sum_{i=1}^{N} \nabla_{\Theta_k} \ell(f_{\Theta_k}(\boldsymbol{x_i}), y_i)$. Again, lets write out what this equation says in English and map it back to the symbols:



$$\Theta_{k+1} = \Theta_k - \eta \cdot \frac{1}{N}\sum_{i=1}^{N} \nabla_{\Theta_k} \ell\left(f_{\Theta_k}(\boldsymbol{x}_i), y_i \right)$$

The above equation shows the math for what is known as *gradient decent*. This looks almost exactly the same as what we did in chapter 1 to optimize a simple function. This biggest difference is this fancy new ∇ symbol. This "nabla" symbol ∇ is used to denote the *gradient*. Last chapter we used derivative and gradient interchangeably, because we only had *one* parameter. Since we have a whole set of parameters now, the gradient is the language we use to refer to the derivative with respect to every parameter. If we want to only alter a select sub-set z of the parameters, we would write that as ∇_z. This means ∇ is going to be a tensor, with one value for *every* parameter.

We follow the gradient (∇) to tell us how to adjust Θ, and just like before we head in the opposite direction of the sign. The important thing to remember here is that PyTorch provides us with automatic differentiation. That means if we use the PyTorch API and framework, we do not have to worry about how to compute ∇_Θ. In fact, we don't have to keep track of everything inside of Θ either.

All we *need* to do, is define what our model $f(\cdot)$ looks like, and what our loss function $\ell(\cdot, \cdot)$ is. That alone will take care of almost all the work for us. We can write a function that performs this whole process right now.

2.1.1 Linear Regression

The framework we have described to train a model $f(\cdot)$ using gradient descent is widely applicable. The process demonstrated by Figure 2.1 of iterating over all the data and performing these gradient updates is the training loop. We can re-create many different types of machine learning methods, like Linear and Logistic regression, using PyTorch and this approach. To do so, one needs to simply define $f(\cdot)$ in the correct manner. In fact, we are going to start by re-creating one of the bread-and-butter algorithms, Linear Regression, to introduce all of the code infrastructure that PyTorch will provide for us to build into a larger neural network later.

First thing to do is make sure we have all the needed some standard imports. From PyTorch this includes `torch.nn` and `torch.nn.functional`, which will provide some common building blocks that we will make use of throughout this book. `torch.utils.data` has the tools for working with `Datasets`, and `idlmam` will provide code we have written in previous chapters as we progress in this book.

```
import torch
import torch.nn as nn
```

```
import torch.nn.functional as F
from torch.utils.data import *
from idlmam import *
```

2.1.2 The Training Loop

Now that we have those additional imports, lets start by writing a *training loop*. Lets assume we have a loss function `loss_func` ($\ell(\cdot,\cdot)$), that takes in an `prediction` ($\hat{y}$) and a `target` (y), returning a single score for how well a `model` ($f(\cdot)$) has done. We will need an iterator that loads the training data for us to train on. This `training_loader` will gives us pairs of `inputs` with their associated `labels` for training.

A diagram showing all the steps of a training loop are shown in the following figure. The yellow "Prep" section shows the object creation that needs to be done before the training can start. We have to pick the device that will do all the computations (normally a GPU), define our model $f(\cdot)$, and create an optimizer for the model's parameters θ. The red regions indicate the start/repetition of the "loop", which provides new data for us to train on. The blue region computes the prediction $\hat{y}$ and loss $\ell(\hat{y},y)$ for the model with it's *current* parameters Θ. The green section is taking the loss and computing the gradients and updates to alter the parameters Θ. Notice that the colors match up to the steps we saw in , but with a little more detail showing the PyTorch functions we need to call.

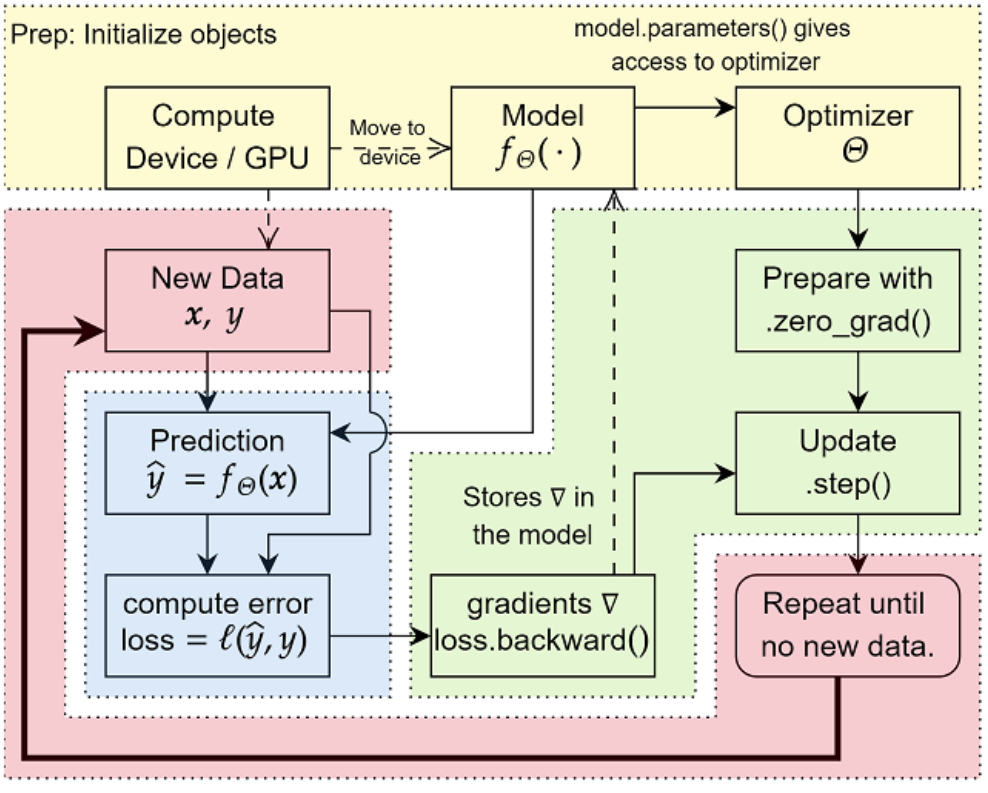


Figure 2.2: Diagram of the training loop process. It includes the same three major steps to training a model in

PyTorch we originally described with matching color-coding. New is the intialization of the objects which are re-used on ever training loop. Solid areas show steps, dashed arrows show effects.

Using PyTorch, we can write a minimal amount of code that is enough to train many different kinds of neural networks. The `train_simple_network` function in the next code block that follows all the parts of the Figure 2.2 process. Lets talk about what happens in this function in order. First we create an `optimizer` that takes in the model's `parameters()` Θ that will be altered. We then move the model to the correct compute `device`, and repeat the optimization process for some number of `epochs`. Each epoch means we used every data point $\boldsymbol{x}_i$ *once*.

Each epoch involves putting the model into training mode with `model.train()`. The `training_loader` gives us our data in groups of tuples $(\boldsymbol{x}, y)$, which we move to the same compute `device`. The inner loop over these tuples will clean up the optimzer state with `zero_grad()`, then pass the `inputs` to the `model` to get a prediction `y_hat`. Our `loss_fun` takes in the prediction `y_hat` and the true `labels` to calculate a `loss`, describing how badly our network has done. Then we compute the gradients with `loss.backward()` and take a `step()` with the `optimizer`.

```
def train_simple_network(model, loss_func, training_loader, epochs=20, device="cpu"):
    #Yellow step is done here. We create the optimizer and move the model to the compute device
    #SGD is Stochastic Gradient Decent over the parameters Θ
    optimizer = torch.optim.SGD(model.parameters(), lr=0.001)

    #Place the model on the correct compute resource (CPU or GPU)
    model.to(device)
    #The next two for loops handle the Red steps, iterating through all the data (batches)
       multiple times (epochs)
    for epoch in tqdm(range(epochs), desc="Epoch"):

        model = model.train()#Put our model in training mode
        running_loss = 0.0

        for inputs, labels in tqdm(training_loader, desc="Batch", leave=False):
            #Move the batch of data to the device we are using. this is the last red step
            inputs = moveTo(inputs, device)
            labels = moveTo(labels, device)

            #First a yellow step, prepare the optimizer. Most PyTorch code will do this first to
      make sure everything is in a clean and ready state.

            #PyTorch stores gradients in a mutable data structure. So we need to set it to a
      clean state before we use it.
            #Otherwise, it will have old information from a previous iteration
            optimizer.zero_grad()

            #The next two lines of code perform the two blue steps
            y_hat = model(inputs) #this just computed f_θ(x_i)

            # Compute loss.
            loss = loss_func(y_hat, labels)

            #Now the remaining two yellow steps, compute the gradient and ".step()" the
      optimizer
            loss.backward()# ∇_Θ just got computed by this one call

            #Now we just need to update all the parameters
            optimizer.step()# Θ_{k+1} = Θ_k − η · ∇_Θ ℓ(ŷ, y)
```

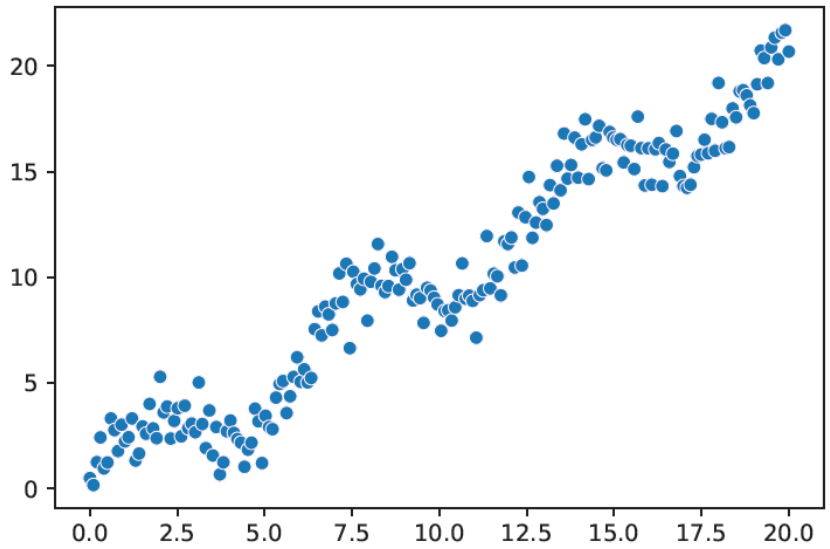
```
            #Now we are just grabbing some information we would like to have
            running_loss += loss.item()
#Caption: This code defines a simple training loop, which can be used to learn the parameters Θ
      to almost any neural network f_θ(·) we will use in this book.
```

This code we have just described is sufficient to train almost all of the neural networks we will design during this book. Now that we have the code to train a network, we need to come up with some data, a network and a loss function to work with. Lets start by learning a simple *Linear Regression* model. You should recall from a machine learning class or training that *regression* problems are ones where we want to predict a numeric value. For example, if we wanted to predict the miles per gallon (mpg) of a car based on its features (e.g., weight in pounds, engine size, year produced), that would be a regression problem. Thats because the mpg could be 20, 24, 33.7, or hypothetically even 178.1342, and most any possible single number.

Below, we've created a synthetic regression problem that has a linear and non-linear component, with some noise added to make it interesting. because of how strong the linear component is, a linear model will do *ok* , but won't be perfect.

```
#Crate a 1-dimensional input
X = np.linspace(0,20, num=200)
#create an output
y = X + np.sin(X)*2 + np.random.normal(size=X.shape)
sns.scatterplot(x=X, y=y)
```

```
[6]: <AxesSubplot:>
```



Ok, we have created a nice simple toy problem where there is a strong linear trend, with a smaller but consistent oscillation up and down. We use toy problems like to to do experiments so we can see the results and get a more intuitive understanding of what is happening. But we need it in a form that PyTorch will understand. Below we create a simple dataset object that knows we have a 1-dimensional problem. The training data will be shaped as $(n, 1)$ matrix, where n is the number of data points we have. The labels (y) will take a similar form. When we get an item, we

grab the correct row of the dataset, and return a PyTorch `tensor` object that is a `torch.float32` type, which is the default type you should be using for most things in PyTorch.

```
class Simple1DRegressionDataset(Dataset):

    def __init__(self, X, y):
        super(Simple1DRegressionDataset, self).__init__()
        self.X = X.reshape(-1,1)
        self.y = y.reshape(-1,1)


    def __getitem__(self, index):
        return torch.tensor(self.X[index,:], dtype=torch.float32), torch.tensor(self.y[index],
       dtype=torch.float32)

    def __len__(self):
        return self.X.shape[0]

training_loader = DataLoader(Simple1DRegressionDataset(X, y), shuffle=True)
```

NOTE **The use of the `reshape` function is an important one to understand, with behavior we will regularly use throughout this book. Suppose we have a tensor with 6 total values. That could be a vector of length 6, a 2×3 matrix, a 3×2, or it could even be a tensor with three dimensions where one dimension has a size of "1". So long as the total number of values stays the same, we can *reinterpret* the tensor as having a different shape, where the values have just been moved around.**

Figure 2.3 shows how this can be done. What is special about reshape is it lets us specify all but one of the dimensions, and automatically puts the "leftovers" into the unspecified dimension. This "leftover" dimension is denoted with `-1`, and we can see how as we add more dimensions, there are more ways we can ask NumPy or PyTorch to reshape a tensor. The `view` function has the same kind of behavior. Why have two functions with the same behavior? The `view` function uses less memory and can be faster, but can throw errors if used inappropriately. You will learn the details with more advanced use of PyTorch, but for the sake of simplicity you can *always* call `reshape` safely.

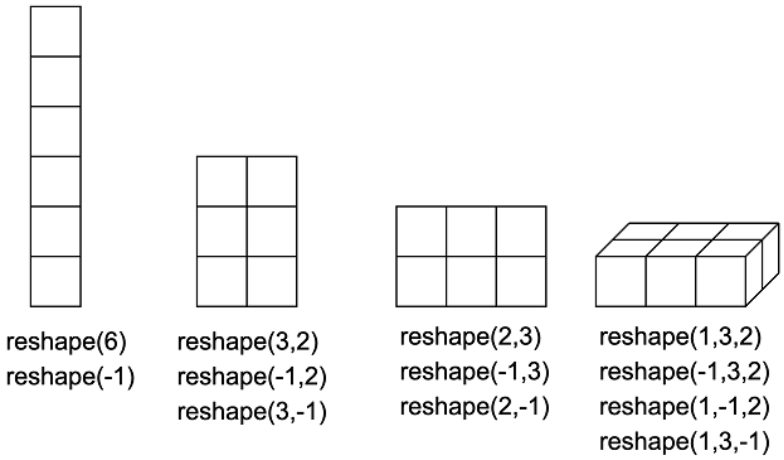


Figure 2.3: Example of four different tensor shapes that can be used to represent the same six values. The *reshape* function takes as many arguments as you want axes in the resulting tensor. If you know the exact size you can

specify it, or if you don't know the size for one axis you can use "-1" to indicate a "leftover" spot.

2.1.3 Defining the Model

At this point we have successfully created a training loop, and a `Dataset` object to load our dataset. The last thing we are missing is a PyTorch model that implements the Linear Regression algorithm as a network. This is the "Model" box from , which I've isolated in Figure 2.4

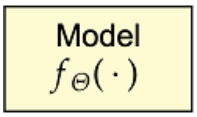


Figure 2.4: The "Model" box denotes the neural network $f_\theta(\cdot)$ that we want to train. This could be a small network, or a very complex one. Either way it is encapsulated into one model object with a single set of parameters θ, and the same process will work for almost any network definition.

Defining the network we will use is very easy in this case. We want a simple linear function, and a linear function means a weight matrix W. Applying a weight matrix W to an input $\boldsymbol{x}$ means taking the matrix vector product. So we want a model $f(\cdot)$ that looks like :

$$f(\boldsymbol{x}) = \underbrace{\boldsymbol{x}^\top \overbrace{\boldsymbol{W}^{d,C}}^{\text{Linear layer/function that takes in } d \text{ inputs and produces } C \text{ outputs.}}}_{\text{dot product/matrix vector product between input and linear layer.}}$$

The vector $\boldsymbol{x}$ has all of our d features (in this case, $d = 1$), and the matrix W will have a row for every feature, and a column for every output. Using $W^{d,\,C}$ is us being extra explicit that it is a tensor with d rows and C columns. Thats a notation we and others will use in this book to be precise about the shape of certain objects. Since we are predicting a single value, this means $C = 1$.

Notice though that this linear function is not quite complete. If $x = 0$, then $f(\boldsymbol{x}) = 0$. Thats a very strong constraint to have on a model. Instead, we will add a *bias term* $\boldsymbol{b}$ that has no interaction with $\boldsymbol{x}$. This looks like :

$$f(\boldsymbol{x}) = \boldsymbol{x}^\top \underbrace{\boldsymbol{W}^{d,C} + \boldsymbol{b}}_{\text{nn.Linear(d, C)}}$$

By adding a bias term, we allow the model to "shift" its solution to the left or right as needed. Lucky for us, PyTorch has a `Module` that implements a linear function like this, which we can access using `nn.Linear(d, C)`. This will create a linear layer with d inputs and C outputs, exactly what we want.

NOTE **The bias term is always a "$+\,\boldsymbol{b}$" on the side that does not interact with anything else. Because we almost *always* use a bias term, and they are annoying / cumbersome to *always* write out, they are often dropped and assumed to exist implicitly. We will do that as well in this book. Unless we state otherwise, assume the bias is implicitly there. If you see three weight matrices, assume there are three bias terms, one for each.**

NOTE **Modules are how PyTorch organizes the building blocks of modern neural network design. Modules have a `forward` function that takes in inputs and produces an output (we need to implement this if we make our own `Module`), and a `backward` function (PyTorch will take care of this for us, unless we have a special reason to intervene). Many standard Modules are provided in the `torch.nn` package, and we can build new ones out of tensors, `Parameter`, and `torch.nn.functional` objects. Modules may also contain other Modules, which is how we will build larger networks.**

2.1.4 Defining the Loss Function

So `nn.Linear` gives us our model $f(\boldsymbol{x})$, we still need to decide on a loss function ℓ. Again, PyTorch will make this is pretty simple for a standard regression problem. Say the ground truth is y, and our prediction is $\hat{y} = f(\boldsymbol{x})$. How do we quantify the difference between y and $\hat{y}$? We can just look at the absolute difference between them. That would be $\ell(y, \hat{y}) = |y - \hat{y}|$. Why *absolute* difference?

If we did not take the absolute value, and $\hat{y} < y$ would produce a positive loss, encouraging the model $f(\boldsymbol{x})$ to make its prediction smaller. But if $\hat{y} > y$ then $y - \hat{y}$ would produce a *negative* loss. If it feels intimidating to reason about this using symbols like y and $\hat{y}$, try plugging in some actual values. So if $\hat{y} = 100$ and $y = 1$, and we computed the loss as $y - \hat{y} = 1 - 100 = -99$, we would end up with a negative loss of -99! A negative loss would be confusing (what would that be, profit?) and actually encourage the network to make its predictions even larger when they are already too big. Our goal is that $\hat{y} = y$, but since the network will blindly march forward trying to *minimize* the loss, it will learn to make $\hat{y}$ unrealistically large in order to exploit a negative loss. This is why we need our loss function to always return zero or a positive value. Otherwise the loss will not make sense. Remember that the loss function is a penalty for errors, and negative penalties would mean encouragement.

Another option is to take the squared difference between y and $\hat{y}$. This would be $\ell(y, \hat{y}) = (y - \hat{y})^2$. This again results in a function which is zero only if $y = \hat{y}$, and only grows as y and $\hat{y}$ move farther away from each other.

Both of these options are pre-implemented for you in PyTorch. The former is called the $L1$ loss, as it corresponds to taking the 1-norm of the different (i.e., $\|y - \hat{y}\|_1$). The later is popularly known as the Mean Squared Error (MSE) loss, and is the most popular, so we will use it going forward.

Loss Function $\ell(y, \hat{y})$	PyTorch Module
$\|y - \hat{y}\|$	`torch.nn.L1Loss`
$(y - \hat{y})^2$	`torch.nn.MSELoss`

How to choose between two loss functions?

You have two different loss functions, L1 and MSE, that are both considered appropriate for a regression problem. How do you choose which one to use? A great question, that we could literally write a second book talking about. In general we won't take long detours about these nuanced differences between loss functions because it would just take up too much space, and for *most* circumstances picking any of the appropriate ones built into PyTorch will give you reasonable results.

What is important is that you know that L1 and MSE are both appropriate for *regression* problems. We will only learn about a few fundamental loss functions in this book, and just knowing which ones are appropriate for the type of problem (e.g., regression vs classification) will get you on the way to a reasonable solution.

Getting comfortable understanding the meaning behind the math will help you down the road to make these more nuanced choices. For this particular case, the MSE loss has the squared term which will make large differences grow larger (e.g., 100^2 will become 10,000) and L1 will keep differences the same (e.g., $|100| = 100$). So if the problem you are trying to solve is one where small differences are OK, but large ones are really bad - the MSE loss might be a better choice. If your problem is one where being off by 200 is twice as bad as being off by 100, then the L1 loss makes more sense. This alone is not a complete picture of why and when you would choose between these two options, but its a good way to think about it and make an initial choice.

PUTTING IT TOGETHER

We now have everything we need to create a linear regression. The `Dataset`, the `train_simple_network` function, a `loss_func` ℓ, and a `nn.Linear` model. The below code shows how we can quickly set this all up, and then pass it to our function to train a model.

```
in_features =1
out_features =1
model = nn.Linear(in_features, out_features)
loss_func = nn.MSELoss()

device = torch.device("cuda")
train_simple_network(model, loss_func, training_loader, device=device)
```

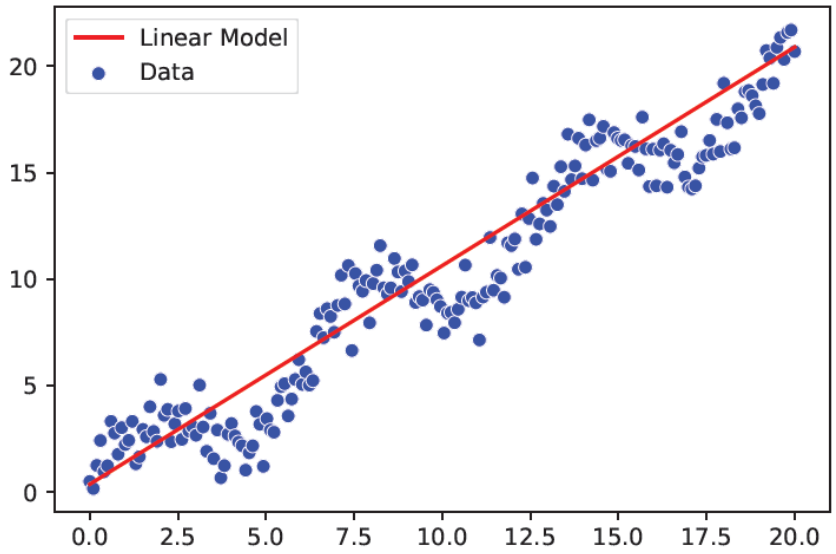
So did it work? Do we have a trained model? Thats easy enough to find out, especially since this is a one-dimensional problem. We can just plot our model's prediction for all the data. We will use the `with torch.no_grad():` context to get those predictions though. This context tells PyTorch that for any computation done within the scope of the `no_grad()` block, *do not calculate gradients*. We only want gradients to be computed during *training*. The gradient calculations take additional time, memory, and can cause bugs if we want to train the model more after performing a prediction. So good practice is to make sure we use the `no_grad()` block when making predictions to avoid all these issues. The following code block will use `no_grad()` to get the predictions.

```
with torch.no_grad():
    Y_pred = model(torch.tensor(X.reshape(-1,1), device=device,
       dtype=torch.float32)).cpu().numpy()
```

When you want to make predictions on new data, it is also called *inference*. This is common jargon amongst machine learning practitioners, and in particular within the deep learning community. This is because neural networks often require a GPU, so deploying a model is a bigger deal. Companies will often buy GPUs designed for *inference* that have less RAM to be more cost effective.

```
sns.scatterplot(x=X, y=y, color='blue', label='Data') #The data
sns.lineplot(x=X, y=Y_pred.ravel(), color='red', label='Linear Model') #What our model learned
```

```
[10]: <AxesSubplot:>
```



The above code plots the result of our network, and it did learn a good linear fit to the somewhat non-linear data. This is exactly what we asked of the model and the results look correct. You have also used all the mechanics and tools for building neural networks. Each of the components we have described, a loss function ℓ, a `Module` for specifying our network, and the training loop, can be swapped in and out in a piece meal fashion to build more powerful and complex models.

2.2 Building Our First Neural Network

Now we have learned how to create a training loop, and use gradient descent to modify a model so that it learns to solve a problem. This is the foundational framework we will use for all "learning" in this book. In order to train a neural network, we only need to replace the `model` object that we defined. The trick is knowing how to define these models. If we do a good job defining a neural network, it should be able to capture & model the non-linear parts of the data. For our toy example, that means getting the smaller oscillations and nut just the larger linear trend.

When we talk about neural networks and deep learning, we will usually be talking about *layers*. Layers are the building blocks we use to define our `model`, and most of the PyTorch `Module` classes are implementing different layers what have different purposes. The linear regression model we just built could be described as having an *input* layer (the input data itself), and a *linear layer* (`nn.Linear`) that made the predictions.

Our first neural network will be a simple *feed-forward fully-connected* neural network. Its called feed-forward because every output from one layer will flow directly into the next layer. So each layer will have one input and one output, and progress in a sequential fashion. It will be called fully-connected because each input of the network will have a connection to everything from the previous layer.

Lets start with what is called a *hidden* layer. The input $\boldsymbol{x}$ is consider the "input layer", and a $\mathbb{R}^C$ (a vector with C outputs for C predictions) dimensional output is called the "output-layer". In our

linear regression model, we essentially had only an input and output layer. You can think of "hidden" layers as anything sandwiched between input and output.

So how do we do that? Well, the easiest option would be to stick another matrix in there. So instead of

$$f(\boldsymbol{x}) = \boldsymbol{x}^{\top}\boldsymbol{W}^{d \times C}$$

we will put something like

$$f(\boldsymbol{x}) = \boldsymbol{x}^{\top}\boldsymbol{W}_{(h_1)}{}^{d \times n}\boldsymbol{W}_{(out)}{}^{n \times C}$$

Notice this new value n of the matrix dimension. Its a new hyper-parameter for us to tune and deal with. That means we get to decide what the value of n should be. It is called the "hidden layer size", or often the "number of neurons" in the first hidden layer. Why neurons?

If you draw out every intermediate output as a "node", and draw arrows representing weights, you get what could be described as a "network". This is shown in . The black lines connecting the input to hidden nodes corresponds to a `nn.Linear(3, 4)` layer, which is a matrix $W^{3 \times 4}$. Every column of that matrix corresponds to the inputs of one of the $n = 4$ "neurons" or outputs of that layer. Each row is a input's connections to each output. So if we wanted to know the strength of the connection from the 2nd input to the 4th output, we would index `W[1,3]`. In the same fashion, the lines from hidden to output of the above figure would be a `nn.Linear(4, 1)` layer.

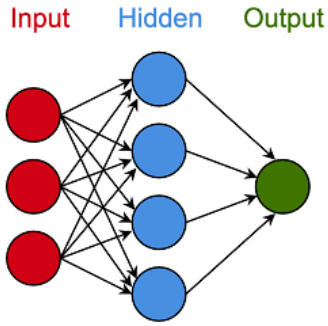


Figure 2.5: A simple feed-forward fully-connected network with one input layer of $d = 3$ inputs, a hidden layers with $n = 4$ neurons, and one output in the output layer. Connections feed directly into the next layer only, and each node in one layer is connected to every neuron in the preceding layer.

Notice also that all the arrows connecting nodes/neurons to each other only move from left to right. That is the feed-forward property we talked about. You should also note that each node in one layer is connected to every other node in the next layer. That is the *fully* connected property.

This network interpretation came in part as inspiration of how neurons in the brain work. A simple toy model of a neuron and it's connections is shown in . On the left is a neuron, where it has many *dendrites* what are connected to other neurons, which act as the inputs. The dendrites get electrical signals from other neurons firing, and carry them to the nucleus (center) of the neuron, which sums all those signals together. Finally, the neuron emits a new signal out from its *axon*.

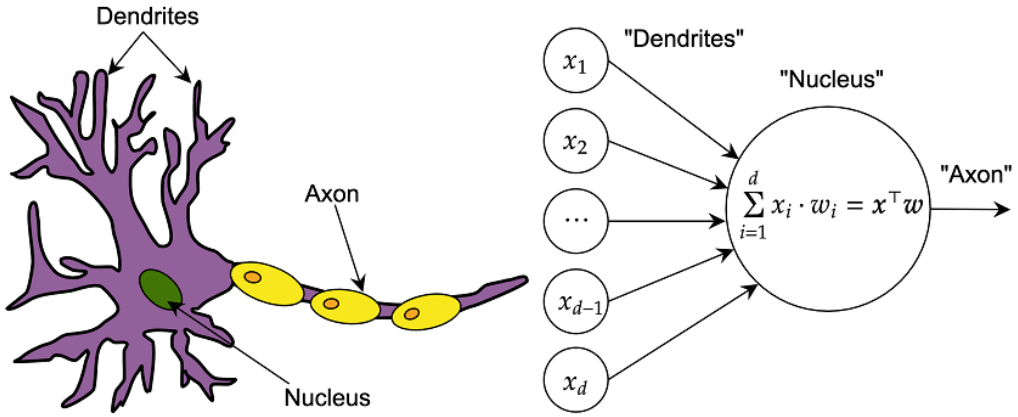


Figure 2.6: Simplified diagram of biological neuron connectivity. By analogy, dendrites are connections / weights between neurons, the axon caries the result of the neuron forward. This is a loose inspiration, and an over simplification of how real neurons work.

WARNING **Because neural networks where originally inspired by how neurons are connected and wired in the brain, the naming and analogy stuck. However, do not let it drag you too far in. The above description is a *very* simplified model of how neurons actually work. The functionality of a neural network is very far away from what little we do know about how the brain works in reality. You should take it as only a mild inspiration, not a literal analogy.**

According to these cool diagrams, and keeping with the idea of adding just one small change, we can insert two linear layers one after the other, we will have our first neural network. This is where the `nn.Sequential Module` comes into play. This is a `Module` that takes a list or sequence of Modules as its input. It will then run that sequence in a feed-forward fashion, using the output of one `Module` as the input to the next, until we have no more Modules. Figure 2.7 shows how we can do this for the toy network with 3 inputs and 4 hidden neurons.

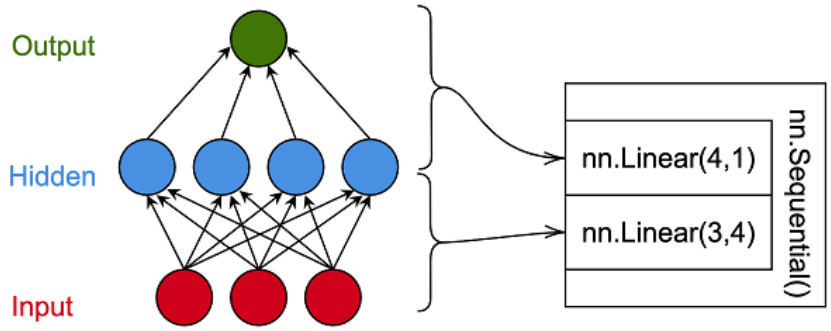


Figure 2.7: Converting the conceptual feed-forward full-connected network into a PyTorch `Module`. The `nn.Sequential` is a wrapper that takes in two `nn.Linear` layers. The first `nn.Linear` defines the mapping from input-to-hidden layer, and the second `nn.Linear` the hidden-to-output.

Rending this to practice is easy because all the other code we have written will still work. The following code creates a new simple `model` that is a sequence of two `nn.Linear` layers. Then we just pass the `model` into the same `train_simple_network` function and continue as before.

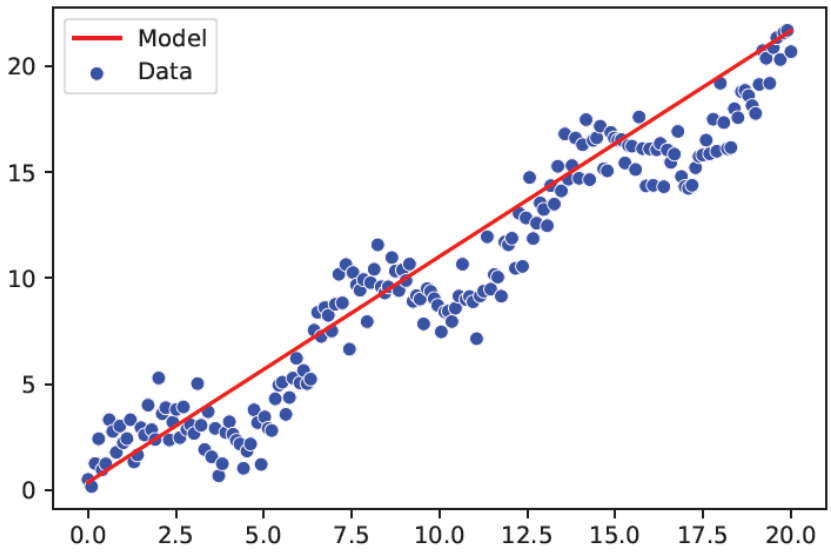
```
#Input "Layer" is implicitly the input
model = nn.Sequential(
    nn.Linear(1, 10), #Hidden Layer
    nn.Linear(10,1), #Output Layer
)

train_simple_network(model, loss_func, training_loader)
```

NOTE **The `nn.Sequential` class provides the easiest way to specify neural networks in PyTorch, and we will make use of it for *every* network in this book! So it is worth getting familiar with. Eventually we will build more complex networks that can not be described *entirely* as a feed-forward process. Even still, we will use the `nn.Sequential` class to help us organize the sub-components of a network that can be organized that way. This class is essentially your go-to-tool for organizing your models in PyTorch.**

Now we can perform inference with our new fancy neural network that has a hidden layer and see what we get. This uses the exact same inference code from before. The only difference is our `model` object that we've redesigned.

```
with torch.no_grad():
    Y_pred = model(torch.tensor(X.reshape(-1,1), dtype=torch.float32)).cpu().numpy()

sns.scatterplot(x=X, y=y, color='blue', label='Data') #The data
sns.lineplot(x=X, y=Y_pred.ravel(), color='red', label='Model') #What our model learned
```

```
[12]: <AxesSubplot:>
```



So what gives? We made our `model` $f(\cdot)$ more sophisticated, training took longer, and it did about the same/maybe worse? A little bit of linear algebra will answer why this happened. Recall we defined

$$f(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{W}^{d,n}_{(h_1)} \boldsymbol{W}^{n,C}_{(\text{out})}$$

Here $\boldsymbol{W}_{(h_1)}{}^{d \times n}$ is our hidden layer. Since we have one feature $d = 1$, and we have $n = 10$ hidden units. $\boldsymbol{W}_{(out)}{}^{n \times C}$ is our output layer, where we still have the $n = 10$ hidden units from the previous layer, and $C = 1$ total outputs.

But, we can simplify the two weight matrices. If we have a matrix with shape (a, b) and a second matrix with shape (b, c) that we multiply together, we will get a new matrix with shape (a, c). Which means

$$\boldsymbol{W}^{d \times n}_{(h_1)} \boldsymbol{W}^{n \times C}_{(\text{out})} = \tilde{\boldsymbol{W}}_{d,c}$$

and therefore

$$f(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{W}^{d \times n}_{(h_1)} \boldsymbol{W}^{n \times C}_{(\text{out})} = \boldsymbol{x}^\top \tilde{\boldsymbol{W}}_{d,c}$$

Thats equivalent to the original linear model we started with. What this shows is that *adding any number of sequential linear layers will be equivalent to having used just one linear layer*. Linear operations beget linear operations, and are usually redundant. Placing multiple linear layers one-after-the-other is a common error I see in novice/junior practitioner's code.

2.2.1 Adding Non-Linearities

In order to get any kind of benefit, we will need to introduce some kind of *non-linearity* between every step. By inserting a non-linear function after every linear operation, we allow the network to build up more complex functions. We will call these non-linear functions that are used in this way *activation functions*. The analogy from biology is that a neuron sums together all of its inputs linearly, and eventually "fires" or "activates", sending a signal to other neurons in the brain.

Now the question is, what should we use as our activation functions? The first two we will look at are the *sigmoid* ($\sigma(\cdot)$) and the *hyperbolic tangent* ($\tanh(\cdot)$) functions, which were two of the originally chosen, and still widely used, activation functions.

The $\tanh$ function is an historically popular non-linearity. It maps everything into the range

$$\text{The tanh: } \tanh(x) = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

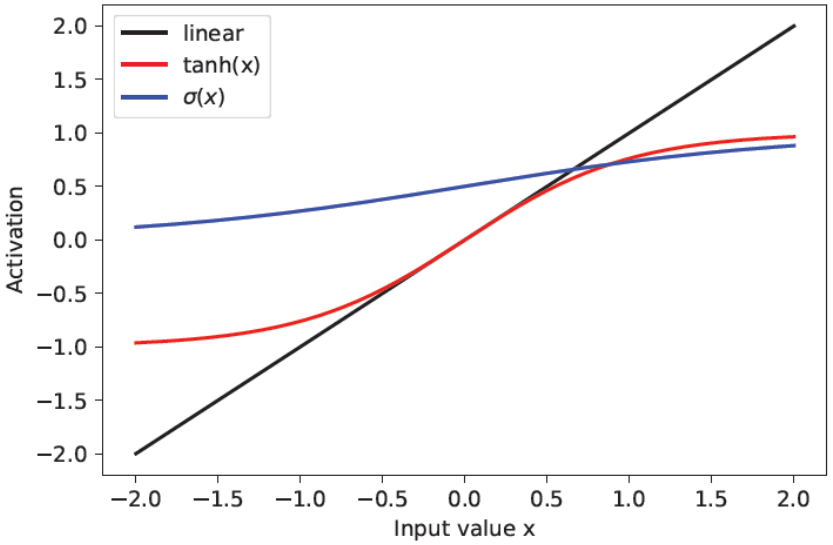
The sigmoid is the historical non-linearity, and is where the notation σ is most often used / comes from. It maps everything into the range $[0, 1]$

$$\sigma(x) = \frac{e^x}{e^x + 1}$$

Lets quickly plot what these look like. The input will be on the x-axis, and the activation will be on the y-axis.

```
activation_input = np.linspace(-2,2, num=200)
tanh_activation = np.tanh(activation_input)
sigmoid_activation = np.exp(activation_input)/(np.exp(activation_input)+1)
sns.lineplot(x=activation_input, y=activation_input, color='black', label="linear")
sns.lineplot(x=activation_input, y=tanh_activation, color='red', label="tanh(x)")
ax = sns.lineplot(x=activation_input, y=sigmoid_activation, color='blue', label="$\sigma(x)$")
ax.set_xlabel('Input value x')
ax.set_ylabel('Activation')
```

```
[13]: Text(0, 0.5, 'Activation')
```



As promised, the sigmoid activation ($\sigma(x)$) maps everything to a minimum of 0, and a maximum of 1. The $\tanh$ function goes from -1 to 1. Notice how there is a range of the input, around 0, where the $\tanh$ function *looks* linear, but then diverges away? Thats perfectly ok. In fact, that can even be desirable. The important thing from a learning perspective is that neither $\tanh(\cdot)$ or $\sigma(\cdot)$ can be *perfectly* fit by a linear function.

We will talk about the properties of these functions more in future chapters. For now, lets use the $\tanh$ function. So we will define a new model that matches the following:

$$f(\boldsymbol{x}) = \tanh\left(\boldsymbol{x}^{\top}\boldsymbol{W}^{(h_1)}_{d\times n}\right)\boldsymbol{W}^{(\text{out})}_{n\times C}$$

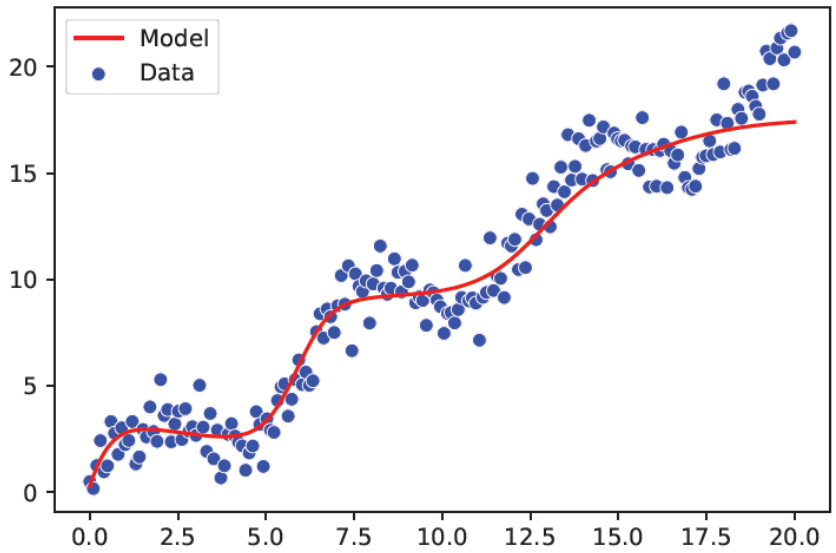
Instead of stacking `nn.Linear` layer directly after another `nn.Linear` layer, we will call the $\tanh$ function after the first linear layer. When using PyTorch, we generally want to end our network with a `nn.Linear` layer. So for this model, we have two `nn.Linear` layers, so we will use $2-1=1$ activations. This is as simple as adding a `nn.Tanh` node into our sequential network specification, which PyTorch has built in. Lets see what happens when we train this new `model`.

```
model = nn.Sequential(
    nn.Linear(1, 10),#hidden layer
    nn.Tanh(),#activation
    nn.Linear(10,1),#output layer
```

```
)

train_simple_network(model, loss_func, training_loader, epochs=200)
```

```
with torch.no_grad():
    Y_pred = model(torch.tensor(X.reshape(-1,1), dtype=torch.float32)).cpu().numpy()

sns.scatterplot(x=X, y=y, color='blue', label='Data') #The data
sns.lineplot(x=X, y=Y_pred.ravel(), color='red', label='Model') #What our model learned
```

```
[15]: <AxesSubplot:>
```



Its doing much better. We can see that the network is now learning a non-linear function, with bends that move and adapt to match the data's behavior. Its not perfect though, especially for larger values of the input x on the right side of the plot. We also had to train for many more `epochs` than we did previously. Thats also pretty common. This is part of why we use GPUs so much in deep learning, because we have larger models means each update requires more computation, and the larger models need more updates to converge, resulting in longer training times.

In general, the more complex a function that needs to be learned, we should expect to have to perform more rounds of training, and maybe even get more data too. However, there are still many ways to improve the quality and rate at which our neural networks learn from the data. These are all things we will review in more detail in later sections of the book. For now, our goal is to learn the basics.

2.3 Classification Problems

You have now built your first neural network by extending a linear regression model, but what about classification problems? In this situation you have C different *classes* that an input might belong to. For example, a car could be an SUV, sedan, coupe, or truck as four different classes.

As you may have guessed, this will involve having an output layer that looks like `nn.Linear(n, C)`, where again n was the number of hidden units in the previous layer and C is the number of classes / outputs. It would be difficult for us to make C predictions if we had less than C outputs.

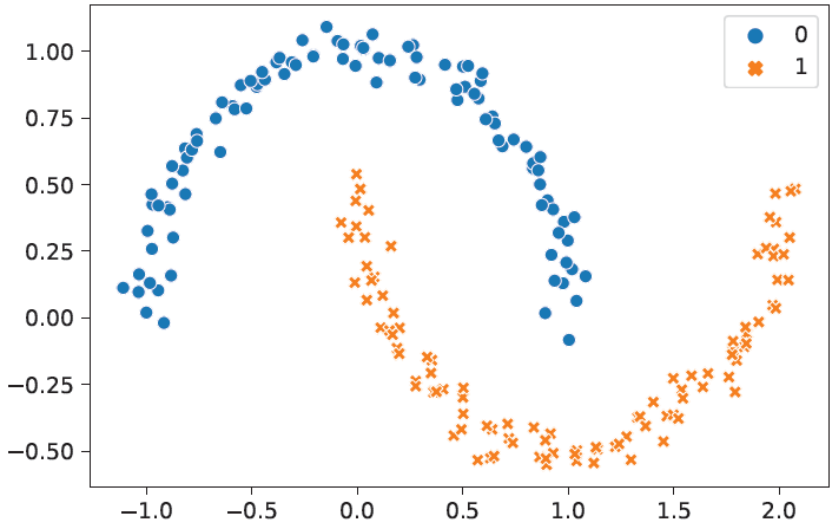
Similar to how we can walk from linear regression to a non-linear regression neural network, we can make the same walk from *logistic regression* to a non-linear classification network. If you do not remember logistic regression is a popular algorithm for classification problems that finds a linear solution to try and separate C classes.

2.3.1 Classification Toy Problem

Before we can build a logistic model, first we need a dataset. Getting our data loaded and into a `Dataset` object is always going to be our first and most important step to using PyTorch. For this example we will use the `make_moons` class from scikit-learn.

```
from sklearn.datasets import make_moons
X, y = make_moons(n_samples=200, noise=0.05)
sns.scatterplot(x=X[:,0], y=X[:,1], hue=y, style=y)
```

```
[16]: <AxesSubplot:>
```



The moons dataset has $d = 2$ input features now, and we can see in the scatter plot above the two different clases as circles and crosses. This is a good toy problem for us since a linear classification model can do an *ok* job separating most of the circles from the crosses, but won't be able to *perfectly* solve the problem.

To make our lives easier, we will use the built in `TensorDataset` object to wrap our current data. This only works if we can fit all of our data into RAM. But if you can, this is the easiest way to prepare data for. You can use your favorite Pandas or NumPy methods to load in the data and then start modeling.

We did make one important change though. Our vector of labels y is now a `torch.long` rather than a `torch.float32`. Why is this? Because the labels are now classes, which start from 0 and go up to

$C-1$ to represent C different classes. There is no 0.25 class, only integers are allowed! For this reason, we use the long data type (a 64-bit integer) rather than a floating point value, since we only need to concern ourselves with integers. For example, if our classes were "cat", "bird", and "car", we would use "0, 1, 2" to represent the the three classes. You may recognize this as being very close to a "one-hot encoding", where each class is given it's own dimension. PyTorch is going to do that last step for us under-the-hood in order to avoid wastefully representing all the non-present classes that one-hot encoding does.

Why not a `torch.int` or `torch.int32` type, to match the 32-bit floats? Because PyTorch is written to work with 64-bit integers, and most of the code base has hard-coded to 64-bit integers. It was a design choice made early on, and we must abide by it to use PyTorch. Almost every `Module` PyTorch provides that expects integer inputs will throw an error if you give it `torch.int32` as the `dtype` of a tensor instead of `torch.int64`.

```
classification_dataset = torch.utils.data.TensorDataset(torch.tensor(X, dtype=torch.float32),
                                                         torch.tensor(y, dtype=torch.long))
training_loader = DataLoader(classification_dataset)
```

Now we define a linear classification model just like we did previously. In this case, we have two features, and we have two outputs (one for each class). So our model will be *slightly* bigger. Notice that even though the *target* vector y is going to be represented as a single integer, the network will have C explicit outputs. This is because the labels are *absolute*, there is only one true class per data point. The network however must always consider all C classes as potential options, and so must make a prediction for each class separately.

```
in_features =2
out_features =2
model = nn.Linear(in_features, out_features)
```

2.3.2 Classification Loss Function

The big question is, what do we use as our loss function? This was an easy question to answer when we did a regression problem. We had two inputs, and they were both floating point values, so we could just subtract them to determine how far away the two values are. This is different though. Now our prediction $\hat{y} \in \mathbb{R}^C$, because we need to have a prediction for each of our C different classes. But our labels are one value of a set of integers, $y \in \{0, 1, \ldots, C-1\}$. If we can define a loss function $\ell(\hat{y}, y)$ that takes in a vector of predictions $\hat{y} \in \mathbb{R}^C$ and compare it to a correct class y, we can re-use everything from the training loop in and our previously defined neural network. Luckily this function already exists for us, but its worth talking about in detail because of how foundational it is to everything we will do through this book. We need two components: the softmax function and the cross-entropy, which combined are often just called the *cross-entropy loss*.

SOFTMAX

First, we intuitively want the dimension in $\hat{y}$ that has the largest value to correspond to the correct class label y (same as if we used `np.argmax`). If we wrote this in math, that would mean we want:

$$y = \underbrace{\underset{i}{\text{argmax}}\hat{y}_i}_{\text{np.argmax}(\hat{\boldsymbol{y}})}$$

The other thing we want is that our predictions should be a sensible probability. Why? Consider that the correct class is $y = k$, and that we succeed and have $\hat{y}_k$ is the largest value. How "right" or "wrong" is this? What if $\hat{y}_k - \hat{y}_j = 0.00001$? The difference is very small, and we want a way to tell the model that it should make the difference larger.

If we made $\hat{y}$ into probabilities, then they have to sum up to 1. That means:

$$\underbrace{\sum_{i=0}^{C-1} \hat{y}_i}_{\text{np.sum}(\hat{\boldsymbol{y}})} = 1$$

This way we know a model is confident in its prediction when $\hat{y}_k = 1$, and all other values of $j \neq k$ result in $\hat{y}_j = 0$. If the model was less confident, we might see $\hat{y}_k = 0.9$, and if it was completely wrong, $\hat{y}_k = 0$. The constraint that $\hat{y}$ sums to one makes it easy for us to interpret the results.

But how do we ensure this? The values that we get from the last `nn.Linear` layer could be anything, especially when we first start training and have not had a chance to teach the model about what is correct. The *Soft Maximum* (or "softmax") function is what we will use to fix this. The softmax function converts everything to a non-negative number, and ensure that the values sum up to 1.0. The index k with the largest value will have the largest value afterwards as well, *even if it was negative*, so long as every other index was an even smaller number. But smaller values will also receive smaller but non-zero values too. So it will give every value in $0, 1, \ldots, C-1$ a value in the range $[0, 1]$, such that they all sum up to one.

We can write this out as:

The probability of the i'th item is	the score of the i'th item
divided by the score of every item added together.	

$$\text{sm}(\boldsymbol{x})_i = \frac{\exp(x_i)}{\sum_{j=1}^{d} \exp(x_j)}$$

Lets quickly look at the results of calling softmax on two different vectors.

$$\mathrm{sm}(\boldsymbol{x} = [\ 3,\ \underset{\text{Largest}}{4}\ ,\ 1]) = [0.259,\ \underset{\text{Largest}}{0.705},\ 0.036]$$

$$\mathrm{sm}(\boldsymbol{x} = [-3,\ -4,\ \underset{\text{Largest}}{-1}]) = [0.114,\ 0.042,\ \underset{\text{Largest}}{0.844}]$$

In the first case, 4 is the largest value, and so it receives the largest normalized score of 0.705. The second case -1 is the largest value, and it receives a score of 0.844. Why did second case result in a larger score even though -1 is smaller than 4? It is because softmax is relative, 4 is only one larger than 3 so its not too far away from the others. The second case -1 is three larger than -4, so its a bigger difference, and so it receives a bigger score.

Why is it called "softmax"?

Before we continue on, I find it helpful to explain \emph{why} the softmax function is called "softmax". It is because we can use this score to compute a "soft" maximum, where every value contributes a portion of the answer. If we take the dot product between the softmax scores and the original values, it will be approximately equal to the maximum value. Lets look at how that happens.

$\mathrm{sm}\ \boldsymbol{x} = [3, 4, 1]) = [0.259, 0.705, 0.036]$

$\mathrm{sm}\ \boldsymbol{x}^\top\ \boldsymbol{x} = 0.259 \cdot 3 + 0.705 \cdot 4 + 0.036 \cdot 1 = \mathbf{3.633}$

$\max_i x_i = \mathbf{4}$

–

$\mathrm{sm}\ (\boldsymbol{x} = [-3, -4, -1]) = [0.114, 0.042, 0.844]$

$\mathrm{sm}\ (\boldsymbol{x})^\top\ \boldsymbol{x} = 0.114 \cdot -3 + 0.042 \cdot -4 + 0.844 \cdot \text{-}1 = \mathbf{-1.354}$

$\max_i x_i = -1$

The value of $\mathrm{sm}\ (\boldsymbol{x})^\top\ \boldsymbol{x}$ is going to be approximately equal (≈) to finding the maximum value of $\boldsymbol{x}$. Because every value contributes to at least a portion of the answer, it is also the case that $\mathrm{sm}\ (\boldsymbol{x})^\top\ \boldsymbol{x} \leq \max_i x_i$. So it can get close, but only becomes equal to the maximum when *all* values are the same.

CROSS-ENTROPY

With this function in hand, we have one of the two tools we need to define a good loss function for classification problems. The second tool we need is called the *cross entropy* loss. If we had two probability distributions $\boldsymbol{p}$ and $\boldsymbol{q}$, the cross entropy between these two distributions is:

$$\ell(\boldsymbol{p}, \boldsymbol{q}) = -\sum_{i=1}^{d} \boldsymbol{p}_i \cdot \log(\boldsymbol{q}_i)$$

Why cross entropy? Its a statistical tool that tells us how much extra information it is going to take for us to encode information if we used the distribution defined by $\boldsymbol{q}$ when the data *actually*

follows the distribution ***p***. This has glossed over some of the precision of what the Cross-Entropy function is doing, but gives you an intuitive idea at a high level. Cross entropy boils down to telling us how different two distributions are, allowing us to tackle problems like how much turkey vs chicken[1] we should order.

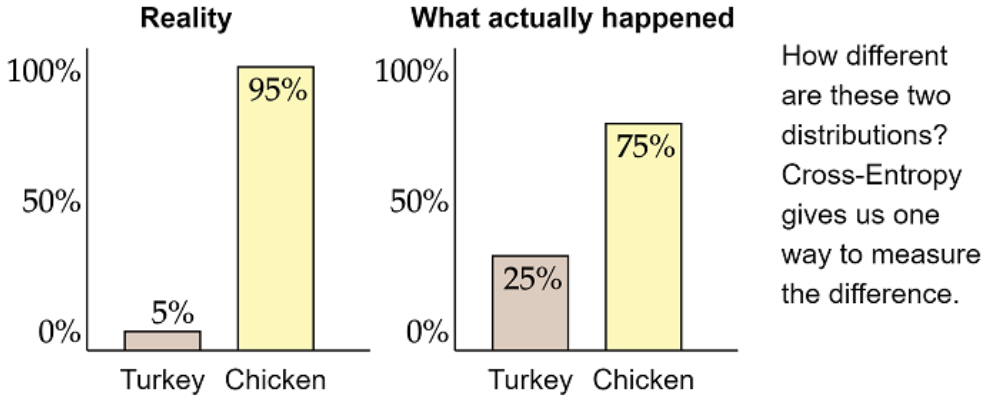


Figure 2.8: There are two different distributions. The real distribution on the left, and our prediction on the right. In order to learn, we need a loss function to tell us precisely how different these are. Cross entropy solves that problem. In most cases, our label defines "reality" as being 100\% one class and 0\% for all the others.

Think of it as trying to minimize cost. Imagine you are ordering lunch for a group of people and you expect 70% to eat chicken and 30% to eat turkey. Thats our predicted distribution ***q***. In reality, 5% want chicken and 95% want turkey. Thats the true distribution ***p***. In this scenario, we *think* we want to order more chicken (and bargain down the price per chicken based on our bulk order). But if we do order all that chicken we will incur a very high cost, because we will be short on turkey (and have wasted/unused chicken). Cross-Entropy is just a way to quantify how different these distributions are so that we can order the right amount of each thing.

Now, with these two tool combined, we arrive at a simple loss function and approach. We first apply the softmax function ($\text{sm}(x)$), followed by computing the cross-entropy. If $\hat{y}$ is our vector output from the network, and y is the correct class index, this simplifies down to:

The loss between the prediction $\hat{y}$ (a.k.a. logits) and the correct classy y is the negative log (makes is so that "good" values are smaller and "bad" are larger) of the predicted probability of the correct class.

$$\ell(\hat{y}, y) = -\log\left(\text{sm}(\hat{y})_y\right)$$

This may seem a bit mysterious, and thats *ok*. This result comes from simplifying the equations and derivations are not the point of this book. The reason we worked through the details that we did is that the softmax and Cross-Entropy functions are *ubiquitous* in deep learning research

today, and some extra effort now to understand what these functions do will make our lives easier later on in the book. The important thing is to feel like you know what a softmax function does (normalized inputs into probabilities) and that it can be used with Cross-Entropy to quantify how different two distributions (arrays of probabilities) are.

There is a statistical interpretation of how we arrived at this loss function, and we use it because it has a strong statistical grounding and interpretation. It assures us that we will be able to interpret the results as being a probability distribution. For the case of a linear model, it results in the well known algorithm *logistic regression*.

Using this combination of softmax followed by Cross-Entropy is so standard and well known, PyTorch integrates them into a single loss function `CrossEntropyLoss`, which performs both steps for us. This is good, because implementing the softmax and Cross-Entropy functions manually can lead to some tricky numerical stability issues, and so a good implementation is not as direct as you might think.

TRAINING A CLASSIFICATION NETWORK

But now that we know that, we can more forward and train a `model`. This was a lot of text to explain how we arrive at the single line of code below to define our loss function. Now we can train our model, and see how well it performs.

```
loss_func = nn.CrossEntropyLoss()
train_simple_network(model, loss_func, training_loader, epochs=50)
```

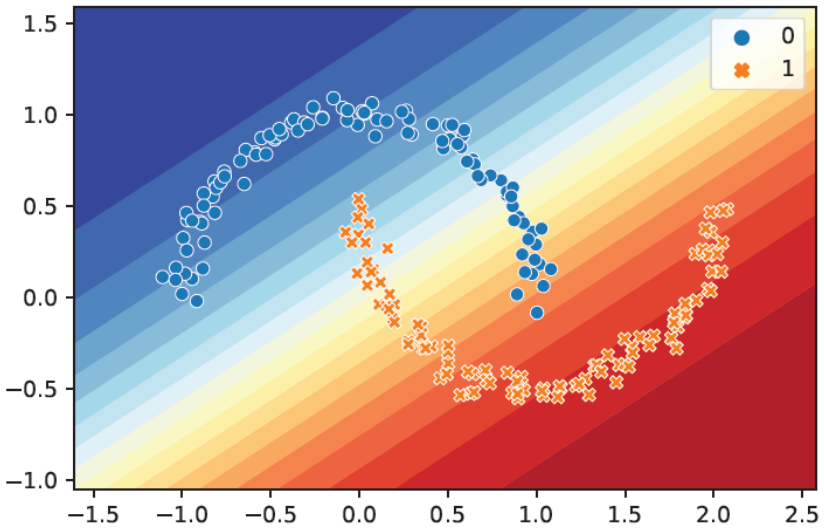
With our model trained, lets visualize the results. Since this is a 2-d function, its a little more complicated than our previous regression case. We will use a kind of contour-plot to show the decision surface of our algorithm. Where dark blue represents the first class, dark red the second class, and the color will lighten/transition as the model's confidence decreases and increases.

Notice that in the below function, we call the PyTorch function `F.softmax` to perform the conversation from raw outputs into a actually probability distributions. It is common jargon to call the value that goes into the softmax the "logits", and the outputs $\hat{y}$ the probabilities. We will try to avoid using the term logits too much in this book, but you should be familiar with it. It comes up often when people are trying to discuss the nitty gritty details of an implementation or approach.

```
def visualize2DSoftmax(X, y, model, title=None):
    x_min = np.min(X[:,0])-0.5
    x_max = np.max(X[:,0])+0.5
    y_min = np.min(X[:,1])-0.5
    y_max = np.max(X[:,1])+0.5
    xv, yv = np.meshgrid(np.linspace(x_min, x_max, num=20), np.linspace(y_min, y_max, num=20),
       indexing='ij')
    xy_v = np.hstack((xv.reshape(-1,1), yv.reshape(-1,1)))
    with torch.no_grad():
        logits = model(torch.tensor(xy_v, dtype=torch.float32))
        y_hat = F.softmax(logits, dim=1).numpy()

    cs = plt.contourf(xv, yv, y_hat[:,0].reshape(20,20), levels=np.linspace(0,1,num=20),
       cmap=plt.cm.RdYlBu)
    sns.scatterplot(x=X[:,0], y=X[:,1], hue=y, style=y, ax=cs.ax)
    if title is not None:
        cs.ax.set_title(title)

visualize2DSoftmax(X, y, model)
```

```
/home/edraff/anaconda3/lib/python3.7/site-packages/ipykernel\_launcher.py:13:
```

```
MatplotlibDeprecationWarning:
The ax attribute was deprecated in Matplotlib 3.3 and will be removed two minor
releases later.
  del sys.path[0]
```



Above we can now see the results of our model on this data. Overall a decent job, most of the blue circles are in the blue region, and the red crosses in the red region. There is a middle ground where errors are being made, because our problem can not be fully solved with a linear model.

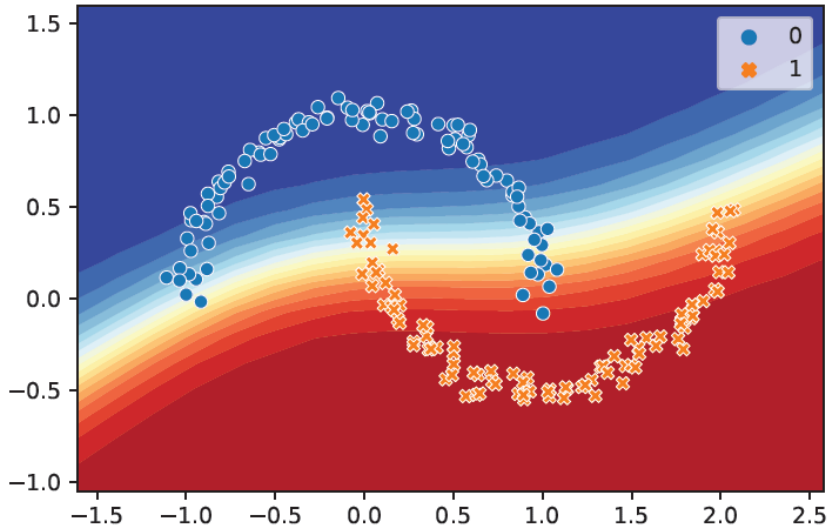
Now we do the same as we did with our regression problem, we will add in a hidden layer to increase the complexity of the neural network. In this case, we will go ahead and add two hidden layers, just to show how easy it is. I've arbitrarily selected $n = 30$ hidden units for both hidden layers.

```
model = nn.Sequential(
    nn.Linear(2, 30),
    nn.Tanh(),
    nn.Linear(30, 30),
    nn.Tanh(),
    nn.Linear(30,2),
)
train_simple_network(model, loss_func, training_loader, epochs=250)
```

You should notice that these models are starting to take some time to train. 250 `epochs` required 36 seconds when I ran this. The results appear to be worth it though, if we look at a plot of our data we see that the model has higher confidence for the regions that are unambiguously circles or crosses. You can also see that the threshold is starting to bend and curve, as the neural network learns a non-linear separation between the two classes.

```
visualize2DSoftmax(X, y, model)
```

```
/home/edraff/anaconda3/lib/python3.7/site-packages/ipykernel\_launcher.py:13:
MatplotlibDeprecationWarning:
The ax attribute was deprecated in Matplotlib 3.3 and will be removed two minor
releases later.
  del sys.path[0]
```



2.4 Better Training Code

We've now successfully trained fully connected networks for regression and classification problems. There is still a lot of room for us to improve our approaches. In particular, we have been training and evaluating visually on the same data. *This is not ok*. We also have another issue when dealing with classification problems: minimizing the cross-entropy loss is not really our goal. Our goal was to minimize errors, but we can't define "errors" in a differentiable way that will work with PyTorch, so we had to use cross-entropy as a proxy metric instead. So reporting the loss after every epoch for a classification problem is not as helpful, because it was not our true goal.

We are going to talk about a number of changes we will make to our training code, that will give us much more robust tools. Like all good machine learning practitioners, we be making and using a *training* set and a *testing* set. We will also add to evaluate other metrics that we *do* care about, so that we can track performance as we train.

2.4.1 Custom Metrics

As mentioned, the metrics we care about (e.g., accuracy) may not be the same as the loss we used to train our model (e.g., cross-entropy). There are many ways that these could *not* match together perfectly. This is because a loss function must have the property of being *differentiable* , and most of the time our true goal does not have this property. So we will often have two sets of scores: the metrics by which the developers and humans understand the problem, and the loss function that lets the network understand the problem.

To help with this, we will modify our code so that we can pass in functions to compute different metrics from the labels and predicted values. We will also want to know about about how these metrics vary across our training, and validation datasets. So we will record multiple versions, one for each type of data set.

To make our lives easier, we will make it so our code will work well with most of the metrics provided by the Scikit-Learn library. To do this, lets assume we have an array `y_true` that

contains the correct output label for every data point. With it we will need another array `y_pred` that contains the prediction of our model. If we are doing regression, each prediction is a scalar $\hat{y}$. If classification, each prediction is a vector $\hat{y}$.

We need some way for the user (thats you, the reader) to specify which functions they want to evaluate, and a place to store the results. For the score functions, lets use a dictionary `score_funcs` that takes the name of the metric as the key, and a function reference as the value. So that would look like

```
score_funcs={'Acc':accuracy_score, 'F1': f1_score}
```

if we used the functions provided by scikit-learn's `metics` class (see https://scikit-learn.org/stable/modules/model_evaluation.html#common-cases-predefined-values). This way we can specify as many custom metrics as we want, so long as we implement a function `score_func(y_ture, y_pred)`. Then we just need a place to store the computed scores. After each epoch of the loop, we can use another dictionary `results` that maps strings as keys to a `list` of results. Well use a list so that we have one score for each epoch.

```
results[prefix+" loss"].append(np.mean(running_loss))
for name, score_func in score_funcs.items():
    results[prefix+" "+name].append(score_func(y_true, y_pred))
```

If we justed used the `name` of each score function, we would not be able to differentiate between the score on the training set and the testing set. This is important, because if there is a wide gap that could be an indicator of over-fitting, and a small gap could indicate under-fitting. So we will use a `prefix` to distinguish between `"train"` and `"test"` scores.

If we are being proper in our evaluations, we should use only the validation/test performance to make adjustments and changes to our code, hyper-parameters, network architecture, etc. This is another reason we need to make *sure* that we are distinguishing between training and validation performance. You should *never* use training performance to make a decision about how well a model is doing.

2.4.2 Training and Testing Passes

So we are going to modify our training function to better support real life work. That includes supporting a training epoch where we alter the model weights, and a testing epoch where we only record our performance. Its important that we make the testing epoch *never* adjust the weights of the model.

Performing one epoch of training or evaluation actually requires a *lot* of different inputs.

1. `model`: The PyTorch `Module` to run for one epoch that represents our model $f(\cdot)$.
4. `optimizer`: the object that will update the weights of the network, and should only be used if we are performing a training epoch.
5. `data_loader`: The `DataLoader` object that returns tuples of (input, label) pairs.
6. `loss_func`: The loss function $\ell(\cdot,\cdot)$ that takes in two arguments, the `model` outputs ($\hat{\boldsymbol{y}} = f(\boldsymbol{x})$ and the `labels` (y), and returns a loss to use for training
7. `device`: The compute location to perform training
8. `results`: A dictionary of strings to lists for storing results, as described above.
9. `score_funcs`: A dictionary of scoring functions to use to evaluate the performance of the `model`, as we described above.
10. `prefix`: A string to pre-fix to any scores placed into the `results` dictionary.

Last, because neural networks can take a while to train, lets include an optional argument `desc` to provide a descriptive string for a progress bar. So that will give us all the inputs we need for a function that processes one epoch, which we could give the below signature.

```
def run_epoch(model, optimizer, data_loader, loss_func, device,
              results, score_funcs, prefix="", desc=None):
```

At the start of this function, we will need to allocate space to store results such as the losses, predictions, and the time we started computing at.

```
running_loss = []
y_true = []
y_pred = []
start = time.time()
```

The training loop will look almost identical to the one we have used this far. The only thing we need to change is *if* we use the optimizer or not. We can check this by looking at the `model.training` flag, which will be `True` if our model is in training mode (`model = model.train()`) or `False` if its in evaluation/inference mode (`model = model.eval()`). So we can just wrap the `backward()` call on the loss and `optimizer` calls into an `if` statement at the end of each loop.

```
if model.training:
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Last thing we need is to store the `labels` and predictions `y_hat` into `y_true` and `y_pred` respectively. This can be done by calling `.detach().cpu().numpy()` to convert both from PyTorch tensors into NumPy arrays, and then we simply `extend` the lists of all labels by the current labels we are processing.

```
if len(score_funcs) >0:
    #moving labels & predictions back to CPU for later
    labels = labels.detach().cpu().numpy()
    y_hat = y_hat.detach().cpu().numpy()
    #add to predictions so far
    y_true.extend(labels.tolist())
    y_pred.extend(y_hat.tolist())
```

2.4.3 Saving Checkpoints

The last modification we will make is the ability to save a simple checkpoint of the most recently completed epoch. In PyTorch, we are given a `torch.load` and `torch.save` function that can be used for this purpose. While there is more than one way to use these methods, we recommend using the dictionary style approach that you will see below. This lets us save the model, the optimizer state, and other information as well, all in one object.

```
torch.save({
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'results' : results
    }, checkpoint_file)
```

The second argument `checkpoint_file` is simply a path to where we should save the file. We can put any picklable object into this dictionary to be saved. In our case well denote the number of training epochs, the `model` state (thats the weights /parameters Θ), and any state used my the `optimizer` as well.

We need to be able to save our model so that when we are ready to use it, we do not have to train it from scratch again. Saving after every epoch is a better idea though, especially when you start to train networks that can take weeks to complete. Sometimes, we may find that our code fails after many epochs, or a power failure interrupts our job. By saving the model after every epoch, we can resume our training from the last epoch, rather than having to start from scratch.

2.4.4 Putting It All Together

Now we have everything to build a better function for training our neural networks. Note that its not just the networks we have talked about (e.g., fully-connected), but almost all networks we will discuss in this book. The signature for this new function will look like:

```
def train_simple_network(model, loss_func, train_loader,
    val_loader=None, score_funcs=None, epochs=50,
    device="cpu", checkpoint_file=None):
```

Where the arguments are 1. `model`: The PyTorch `Module` to run for one epoch that represents our model $f(\cdot)$. 2. `loss_func`: The loss function $\ell(\cdot,\cdot)$ that takes in two arguments, the `model` outputs ($\hat{\boldsymbol{y}} = f(\boldsymbol{x})$ and the `labels` (y), and returns a loss to use for training 3. `train_loader`: The `DataLoader` object that returns tuples of (input, label) pairs used for training the model. 4. `val_loader`: The `DataLoader` object that returns tuples of (input, label) pairs used for evaluating the model. 5. `score_funcs`: A dictionary of scoring functions to use to evaluate the performance of the `model`, as we described above. 6. `device`: The compute location to perform training 8. `checkpoint_file`: A string indicating the location to save checkpoints of the model to disk.

The gist of this new function is below, and you can find the full version in your `idlmam.py` file that comes with the book.

```
#Perform some book keeping / setup
#Prepare optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
#Place the model on the correct compute resource (CPU or GPU)
model.to(device)
for epoch in tqdm(range(epochs), desc="Epoch"):
    model = model.train()#Put our model in training mode
    total_train_time += run_epoch(model, optimizer, train_loader, loss_func, device, results,
       score_funcs, prefix="train", desc="Training")

    results["total time"].append( total_train_time )
    results["epoch"].append( epoch )

    if val_loader is not None:
        model = model.eval()
        with torch.no_grad():
            run_epoch(model, optimizer, val_loader, loss_func, device, results, score_funcs,
       prefix="val", desc="Testing")
    #Save a checkpoint if checkpoint_file is not None
```

We use the `run_epoch` function to perform a training step after putting the model into the correct mode, and that function itself records the results from training. Then if our `val_loader` is given, we switch to `model.eval()` mode and enter `with torch.no_grad()` context so that we do not alter the model in any way, and can just examine it's performance on the held out data. We use the prefixes `"train"` and `"val"` for the results from the train and testing runs respectively.

Finally, we will have this new training function convert the results into a Pandas dataframe, which will make it easy for us to acces and view them later.

```
return pd.DataFrame.from_dict(results)
```

With this new and improved code, lets re-train our model on the moons dataset. Since accuracy is what we really care about, we will import the accuracy metrics from Scikit-Learn. We will also include the F1 score metrics, simply to demonstrate how the code can handle two different metrics at the same time.

```
from sklearn.metrics import accuracy_score
from sklearn.metrics import f1_score
```

We also want to do a better job at evaluation, and include a validation set. Since the moons data is synthetic, we can easily just create a new set of data for validation. Rather than performing 200 epochs of training like before, lets also generate a larger training set.

```
X_train, y_train = make_moons(n_samples=8000, noise=0.4)
X_test, y_test = make_moons(n_samples=200, noise=0.4)
train_dataset = TensorDataset(torch.tensor(X_train, dtype=torch.float32), torch.tensor(y_train,
       dtype=torch.long))
test_dataset = TensorDataset(torch.tensor(X_test, dtype=torch.float32), torch.tensor(y_test,
       dtype=torch.long))
training_loader = DataLoader(train_dataset, shuffle=True)
testing_loader = DataLoader(test_dataset)
```

Now we have everything we need to train our model again. We will use `"model.pt"` as the location to save our model's results. All it requires is declaring a new `model` object and calling our new `train_simple_network` function.

```
model = nn.Sequential(
    nn.Linear(2, 30),
    nn.Tanh(),
    nn.Linear(30, 30),
    nn.Tanh(),
    nn.Linear(30,2),
)
results_pd = train_simple_network(model, loss_func, training_loader, epochs=5,
       val_loader=testing_loader, checkpoint_file='model.pt',
       score_funcs={'Acc':accuracy_score,'F1': f1_score})
```

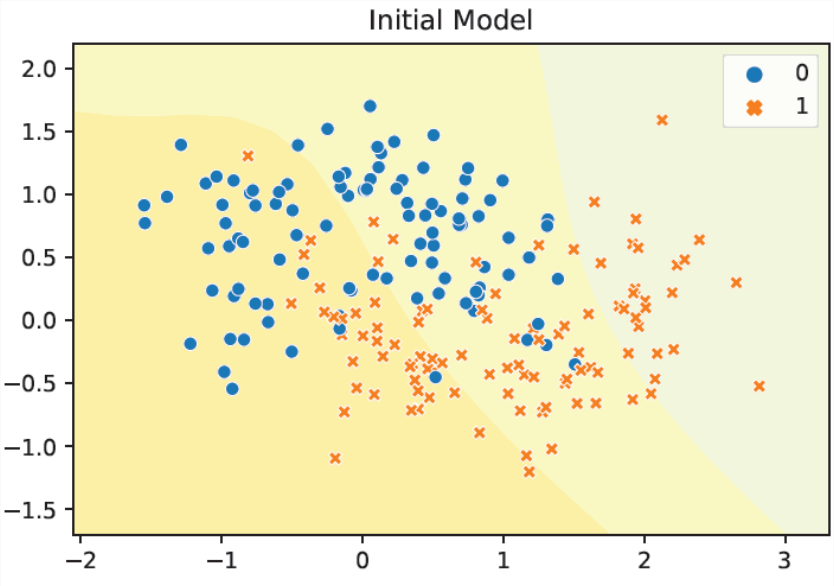
Now lets look at some results. First, lets see that we can load our check point `model`, rather than just use the one already trained. In order to load a `model`, we first need to define a *new* `model` that has all the same sub-modules as the original, and they all need to be the same size. This is necessary so that the weights all match up. If you saved a model with 30 neurons in the second hidden layer, we need to have a new model with 30 neurons as well, otherwise there will be too few or too many, and an error will occur.

One reason we use the `torch.load` and `torch.save` functions is the `map_location` argument that it provides. This handles loading a model from the data to the correct compute device for us. Once we load in the dictionary of results, we can use the `load_state_dict` function to restore the states of our original model into this new object. Then we can apply it to the data, and see that we get the same results.

```
model_new = nn.Sequential(
    nn.Linear(2, 30),
    nn.Tanh(),
    nn.Linear(30, 30),
    nn.Tanh(),
    nn.Linear(30,2),
)

visualize2DSoftmax(X_test, y_test, model_new, title="Initial Model")
```

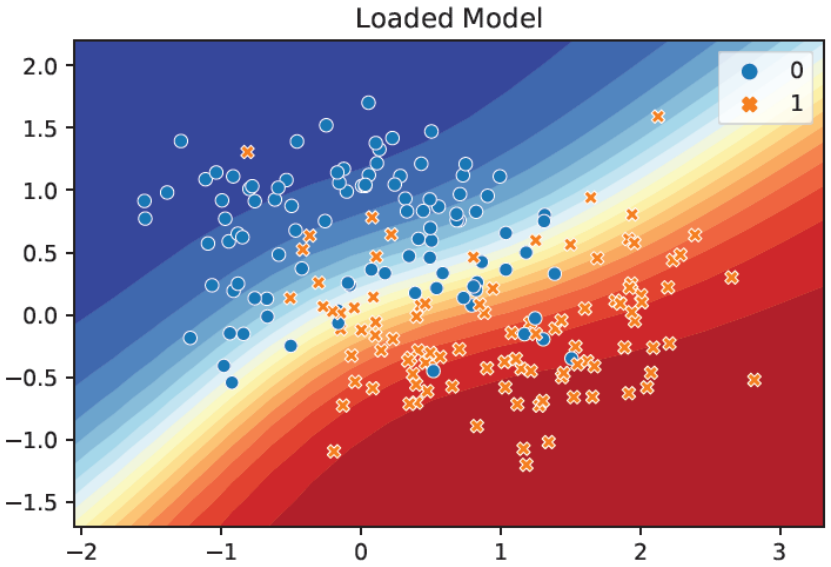
```
plt.show()

checkpoint_dict = torch.load('model.pt', map_location=device)


model_new.load_state_dict(checkpoint_dict['model_state_dict'])

visualize2DSoftmax(X_test, y_test, model_new, title="Loaded Model")
plt.show()
```

```
/home/edraff/anaconda3/lib/python3.7/site-packages/ipykernel\_launcher.py:13:
MatplotlibDeprecationWarning:
The ax attribute was deprecated in Matplotlib 3.3 and will be removed two minor
releases later.
  del sys.path[0]
/home/edraff/anaconda3/lib/python3.7/site-packages/ipykernel\_launcher.py:15:
MatplotlibDeprecationWarning:
The ax attribute was deprecated in Matplotlib 3.3 and will be removed two minor
releases later.
  from ipykernel import kernelapp as app
```




```
/home/edraff/anaconda3/lib/python3.7/site-packages/ipykernel\_launcher.py:13:
MatplotlibDeprecationWarning:
The ax attribute was deprecated in Matplotlib 3.3 and will be removed two minor
releases later.
  del sys.path[0]
```

```
/home/edraff/anaconda3/lib/python3.7/site-packages/ipykernel\_launcher.py:15:
MatplotlibDeprecationWarning:
The ax attribute was deprecated in Matplotlib 3.3 and will be removed two minor
releases later.
  from ipykernel import kernelapp as app
```

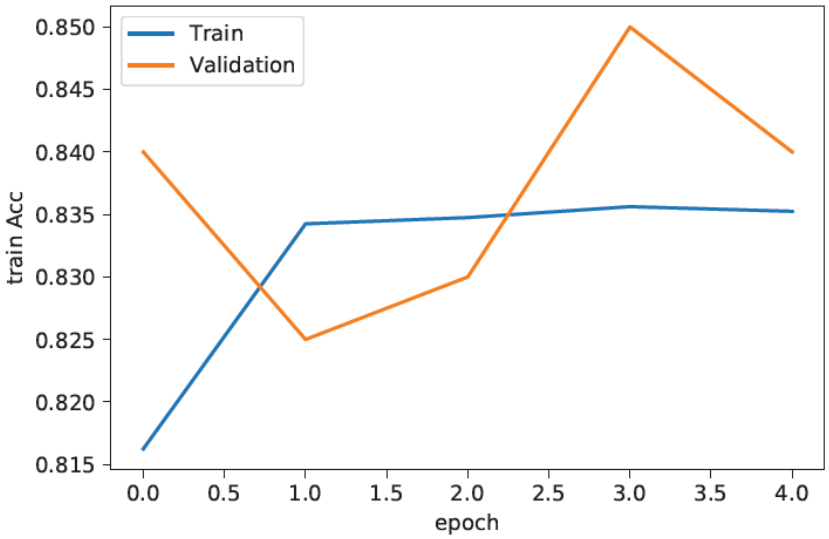


We can easily see that the initial model, because its weights are random values and untrained, does not give very good predictions. If you run the code several times, you should see many slightly different, but all equally unhelpful results. But after we load the previous `model` state into the `model_new`, we get the nice crisp results we expect.

Our new training function was written to return a Pandas dataframe object with information about the model after every epoch. This gives us some valuable information that we can easily visualize. For example, below we can quickly plot the training and validation accuracy as a function of the epoch that was finished.

```
sns.lineplot(x='epoch', y='train Acc', data=results_pd, label='Train')
sns.lineplot(x='epoch', y='val Acc', data=results_pd, label='Validation')
```

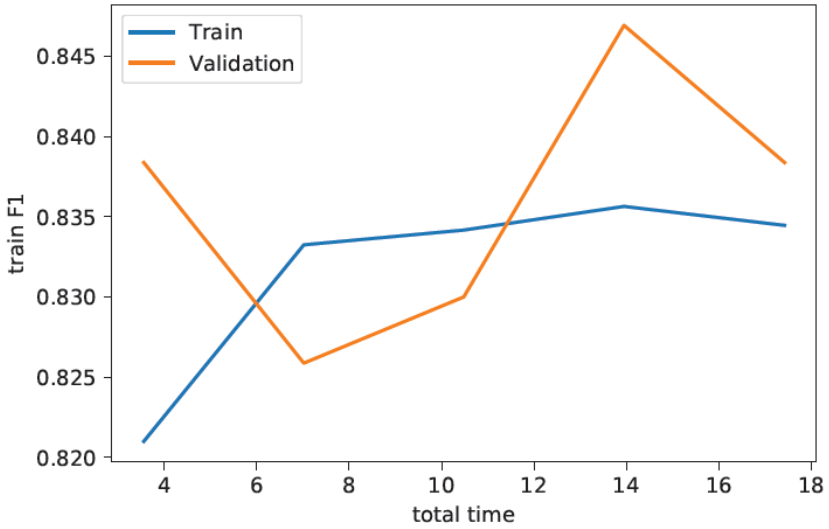
```
[29]: <AxesSubplot:xlabel='epoch', ylabel='train Acc'>
```



Its now easy to see that using more data, it took about 2 epochs for our model to top out on the noisier training training data. Two score functions were provided, so lets look at the F1 score as a function of the literal amount of training time in seconds. This will become more useful in the future if we want to compare how quickly two different models can learn.

```
sns.lineplot(x='total time', y='train F1', data=results_pd, label='Train')
sns.lineplot(x='total time', y='val F1', data=results_pd, label='Validation')
```

```
[30]: <AxesSubplot:xlabel='total time', ylabel='train F1'>
```



For this toy dataset F1 and Accuracy are giving very similar scores due to the fact that both classes have similar behavior and are balanced in size. A more interesting trend you should notice is that the training accuracy increases and then stabilizes, but the validation accuracy has more

asperity as it moves up and down with each epoch of training. *This is normal*. The model will start to overfit to the training data which makes its performance look stable, and slowly inch upward as it is done "learning" and starts to "memorize" the harder data points. Because the validation data is separate, these small changes that may be good or bad on new data are unknown to the model, and so it can not adjust to get them consistently correct. Its important that we keep a separate validation or test set so that we can see this less biased view of how the model will actually perform on new data.

2.5 Training in Batches

If you look at the x-axis of the previous figure, when we plotted the F1 score as a function of training time, you may notice that it took us almost a whole minute to train a model on just 8000 data points with only $d = 2$ features. Given this long training time, how could we ever hope to scale up to larger datasets?

To fix this we need to train on *batches* of data. A batch of data is literally just some larger group of data. Lets say we had the following dataset of $N = 4$ items.

$$X^{4\times 2} = \begin{bmatrix} \boldsymbol{x_1} = [1,2] \\ \boldsymbol{x_2} = [3,4] \\ \boldsymbol{x_3} = [5,6] \\ \boldsymbol{x_4} = [7,8] \end{bmatrix}, \; y = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Our current code, over one epoch, will perform four updates. One update for each item in the dataset. This is why it is called *stochastic* gradient descent. The word stochastic is jargon we use to mean "random", but usually with some underlying purpose or "invisible hand" that drives the randomness. The stochastic part of the name SGD comes from us using only a portion of the shuffled data, instead of the whole dataset, to compute the gradient. Because it is shuffled we will get a different result every time.

If we push all N data points through the model and compute the loss over the whole datasets, $\nabla \sum_{i=1}^{N} \ell(f(\boldsymbol{x}_i), y_i)$, we will get the *true* gradient. This can also make our training more *computationally efficient* by processing all of the data at once instead of one at a time. So instead of passing in a vector with a shape of (d) as the input to a `model` $f(\cdot)$, we will pass in a matrix of shape (N, d). PyTorch modules are designed for this situation by default, we just need a way to tell PyTorch to group our data up into a larger batch. Turns out the `DataLoader` has this functionality built in with the optional `batch_size` argument. If unspecified, it defaults to `batch_size=1`. If we set this to `batch_size=len(train_dataset)`, then we will perform true gradient descent.

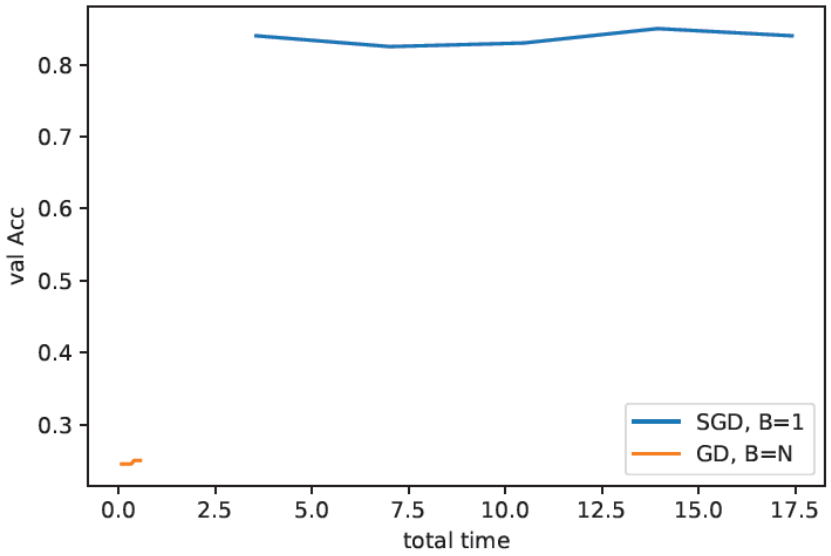
```
training_loader = DataLoader(train_dataset, batch_size=len(train_dataset), shuffle=True)
testing_loader = DataLoader(test_dataset, batch_size=len(test_dataset))
model_gd = nn.Sequential(
    nn.Linear(2, 30),
    nn.Tanh(),
    nn.Linear(30, 30),
    nn.Tanh(),
```

```
    nn.Linear(30,2),
)
results_true_gd = train_simple_network(model_gd, loss_func, training_loader, epochs=5,
       val_loader=testing_loader, checkpoint_file='model.pt',
       score_funcs={'Acc':accuracy_score,'F1': f1_score})
```

Five epochs of training just wen down to {{results_true_gd ['total time'].values [-1]}} seconds. Clearly training on *more* data at once time has allowed us to benefit from the parallelism available inside of a modern GPU. But if we plot the accuracy below, we see that training the gradient descent $(B = N)$ has produced a less accurate model.

```
sns.lineplot(x='total time', y='val Acc', data=results_pd, label='SGD, B=1')
sns.lineplot(x='total time', y='val Acc', data=results_true_gd, label='GD, B=N')
```

```
[32]: <AxesSubplot:xlabel='total time', ylabel='val Acc'>
```



Lets look at a toy example to explain why this is happening. Below shows a function that we are optimizing, and if we are using gradient descent (that looks at *all* the data), we take steps that are leading us in the correct direction. But, each step is expensive, so we can only take a few steps. This toy example shows us taking four total updates/steps corresponding to four epochs.

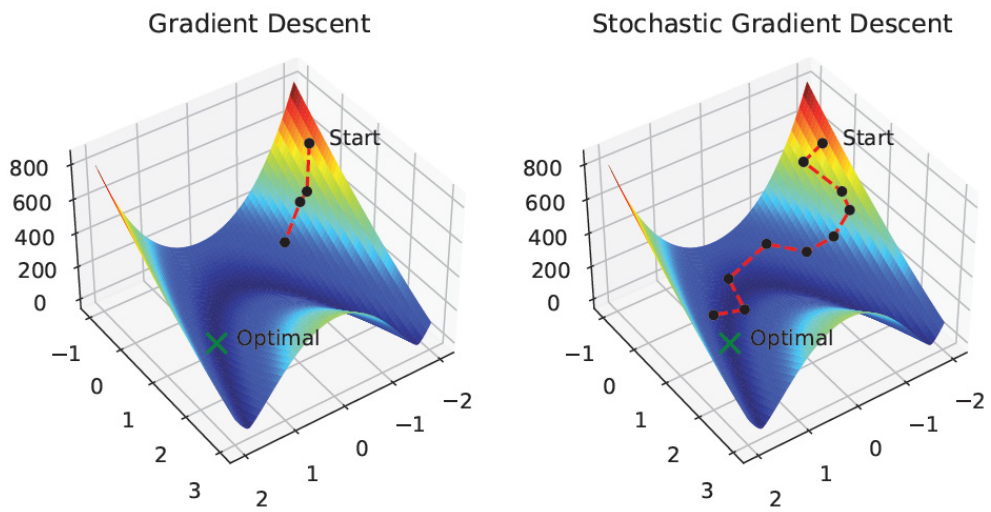


Figure 2.9: The left image shows gradient desent over 4 epochs of the data. That means it can only make 4 steps of progress, but each step is headed in the right direction. Stochastic Gradient Descent (SGD) on the right makes multiple updates per epoch by looking at just some of the data. This means each step is noisy, and not always in the correct direction. But they are usually in a useful direction, and so SGD can often make more progress toward the goal in fewer epochs.

When we use *stochastic* gradient descent, we perform N updates per epoch, so we get more updates or steps in for a fixed number of epochs. But, because of the *stochastic* or "random" behavior of using just one data point for each update, the steps we take are noisy. So they don't always head in the correct direction. The larger total number of steps eventually gets us closer to the answer. The cost is an increase in runtime because we lost the computational efficiency of processing all of the data at once.

The solution we use in practice is to balance between these two extremes. Lets choose a batch size big enough to leverage the GPU more efficiently, but small enough that we still get to perform many more updates per epoch. We will use B to denote the batch size we use, and for most applications you will find $B \in [32, 256]$ is a good choice. Another good rule of thumb is to make the batch size as large you as can fit into GPU memory, and add more training epochs until the model converges. This requires a bit more work on your part because as you are developing your network and make changes, the largest batch size you can fit onto your GPU may change.

NOTE **Because we only use the validation data to *evaluate* how our model is doing, and not to update the weights of the model, the batch size used to the validation data has no particular trade off. We can just increase the batch size to whatever runs fastest, and go with that. The results will be the same irrespective of the batch size used for the test data. In practice, most people use the same batch size for training and testing data just for simplicity.**

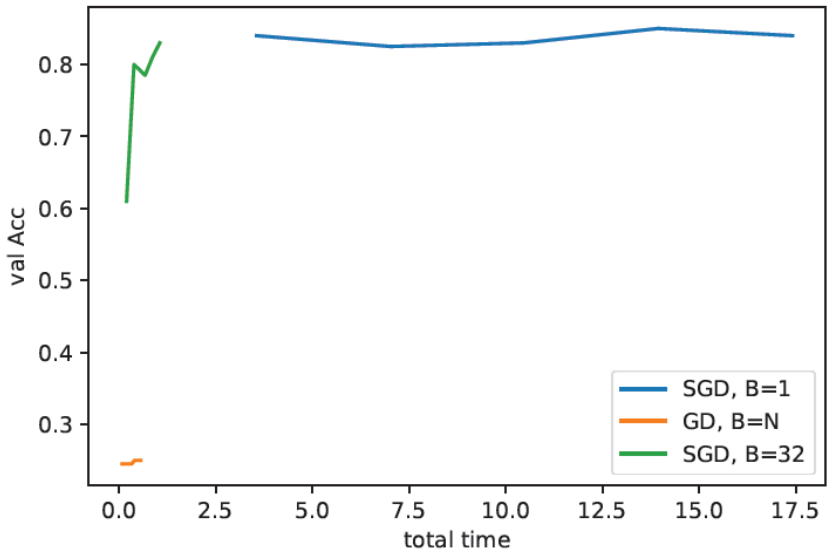
```
training_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
model_sgd = nn.Sequential(
    nn.Linear(2, 30),
    nn.Tanh(),
    nn.Linear(30, 30),
    nn.Tanh(),
    nn.Linear(30,2),
```

```
)
results_batched = train_simple_network(model_sgd, loss_func, training_loader, epochs=5,
       val_loader=testing_loader, checkpoint_file='model.pt',
       score_funcs={'Acc':accuracy_score,'F1': f1_score})
```

Now if we plot the results as a function of time, we see the green line giving us the best of both worlds. It runs in only {{round(results_batched [‘total time’].values [-1], 3)}} seconds and gets nearly the same accuracy. You will find that using batches of data like this will have almost no down side and is the preferred approach in modern deep learning.

```
sns.lineplot(x='total time', y='val Acc', data=results_pd, label='SGD, B=1')
sns.lineplot(x='total time', y='val Acc', data=results_true_gd, label='GD, B=N')
sns.lineplot(x='total time', y='val Acc', data=results_batched, label='SGD, B=32')
```

```
[35]: <AxesSubplot:xlabel='total time', ylabel='val Acc'>
```



2.6 Exercises

1. The input range of data can have a large impact on a neural network. This applies to inputs *and* outputs, like for regression problems. Try applying Scikit-learn’s `StandardScaler` to the targets $\boldsymbol{y}$ of the toy regression problem at the start of this chapter, and train a new neural network on it. Does changing the scale of the outputs help or hurt the model’s predictions?
2. The AUC metric does not follow the standard pattern in scikit-learn, as it requires `y_pred` to be a vector of shape (N) instead of a matrix of shape $(N, 2)$. Write a wrapper function for AUC that will make it compatible with our `train_simple_network` function.
3. Write a new function `resume_simple_network`, which loads a `checkpoint_file` from disk, restores both the `optimizer` and `model` state, and continues training to a specified total number of epochs. So if the model was saved after 20 epochs, and you specify 30 epochs, it should only perform 10 more epochs of training.
4. When performing experiments, we may want to go back and try versions of our model from different epochs, especially if we are trying to find when some weird behavior started to

occur. Modify the `train_simple_network` function to take a new argument `checkpoint_every_x`, that will save a version of the model every `x` epochs with different file names. That way you can go back and load a specific version, but don't fill your hard drive with a model for *every* epoch.

5. The "deep" part of deep learning refers to the number of layers in a neural network. Try adding more layers (up to 20) to the models we used for the `make_moons` classification problem. How do more layers impact the performance?
6. Try changing the number of neurons used in the hidden layers of the `make_moons` classification problem. How does it impact performance?
7. Use scikit-learn to load the breast cancer wisconsin dataset, and convert it into a `TensorDataset` and then split it into 80% for training and 20% for testing. Try to build your own classification neural network for this data.
8. We saw results on the `make_moons` dataset with a batch size of $B = \{1, 32, N\}$. Write a loop to train a new model on that same dataset for every power of two batch size less than N (i.e., $B = \{2, 4, 8, 16, 32, 64, \ldots\}$ and plot their results. Are there any trends that you notice in terms of accuracy and/or training time?

2.7 Chapter summary

1. We wrote a training loop that can be used for almost all neural networks we will learn about in this book.
2. We learned how to specify a loss function for classification (cross-entropy) or regression (MSE or L1) problems.
3. The `nn.Linear` layer can be used to implement Linear and Logistic regression.
4. Fully-Connected networks can be seen as extensions to Linear and Logistic regression by adding more `nn.Linear` layers with *non-linearities* inserted in between each layer.
5. The `nn.Sequential` `Module` can be used to organize sub- `Modules` to create larger networks.
6. We can trade off compute efficency vs number of optimization steps by using the `batch_size` option in a `DataLoader`

FOOTNOTES

1. Notice that the Chicken label is yellow, get it? I thought that was funny. Unlike stereotyped chicken, you shouldn't be afraid of a little math!↩

3

Convolutional Neural Networks

This chapter covers:

- How tensors represent spatial data like images
- Defining convolutions and their uses
- Building and training a Convolutional Neural Network (CNN)
- Adding pooling to make your CNNs more robust
- Augmenting image data to improve accuracy

Convolutional Neural Networks (CNNs) revitalized the field of neural networks while simultaneously ushering in a new branding of "deep learning" starting in 2011 and 2012. CNNs are still at the heart of many of the most successful applications of deep learning, including self-driving cars, speech recognition systems used by smart devices, and optical-character-recognition. All of this stems from the fact that *convolutions* are powerful yet simple tools that help us encode information about the problem into the design of our network architecture. Instead of focusing on feature engineering, we spend more time engineering the *architectures* of our networks.

The success of Convolutions comes from their ability to learn spatial patterns, which has made them the default method to use for any data resembling an image. When you apply a convolution to an image, you can learn to detect simple patterns like horizontal or vertical lines, changes in color, or grid-patterns. By stacking convolutions in layers, they begin to build up more complex patterns built from the simpler convolutions that came before.

Our goal in this chapter is to reach you all the basics so that you can build your own CNNs for new image classification problems. First we need to discuss how images are represented to a neural network. That images are 2D is an important "structure" or meaning we are going to encode into the specific way we organize data in our tensors. You should always care about the structure of your data, because picking the right architecture that matches the structure is the best way to improve the accuracy of your model. Next we want to remove the mystery of what a convolution is, and show how convolutions can detect simple patterns and why its a good approach to use for data that is structured like an image. Next we will take a convolution and create a "convolutional

layer", which can act as a replacement to a `nn.Linear` layer we used in the previous chapter. Finally, we will start to build some CNNs and discuss a few additional tricks to improve their accuracy.

3.1 Spatial Structural Prior Beliefs

As of now, you know how to build and train a very simple kind of neural network. What you have learned is applicable to any kind of tabular (also called "columnar") data, where your data and features could be organized in a spreadsheet. However, other algorithms (e.g., Random Forests and XGBoost) are usually better for such data. If all you have is columnar data, you probably do not want to be using a neural network at all.

When neural networks get really useful, and start to out-perform other methods, is when we use them to impose some kind of *prior belief*. We use the words "prior belief" in a very literal way, there is something we *believe* is true about how the data/problem/world works, *prior* to having ever looked at the data\footnote{If you have any friends who call themselves Bayesian, they might take offense at this definition - but that's *OK*. We are not being Bayesians today. Bayesian statistics often involves a more precise definition of a prior, see here for an introduction.}. Specifically, deep learning has been most successful at imposing *structural* priors. This is where, by how we design the network, we impart some knowledge about the intrinsic nature or "structure" of the data. The most common types of structure encoded into neural networks are spatial correlation (i.e., images in this chapter) and sequential relationships (e.g., weather changes from one day to the next). The below figure shows some of the cases where you want to use a CNN.

Columnar data: Fully-Connected layers

Feature 1	Feature 2	...	Feature *d*
1	Yes	...	2
8.2	No	...	5123
7	Yes	...	542

Audio data: Use a 1D *CNN*

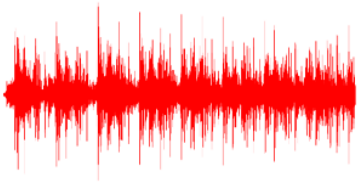


Image dataz: Use a 2D *CNN*

Figure 3.1: Columnar data (data that could go in a spread sheet) should use fully-connected layers, because there is no structure to the data and fully-connected layer impart no prior beliefs. Audio and Images have spatial properties that match how Convolutional Neural Networks see the world. So you should almost always use a CNN for those kinds of data. It is hard to listen to a book, so well stick with images instead of audio.

There are a number of ways to do this, and the list is growing all the time. For now we will talk about what are known as *Convolutional Neural Networks* (CNNs). These have dominated the image-based world.The first thing we need to learn is how an image and it's structure is encoded in PyTorch as tensors, so that we can understand how a convolution can use this structure. Previously we had input with no structure. Our data could be represented by an (N, D) matrix, with N data points and D features. We could have re-arranged the order features, and it would not have changed the meaning behind the data, because there is no structure / importance to how the data is organized. All that matters is that if column j corresponds to a specific feature, we always put that features value in column j (i.e., we just need to be consistent).

Images, however, are structured. There is an order to the pixels. If you shuffled the pixels around, you would fundamentally change the meaning of a picture. In fact, you would probably end up with an incomprehensible "image" if you did that. The below figure shows how this works.

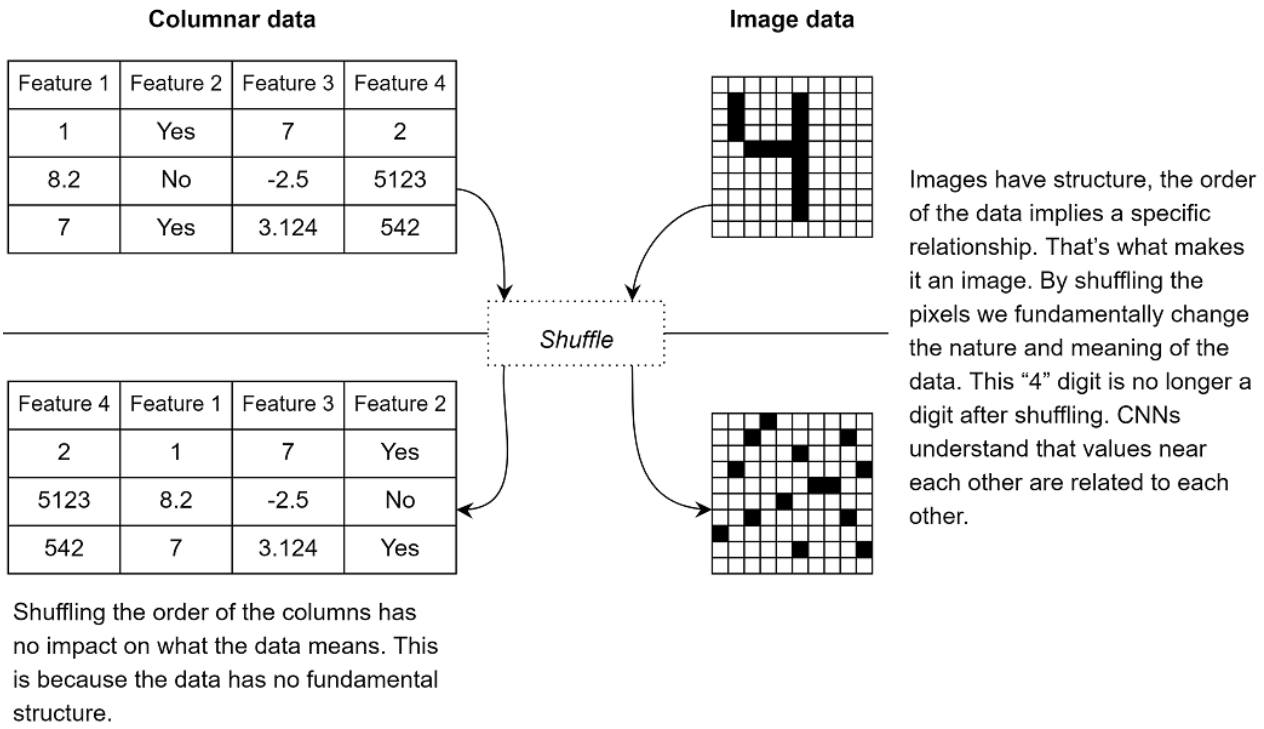


Figure 3.2: Shuffling your data will destroy structure present in your data. The left show columnar data where shuffling has no real impact, because it has no special structure. The right shows an image being shuffled, which is no longer recognizable. This is the structural nature of images that pixels near each other are related to each other. CNNs encode the idea that items located near each other are related, which makes CNNs a good fit for images.

So if we have N images, each has a height H, and a width W. As a starting point, we might consider a "matrix" of image data to have shape

$$(N, W, H)$$

This gives us a three dimensional tensor. This would be fine if we had black-and-white images only. But what about color? To fix this we need to add some *channels* to our representation. Every channel has the same width and height, but represents a different perceptual concept. Color us usually represented with a Red, Green, and Blue *channel*, and we interpret the mixture of red, green, and blue to create a final color image. This is shown in the below figure.

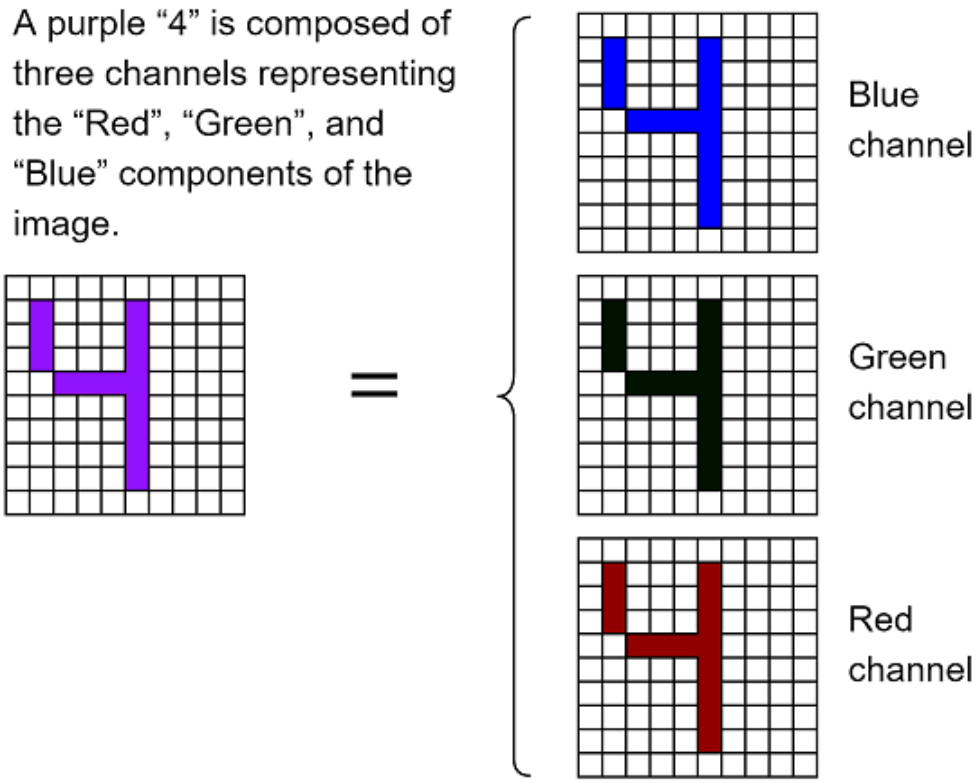


Figure 3.3: Color images are represented by "three" sub-images of the same size, called channels. Each channel represents a different concept, the most common being ``Red, Green, Blue'' (RGB) as shown above. In general, every channel represents a different kind of feature at different locations.

So if we want to include that, we need to have an additional dimension to the tensor for the channels. This becomes

(N, C, W, H)

Now we have a four dimensional tensor, and there is structure to the tensor. By structure we mean that the axes of the tensor, and the order we access data in, has a specific meaning (we can’t shuffle them). If `x` was a batch of color images `x[3,2,0,0]` is saying "from the 4th image $(N = 3)$ grab the blue value $(C = 2)$ of the upper left pixel $(0, 0)$. Or we could grab the red, green, and blue values using `x[3,:,0,0]`. This means we we are processing the pixel values at locations i and j, we know we need to access the index `x[:,:,i,j]`. More important, we need to know something about the *neighboring pixel to the bottom right*, we can access this value using `x[:,:,i+1,j+1]`. Thanks to the input being *structured*, this is true regardless of the values of i and j. Convolutions will use this so that when a convolution is looking at pixel location i,j in an image, it can also consider the neighboring pixel locations.

The Red, Green, Blue channels are the most common standard for images and denoted with the short hand “RGB”, but they are not the only option. These are often called “color spaces”. Another

popular way to represent data is using Hue, Saturation, and Value (HSV) and the Cyan, Magenta, Yellow, and Key (read, black) (CMYK) color space.

3.1.1 Loading MNIST with PyTorch

While it has become a bit cliché, we will start exploring what all of this means using the ubiquitous MNIST dataset. Its a collection of black and white images of the digits 0 through 9, each is 28 pixels wide and 28 pixels tall. PyTorch has a convenient loader for this dataset in a package called `torchvision`, which we will load below. If you are doing *anything* with images and PyTorch, you almost certainly want to be using this package. While MNIST is a toy problem, we will work with it for most chapters because it allows us to run examples in just a few minutes, where real datasets take hours to *weeks* for a *single run*. I've designed these chapters so that the approaches and lessons you will learn do transfer to real problems.

```
import torchvision
from torchvision import transforms
```

Now we can load the MNIST dataset using the below code. The first argument, `"./data"`, is just telling PyTorch where we would like the data stored, and `download=True` to download the dataset if its not already there. MNIST has a pre-defined training and testing split, which we can obtain by setting the `train` flag to `True` or `False` respectively.

```
mnist_data_train = torchvision.datasets.MNIST("./data", train=True, download=True)
mnist_data_test = torchvision.datasets.MNIST("./data", train=False, download=True)
x_example, y_example = mnist_data_train[0]
type(x_example)
```

```
[5]:PIL.Image.Image
```

Now you will notice that the `type` of the data returned was *not* a tensor. We got a `PIL.Image.Image` (pillow.readthedocs.io/en/stable), because the dataset *is* images. We need to use a `transform` to convert the images to tensors, which is why we have imported the `transforms` package from `torchvision`. We can simply specify the `ToTensor` transform, which converts a PIL image into a PyTorch tensor where the minimum possible value is 0.0, and the max is 1.0, so its already in a pretty good numerical range for us to work with. Lets redefine these dataset objects to do that right now. All it takes is adding `transform=transforms.ToTensor()` to the method call, as shown below where we load the train/test splits and print the shape of the first example of the training set.

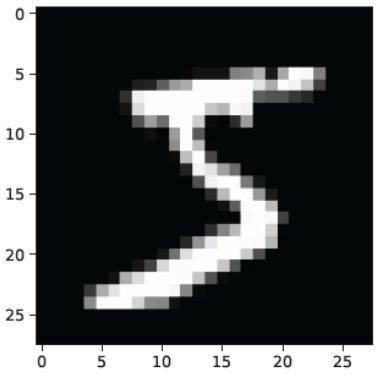
```
mnist_data_train = torchvision.datasets.MNIST("./data", train=True, download=True,
       transform=transforms.ToTensor())
mnist_data_test = torchvision.datasets.MNIST("./data", train=False, download=True,
       transform=transforms.ToTensor())
x_example, y_example = mnist_data_train[0]
print(x_example.shape)
```

```
torch.Size([1,28,28])
```

We have accessed a single example from the dataset, and it has a shape of $(1, 28, 28)$ for $C = 1$ channels (its black and white), and a width and height of 28 pixels. If we want to visualize a tensor representation of an image that is gray scale, `imshow` expects it to have only a width and height (i.e, a shape of (W, H)). The `imshow` function also needs us to tell it explicitly to use gray scale. Why? Because `imshow` is meant for a wider class of scientific visualization, where you might not want other options instead.

```
imshow(x_example[0,:], cmap='gray')
```

```
[7]:<matplotlib.image.AxesImage at  0x7fa4b2161110>
```



Ok, clearly that was the digit "5". Since we are learning how images are presented as tensors, lets go ahead and do a color version. If we just stack three copies of the same digit on top of each other, we will then have a tensor of shape $(3, 28, 28)$. Because the *structure* of the tensor has meaning, this instantaneously makes it a color image by virtue of having 3 channels. The code below does exactly that, stacking the first grayscale image three times and printing its shape.

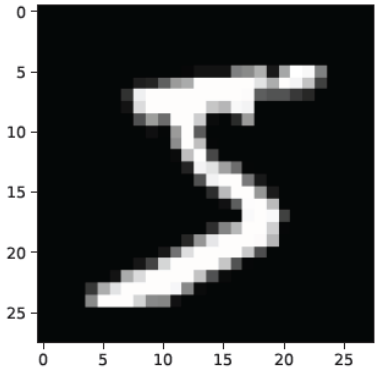
```
x_as_color = torch.stack([x_example[0,:], x_example[0,:], x_example[0,:]], dim=0)
print(x_as_color.shape)
```

```
torch.Size([3,28,28])
```

Now lets visualize the color version. Here we need to be a little careful. In PyTorch, an image is represented as (N, C, W, H). But `imshow` expects a single image as (W, H, C). So we will need to *permute* the dimensions when using `imshow`. If we have r dimensions to our tensor, the `permute` function takes r inputs, the indexes $0, 1, \ldots, r-1$ of the original tensor in the *new order* we want them to appear. Since our image has (C, W, H) right now, maintaing that order would be $(0, 1, 2)$. We want the channel at index 0 to become the last dimension, width the first, and height second, so that would be $(1, 2, 0)$. Lets try it out.

```
imshow(x_as_color.permute(1,2,0))
```

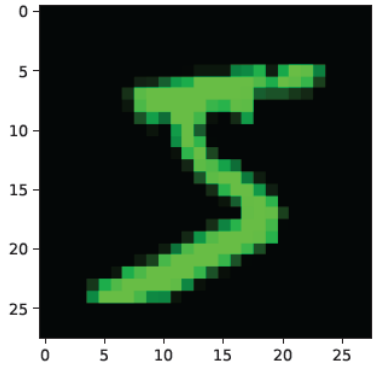
```
[9]:<matplotlib.image.AxesImage at  0x7fa4b0d25850>
```



Why was the above *color* image still black and white? Because the original image was black and white. We have the exact same value copied in the red, green, and blue channels, which is how you represent a black and white image in color. If we zero out the red and blue channels, we would get a green number.

```
x_as_color = torch.stack([x_example[0,:], x_example[0,:], x_example[0,:]])
x_as_color[0,:] =0 #No Red
#Leaving green alone
x_as_color[2,:] =0 #No Blue
imshow(x_as_color.permute(1,2,0))
```

```
[10]:<matplotlib.image.AxesImage at  0x7fa4b0d14e90>
```



Changing the color of the image above is an example of:

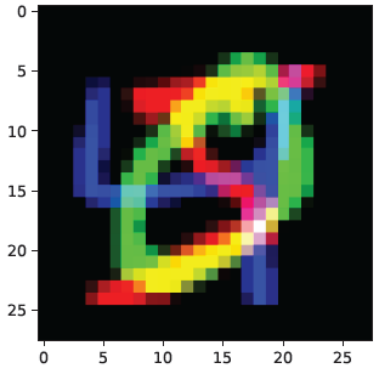
1. How the different channels impact what the data represents
2. What it means for this to be a *structured* data representation.

Just to make sure these two points are clear, lets stack three different images together into one color image. We will reuse the same 5 as the first image in the stack. That means it will go into the red channel, so we should see a red 5, mixed with two other digits that are green and blue respectively.

```
#grab 3 images
x1, x2, x3 = mnist_data_train[0], mnist_data_train[1], mnist_data_train[2]
#drop the labels
x1, x2, x3 = x1[0], x2[0], x3[0]
```

```
x_as_color = torch.stack([x1[0,:], x2[0,:], x3[0,:]], dim=0)
imshow(x_as_color.permute(1,2,0))
```

```
[11]: <matplotlib.image.AxesImage at  0x7fa4b0c8cb90>
```

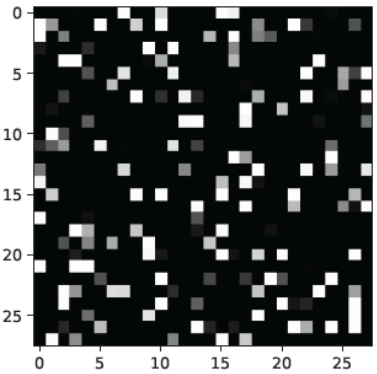


You should see a red 5, a green 0, and a blue 4. Where two of the images overlap, because they were placed in separate color channels, you will see the colors blend. For example in the middle the 4 and 5 intersect, and red + blue = purple.

The order of the data has meaning, and we can't simply arbitrarily re-order things without potentially destroying the structure, and thus, the data. Lets look at this more explicitly. What would happen if we were to shuffle the data within a channel? Does it have the same important structured meaning? Lets look at the digit 5 one last time, but instead we will randomly shuffle the values inside the tensor.

```
rand_order = torch.randperm(x_example.shape[1] * x_example.shape[2])
x_shuffled = x_example.view(-1)[rand_order].view(x_example.shape)
imshow(x_shuffled[0,:], cmap='gray')
```

```
[12]: <matplotlib.image.AxesImage at  0x7fa4b0bfde10>
```



As you can see, this has *completely* changed the meaning of the image. Instead of being a 5 its… nothing really. The location of a value, and it's *near by values* is intrinsically part of that value's meaning. The value of one pixel cant not be separated from its' neighbors. This is the *structural spatial prior* we are going to try and capture in this chapter. Now that we have learned how to

represent the structure of images with tensors, we can learn about convolutions to exploit that structure.

3.2 What are Convolutions?

So what do we change now that we have our data shaped like an image? There is a*prior* we would like to put into our model, which is that there is some kind of *spatial relationship* going on. The prior that convolutions encode is that *things near each other are related to each other, and far away things have no relationship*. Think about the pictures of the digit 5 above. Pick any black pixel. Most of its neighboring pixels are also black. Pick any white pixel, most of its neighbors are white or a shade of white. This is a kind of spatial correlation. It doesn't really matter where in the image this happens, because it tends to happen *everywhere* by the nature of *being an image*.

A convolution is a mathematical function with two inputs. Convolutions take an *input* image, and a *filter* (also called a *kernel*) and output a new image. The goal is that the filter will be able to recognize certain patterns from the input, and highlight them in the output. The below diagram illustrates this idea. A convolution can be used to impose a spatial prior on any tensor with r dimensions. A simple example of this is shown in . Right now we are just trying to understand what a convolution does, we will get to how it works in a moment.

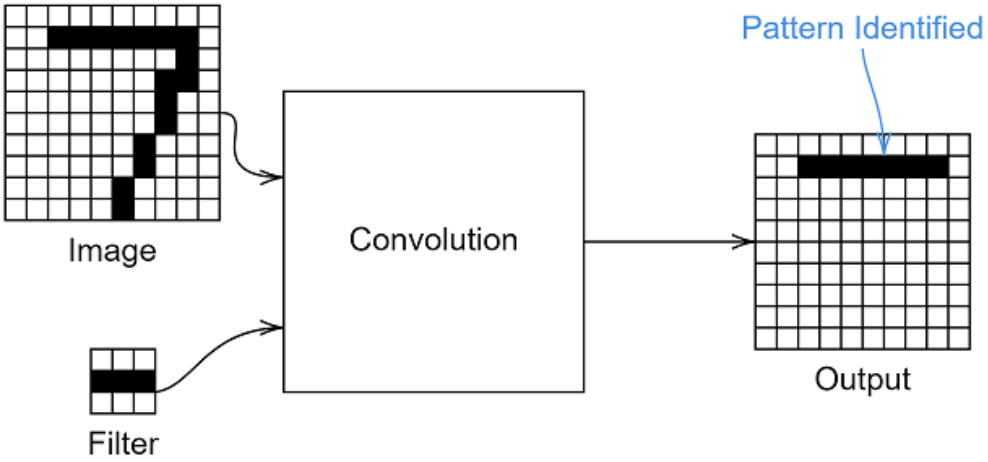


Figure 3.4: xample of a convolution. Some input image and a "filter" are combined by the convolution. The output is a new image that has been altered. The purpose of the filter is to identify / recognize certain patterns within the input. In this example, its identifying the horizontal line on top of a "7".

While we keep saying "images", convolutions are not constrained to working on only two dimensional data. To help us understand how convolutions work, we will start with a one dimensional example, because it makes the math easier to walk through together. Once we understand 1D convolutions, the 2D version we will use for images follows very quickly. Because we want to create multiple layers of convolutions, we will also learn about *padding* which is necessary for that goal. Finally, we will talk about another way to think about convolutions called *weight sharing*. This is a different way of thinking about convolutions that will re-emerge throughout the book (like in the next chapter).

3.2.1 1D Convolutions

To understand how a convolution works lets talk about a one-dimensional "image" first as it is easy to show the details in 1D than 2D. A 1D image would have a shape of (C, W), for the number of channels and the width. There is no height right now because we are talking about just 1D, not 2D. For a 1D input with (C, W) shape, we can define a filter with a shape of (C, K). We get to choose the value of K, and we need C to match up between the image and the filter. Since the number of channels must *always* match, we would call this a "filter of size K" for short. If we apply a filter of size K to an input of shape (C, W) we get an output that has shape $(C, W - 2 \cdot \lfloor K/2 \rfloor)$ Lets look at how that works in Figure 3.5.

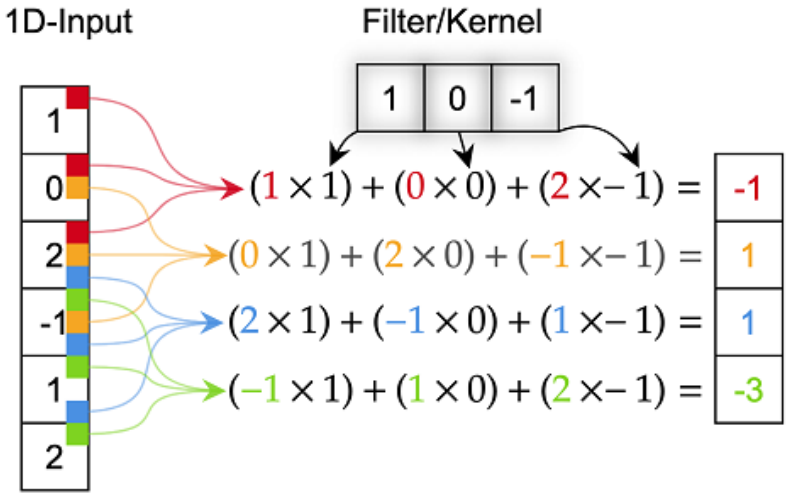


Figure 3.5: A 1D input "1, 0, 2, -1, 1, 2" is convolved with the filter "1, 0, -1". This means we take every sub-sequence of three input items, and multiple them with the filter values, and then add the results together.

An input of shape $(1, 6)$ is on the left and we are applying a filter of size 3 that has the values $[1, 0, -1]$. The output is on the right. For each output, you can see arrows coming in for the *spatially relevant* inputs for that output. So the first output of 1 only the first three inputs are relevant, nothing else in the input impacts that specific output. It's value is calculated by multiplying the first three inputs with the three values in the kernel, and then summing them together. The second value of the output is computed by using the second set of three input values. You will notice it is *always* the same three values from the filter, applied to *every* position in the input. The below block of code shows how you would implement this in raw python.

```
filter = [1,0, -1]
input  = [1,0,2, -1,1,2]
output = []
for i in range(len(input)-len(filter)): #Slide the filter over the input
    result =0
    for j in range(len(filter)): #apply the filter at this location
        result += input[i+j]*filter[j]
    output.append(result)
#now the output is ready to use
```

In effect, we are *sliding* the filter across every location in the input, computing a value at each location, and storing it in the output. Thats what a convolution is. The size of the output shrinks by $2 \cdot \lfloor 3/2 \rfloor$ because we run out of values at the edges of the input. Next well show how to make this work in 2D, and then we'll have the foundation of CNNs.

3.2.2 2D Convolutions

As we increase the number of dimensions r that our tensor has, the idea of convolutions and how they work stays the same: we will slide a filter around the input, multiplying the values in the filter with each area of the image, and then take the sum. We simply make the filter shape match accordingly. Lets look at a 2D example now, which aligns with the images we are going to try and process. Thats shown in Figure 3.6, where the $\circledast$ operator is introduced to mean "convolve".

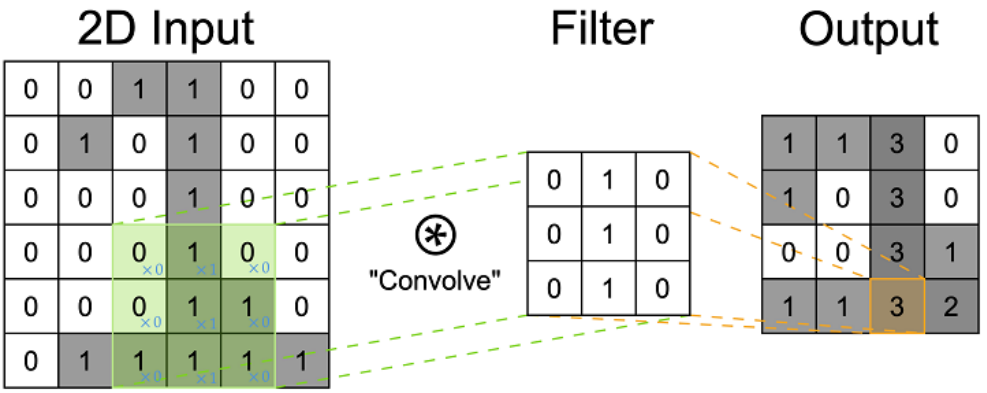


Figure 3.6: An image of a "1" convolved with a 2d filter. The green region shows the part of the image being currently convolved, with the values from the filter shown in light blue. The result in the output is shown in orange. By sliding the green/orange areas together over the entire image, you produce the output result.

Again, the 2D output is a result of multiplying the filter values (pair-wise) at each location, and summing them all together. The highlighted regions of the input are used to create the values in the output. In the bottom right corner of the input cells, you see the filter value we are multiplying by. In deep learning we almost always use "square" filters, meaning all r dimensions of the filter have the exact same number of values. So in this case we would call this a 2D filter of size K, or just "size K" for short. In this case, $K = 3$. The code for a 2D convolution is going to double the number of loops, the below block showing what it would look like.

```
filter = [[0,1,0], [0,1,0], [0,1,0]]
input = [[0,0,1,1,0,0],
         [...],
         [0,1,1,1,1,1]
        ]
height, width = len(input), len(input[0])
output = []
for i in range(height-len(filter)): #Slide the filter over the rows
    row_out = []
    for j in range(width-len(filter)): #Slide the filter over the columns
        result =0
        for k_i in range(len(filter)): #apply the filter at this location
            for k_j in range(len(filter)):
```

```
                result += input[i+k_i][j+k_j]*filter[k_i][k_j]
            row_out.append(result) #build up a row of the output
        output.append(row_out) #add the row to the final output
#output is ready for use
```

Since this 2D input had a shape of $(1, 6, 6)$ and or kernel has a shape of $(1, 3, 3)$, we shrink the width and height by $2 \cdot \lfloor 3/2 \rfloor = 2$. So that means the height will be 6 pixels $- 2 = 4$ pixels, and we get the same result for the width with $6 - 2 = 4$ pixels wide. We now have the exact operation that is the foundation of CNNs for image classification.

3.2.3 Padding

You will notice that right now, every time we apply a convolution the output becomes skinner and shorter than the original input. That means if we kept applying convolutions over and over again, we would eventually be left with nothing. This is not something we usually want, because we are going to create multiple layers of convolutions. Most modern deep learning design practices involve keeping the input and output the same size so that we can more easily reason about what shapes of our network and so that we can make them as deep as we would like, without worrying about the input disappearing to nothing. The solution to this is called *padding*. You should almost always use padding by default so that you can make changes to your architecture without having to concern yourself with changes in the shapes of your tensors. Figure 3.7 shows how this works for the same 2D image.

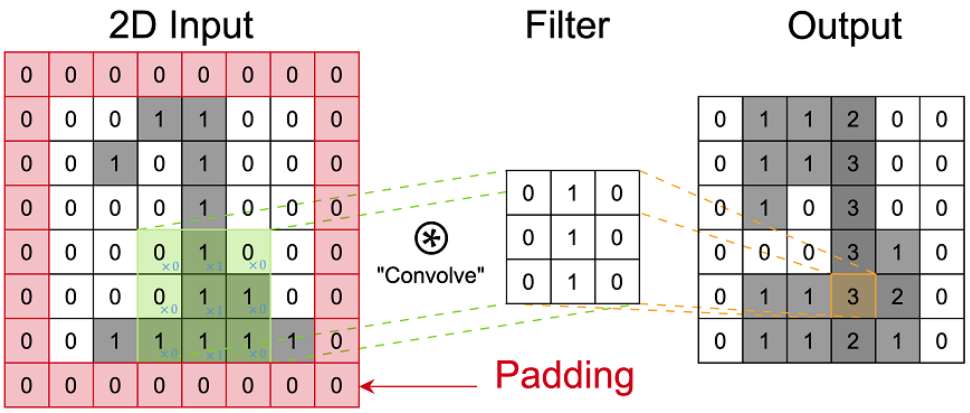


Figure 3.7: The same convolution as before with the same input and filter, but the input is "padded" with the value "0" to make the image larger by one pixel in each direction. This causes the output image to become the same width and height as the original image.

We add an "imaginary" row/column of zeros all the way around the image, and process it as if it was larger than it actually is. This specific example would be called "zero padding by one", because we added one value to all the edges of the image, and the value was filled with "0". If we use a convolutional filter of size K, we can use padding of $\lfloor K/2 \rfloor$ to make sure our output stays the same size as our input. Again, even though there is a height and width that could be padded to different degrees, we generally use the same amount of padding in each dimension because our filters have the same size in each dimensions.

3.2.4 Weight Sharing

There is another way to think about convolutions, which introduces an important concept called *weight sharing*. Lets look at the 1D case again just because it is simpler to write code for. Imagine you had a neural network $f_\Theta(\cdot)$ with parameters (or *weights*) Θ, which takes in an input vector $\boldsymbol{z}$ with K features, $\boldsymbol{z} \in \mathbb{R}^K$. Now lets say we have a larger input $\boldsymbol{x}$ with $C = 1$ channels and D features, where $D > K$. We can not use $f_\Theta(\cdot)$ on $\boldsymbol{x}$ because the shapes $D \neq K$ do not match.

One way we could apply the network $f_\Theta(\cdot)$ to this larger dataset would be to *slide* the network across slices of the input, and *share* the weights Θ for each position. Some python pseudo-code would like like this:

```
x = torch.rand(D)#some input vector
output = torch.zeros(D-K//2*2)
for i in range(output.shape[0]):
    output[i] = f(x[i:i+K], theta)
```

Now, if we fined our network as `f = nn.Linear(K, 1)`, this would actually implement *exactly a 1D convolution*. This insight can teach us some important properties about convolutions and how we will use them in the design of deep neural networks. Right now, the primary think this teaches us is that *convolutions are linear operations, that work spatially*. Like the `nn.Linear` layer from last chapter, this means a convolution followed by a second convolution is equivalent to doing just one, slightly different, convolution. That means:

3. Never repeat convolutions, because it is redundant.
4. We need to include a non-linear activation function after we use convolutions.

NOTE **If we had a rectangular kernel of size $(1, 3, 5)$, the width of the output image would be $2 \cdot \lfloor 3/2 \rfloor = 2$ smaller than the input. The height of the output would become $2 \cdot \lfloor 5/2 \rfloor = 4$, removing *two* from each side. While rectangular kernels are possible, they are rarely used. You will also notice that we have been sticking with kernels that have an *odd* size, nothing divisible by two. This is mostly a choice out of representational convenience, because it means there is an exact "center" that the filter is looking at from the input to produce each output. Again, filters with even sizes are possible, but rarely used.**

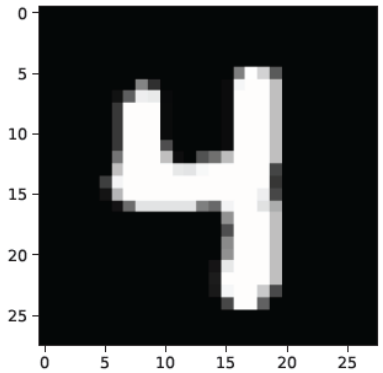
3.3 Utility of Convolutions

At this point we have spent a lot of time talking about what convolutions are. Now its time to start seeing what convolutions can do. Before adding them to a neural network, convolutions have had a rich history of use in computer vision applications on their own. It turns out this simple operation alone can define many useful things, so long as we select the appropriate kernel.

To start, lets again look at a *specific* image of the digit 4 from MNIST. We will load the scipy `convolve` function for use, and define the `img_index` so that you can change what image we are processing and see how these convolutions work across other inputs too.

```
from scipy.signal import convolve
img_indx =58
img = mnist_data_train[img_indx][0][0,:]
plt.imshow(img, vmin=0, vmax=1, cmap='gray')
```

```
[13]: <matplotlib.image.AxesImage at  0x7fa4b09ebb90>
```

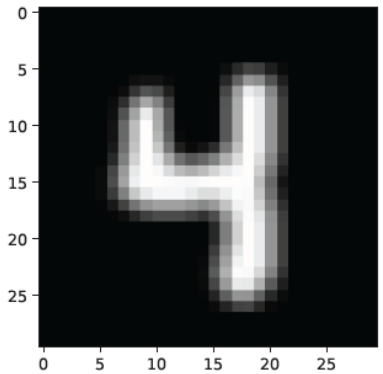


One common computer vision operation is to *blur* an image. Blurring involves taking a local average pixel value and replacing every pixel with the average of its neighbors. This can be useful to wash out small noisy artifacts or. to soften a sharp edge. This can be done with a *blur kernel* where:

$$\text{Image} \circledast \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \text{Blurred Image}$$

We can take this math and convert it directly into code. The matrix will be an `np.asarray` call, we've already loaded the image, and the convolution $\circledast$ is done with the `convolve` function. Then when we show the output image, we get a blurry version of the digit "4".

```
blur_filter = np.asarray([[1,1,1],
                          [1,1,1],
                          [1,1,1]
                         ])/9.0

blurry_img = convolve(img, blur_filter)
plt.imshow(blurry_img, vmin=0, vmax=1, cmap='gray')
plt.show()
```



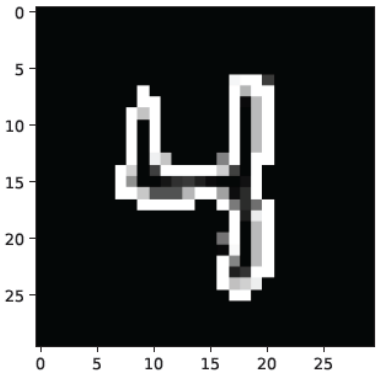
An especially common application of convolutions is to perform *edge detection*. In any computer vision application, its always good to know where the edges are. Edges can help you determine

where the edges of a road are (you want your car to stay in its lane) or find objects (edges in different shapes are easy to recognize). In the case of this "4", that would be the outline of the digit. So if everything in a local area of the image is the *same* we want everything to cancel out and result in no output. We only want there to be an output when there is a local change. Again, this can be described as a kernel where everything around the current pixel is negative, and the center pixel counts for the same weight as all its neighbors.

$$\text{Image} \quad \circledast \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} = \text{Edge Image}$$

This filter is maximized when everything around a pixel is different than itself. Lets see what happens.

```
#We can find edges by focusing on the difference between a pixel, and its neighbors
edge_filter = np.asarray([[-1,-1,-1],
                          [-1,8,-1],
                          [-1,-1,-1]
                         ])


edge_img = convolve(img, edge_filter)
plt.imshow(edge_img, vmin=0, vmax=1, cmap='gray')
plt.show()
```

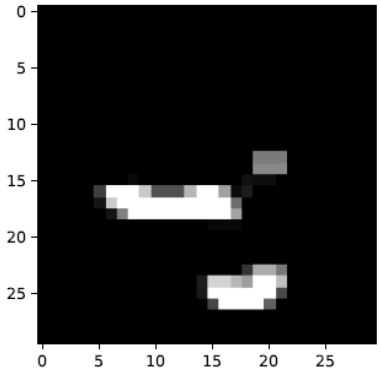


As promised, it found the edges of the digit. The response only occurs at the edges because thats where the most changes are. Outside the digit there is no response because the high center weight cancels out all the neighbors, which is also true for the *inside* region of the digit. Thus, we have now found all the edges inside the image.

We also may want to look for images at a *specific* angle. If we constrain ourselves to a 3×3 kernel, it is easiest to find horizontal or vertical edges. Lets make one for horizontal edges by making everything the kernel values change signs across the horizontal of the filter.

$$\text{Image} \circledast \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} = \text{Horizontal Edge Image}$$

```
#We could look for only horizontal edges
h_edge_filter = np.asarray([[-1,-1,-1],
                            [0,0,0],
                            [1,1,1]
                            ])


h_edge_img = convolve(img, h_edge_filter)
plt.imshow(h_edge_img, vmin=0, vmax=1, cmap='gray')
plt.show()
```



Now we see that we identified only the horizontal edges of the image, which is primarily the bottom bar of the 4. As we define more useful kernels, you can start to imagine how we might compose a *combination* of filters to start recognizing higher level concepts. Imagine we we only had the vertical and horizontal filters, we would not be able to classify all 10 digits, but Figure 3.8 shows how we could narrow down the answer.

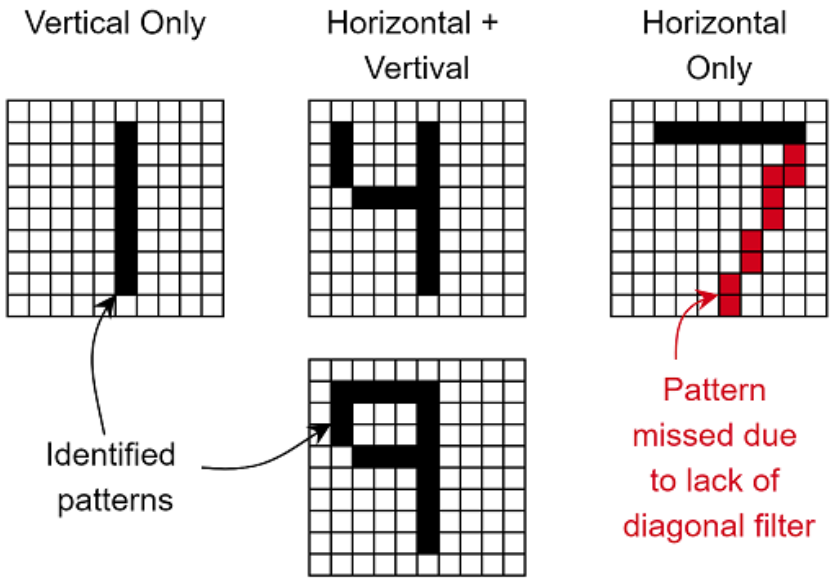


Figure 3.8: Example where we only have a vertical and horizontal edge filter. If only the vertical filter turns on, we are probably looking at the digit "1". If only the horizontal filter turns on, we could be looking at a "7", but we would really like an additional "diagonal" filter to help us confirm. If both the horizontal and vertical filters respond, we could be looking at a "4" or a "9", its hard to tell without more filters.

Designing all the filters you might need by hand was a big part of computer vision for many decades. You could even start to think about how convolutions on top of convolutions could start to learn larger concepts. For example, after identifying horizontal and vertical edges, a new filter might take those as input and look for a an empty space in the center with horizontal edges on top and vertical edges on the side. This would get us the "O" like shape to tell the difference between a "9" and a "4".

Thanks to deep learning though, we don't have to go through all the mental effort of trying to imagine and test all the filters we might need. Instead, we can let the neural network learn the filters itself. That way we save ourselves from a labor intensive process, and the kernels are optimized for the specific problem we care about.

3.4 Putting it to Practice, our first CNN

Now we have discussed what a convolution is, lets bring it back to some mathematical symbols and PyTorch code. We've seen that we can take an image $I \in \mathbb{R}^{C,\ W,\ H}$ and apply a convolution using a filter $g \in R^{C,\ K,\ K}$, to get a new result image $\mathbb{R}^{W,\ H}$. If we are writing this out in math, we would write it as :

$$\underbrace{R}_{\text{The result}} = \underbrace{I}_{\text{input image}} \underbrace{\circledast}_{\text{"Convolve"}} \underbrace{g}_{\text{a filter}}$$

That means each filter is going to look at all C input channels on its own. Lets take a look at that using the 1D input example in Figure 3.9 since it is easier to visualize.

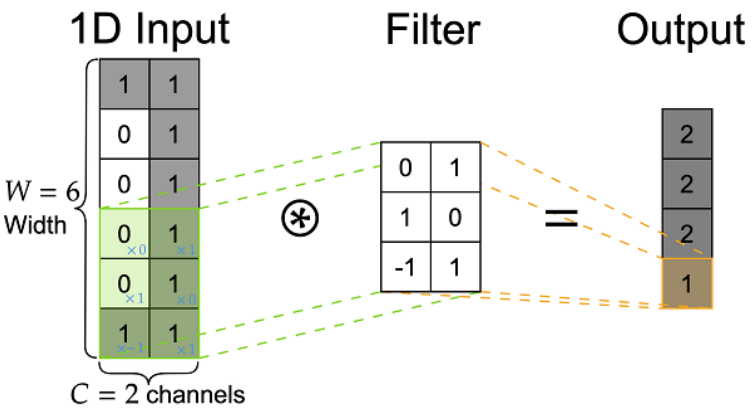


Figure 3.9: Example of a 1D convolution with two channels. Because there are two channels, the 1D input looks like a matrix and has two axes. The filter also has two axes, one for each channel. This filter is slid from top to bottom, and produces\emph{one} output channel. One filter will always produce one output channel, regardless of however many input channels there are.

Because the input has a shape of (C, W), the filter has a shape of (C, K). So when the input has multiple channels, the kernel will have a value for each channel separately. That means for a color image, we could have a filter that looks for "red horizontal lines, blue vertical lines, and no green" all in one operation. But it also means that after applying *one filter* we get *one output*.

Considering the examples above, we probably want more than just *one* filter though. We want to have C_{out} different filters, and lets go ahead and use C_{in} to indicate the number of channels in the input. In this case we would then have a tensor that represents all of the filters as $G \in \mathbb{R}^{C_{out}, C_{in}, K, K}$, so that we get a new result $R \in \mathbb{R}^{K, W', H'}$ when we write $R = I \circledast G$.

So how do we convert this math notation, that we are going to convolve some input image I with a set of multiple filters G? PyTorch provides the `nn.Conv1d`, `nn.Conv2d`, and `nn.Conv3d` functions for handling all of this for us. Each of these implements a convolutional layer for 1, 2, and 3-D data. Lets quickly diagram that out in Figure 3.10 to understand what is happening as a mechanical process.

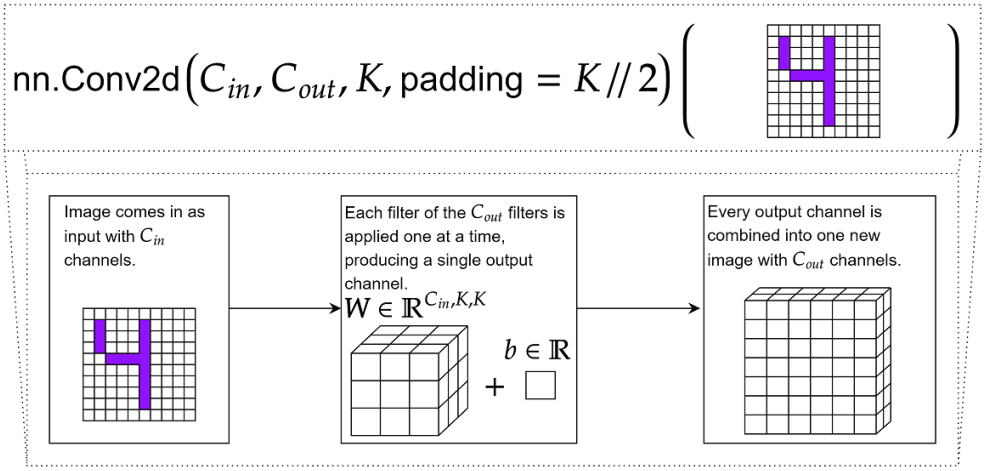


Figure 3.10: The `nn.Conv2d` defines a convolutional layer, and works in three steps. First it takes in the input image which has C_{in}) channels. As part of the construction, the `nn.Conv2d` takes in the number of filters to use C_{out}). Each filter is applied to the input one at a time, and their results combined. So the input tensor's shape will be $((B, C_{in}, W, H))$ and the output tensor will have a shape of $((B, C_{out}, W, H))$.

The same process is used by all three standard convolution sizes. `Conv1d` works on tensors that are (Batch, Channels, Width), `Conv2d` does (Batch, Channels, Width, Height), and `Conv2d` does (Batch, Channels, Width, Height, Depth). The number of channels that are in the input is C_{in}, and the convolutional layer is made of C_{out} filters/kernels. Since each of these filters produces one output channel, that means the output of this layer is going to have C_{out} channels. The value K defines with size of the kernel being used.

To help drive home how this works, lets dive a little deeper and show the full process in every detail. Figure 3.11 shows all the steps and math for an input image with explicitly $C_{in} = 3$, $C_{out} = 2$, and $K = 3$.

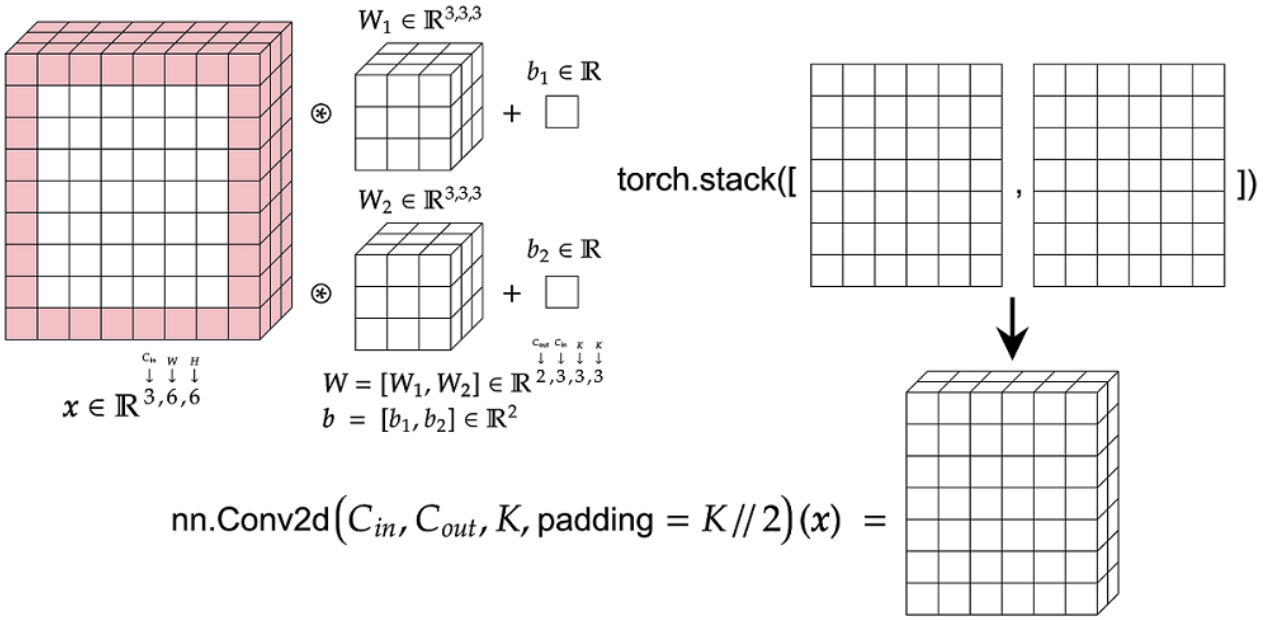


Figure 3.11: Example showing how convolutions get applied to multiple channels of a single image. The left shows the input image, which has padding shown in red. The middle shows the parameters of two filters, W_1 and W_2 (with their associated bias terms b_1, b_2). Each filter is convolved with the input to produce one channel. The results are stacked together to create a new ``image'' that has two channels, one for each filter. This whole process is done for us by the PyTorch `nn.Conv2d` class.

The input image comes in and has $C_{in} = 3$ channels, and we are going to process it using `nn.Conv2d(C_in, C_out, 3, padding=3//2)(x) = output`. Since $C_{out} = 2$, that means we process the input with two *different* filters, add the bias terms for every location, and get two resulting images that have the same height and width as the original image (because we used padding). The results are stacked together into one larger single image with 2 channels because we specified $C_{out} = 2$.

Now, when we had a fully connected layer, we wrote something like Figure 3.12 for a single hidden layer with n hidden units/neurons.

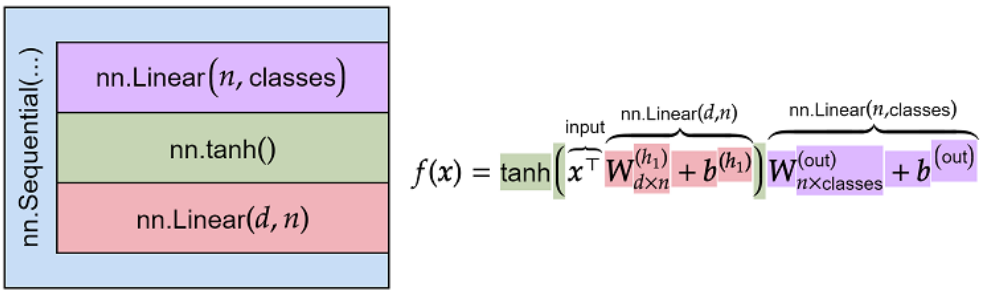


Figure 3.12: One hidden layer in a fully connected network. Left shows the PyTorch `Module`s that match the

equation on the right. Colors show which `Module` maps to which part of the equation.

The mathematical notation we would use to describe a network with one *convolutional* hidden layer is very similar:

$$f(\boldsymbol{x}) = \tanh\left(\overbrace{\boldsymbol{x}_{C_{\text{in}},W,H}}^{\text{input image}} \overbrace{\circledast \boldsymbol{W}^{(h_1)}_{C_{\text{out}},C_{\text{in}},K,K} + \boldsymbol{b}^{(h_1)}}^{\text{nn.Conv2d}(C_{\text{in}},C_{\text{out}},K)}\right) \overbrace{\boldsymbol{W}^{(\text{out})}_{(C_{\text{out}}\cdot w\cdot h),C} + \boldsymbol{b}^{(\text{out})}}^{\text{nn.Linear}(C_{\text{out}}\cdot w\cdot h,C)}$$

That equation may look a little scary, but if we slow down its not that bad. Its "intimidating" because we have included the shape of each tensor in the equation, and annotated where our fancy new `nn.Conv2d module` comes into play. I've included the shape information so that you can see how we are processing different sized inputs, and some people like to follow that level of detail. If we remove all those extra details, its not as scary looking.

Equation 3.1 $f(\boldsymbol{x}) = \tanh(\boldsymbol{x} \circledast \boldsymbol{W}_{(h_1)} + \boldsymbol{b}_{(h_1)})\boldsymbol{W}_{(out)} + \boldsymbol{b}_{(out)}$

Now it looks much friendlier, and it becomes clear that the only thing we have actually changed is replacing the dot product (a liner operation denoted by "$^{\intercal}$") with convolution (a spatially linear operation denoted by "$\circledast$").

This is *almost* perfect, but we have one issue. The output of the convolution has a shape of (C, W, H) but our linear layer ($\boldsymbol{W}_{(out)} + \boldsymbol{b}_{(out)}$) expects something of shape $(C \times W \times H)$, that's *one* dimension that has all the three original dimensions collapsed into it. Essentially, we need to *reshape* the output of our convolutions to remove the spatial interpretation, so that our linear layer can process the result and compute some predictions. This looks like Figure 3.13.

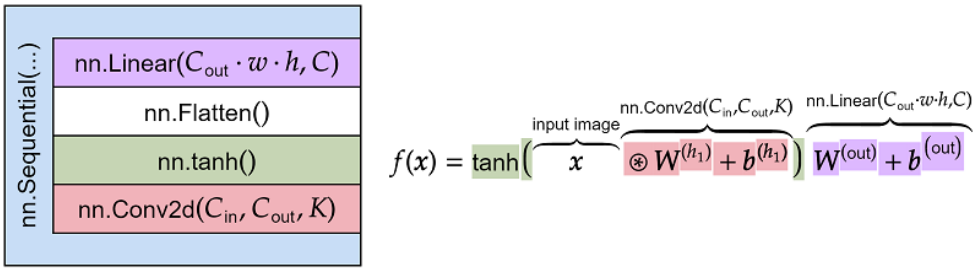


Figure 3.13: One hidden layer in a convolutional connected network. Left shows the PyTorch `Modules` that match the equation on the right. Colors show which `Module` maps to which part of the equation.

This is called *flattening* , and is a common operation in modern deep learning. We can consider Equation 3.1 to have implicitly used this flatten operation. PyTorch provides `module` for doing this called `nn.Flatten`. We will often write this with "implicit" bias terms as

$f(\boldsymbol{x}) = \tanh\left(\boldsymbol{x} \circledast \boldsymbol{W}^{(h1)}\right)\boldsymbol{W}^{(out)}$.

Now lets finally define some code for training this CNN based model. First we need to grab the CUDA compute device, and create a `DataLoader` for the training and testing set. We will use a batch size B of 32.

```
device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")

B =32
mnist_train_loader = DataLoader(mnist_data_train, batch_size=B, shuffle=True)
mnist_test_loader = DataLoader(mnist_data_test, batch_size=B)
```

Now we are going to define some variables. Again, because PyTorch works in batches if data, when we think in a PyTorch context our tensor shapes will begin with B, and since the input are images, the initial shape is (B, C, W, H). We have $B = 32$ because we defined that, and $C = 1$ because MNIST is black and white. Well define a few helper variables like `K` to represent our filter size, and `filters` for how many filters we want to build.

The first model we will is `model_linear`, because it uses only `nn.Linear` layers. It begins with calling `nn.Flatten()`. Notice the specific comment we put into the code `#(B, C, W, H) -> (B, C*W*H) = (B,D)`. This is to remind us that we are changing the shape of the tensor using this operation. The original shape (B, C, W, H) is on the left, and the new shape $(B, C \times W \times H)$ is on the right. Since we have the variable D to represent the total number of features, we also included a note on the value it is equal to `=(B,D)`. It is very easy to loose track of the shapes of tensors when writing code, and is one of the easiest ways to introduce bugs. I always try to include comments like this whenever the shape is altered by a tensor.

```
#How many values are in the input? We use this to help determine the size of subsequent layers
D =28*28 #28 * 28 images
#How many channels are in the input?
C =1
#How many classes are there?
classes =10
#How many filters should we use
filters =16
#how large should our filters be?
K =3
#for comparison, lets define a linear model of similar complexity
model_linear = nn.Sequential(
  nn.Flatten(), #(B, C, W, H) -> (B, C*W*H) = (B,D)
  nn.Linear(D,256),
  nn.Tanh(),
  nn.Linear(256, classes),
)

#A simple convolutional network:
model_cnn = nn.Sequential(
  #Conv2d follows the pattern of:
  #Conv2d(# of input channels, #filters/output-channels, #filter-size)
  nn.Conv2d(C, filters, K, padding=K//2), #$x\circledast G$
  nn.Tanh(),#Activation functions work on any size tensor
  nn.Flatten(), #Convert from (B, C, W, H) ->(B, D). This way we can use a Linear layer after
  nn.Linear(filters*D, classes),
)
```

The `model_linear` is a simple fully connected layer for us to compare against. Our first Convoltuional Neural Network (CNN) is defined by `model_cnn`, where we use the `nn.Conv2d` module to input a convolution. Then we can apply our non-linear activation function $\tanh$ just like before. We only flatten the tensor once we are ready to use a `nn.Linear` layer to reduce the tensor to a

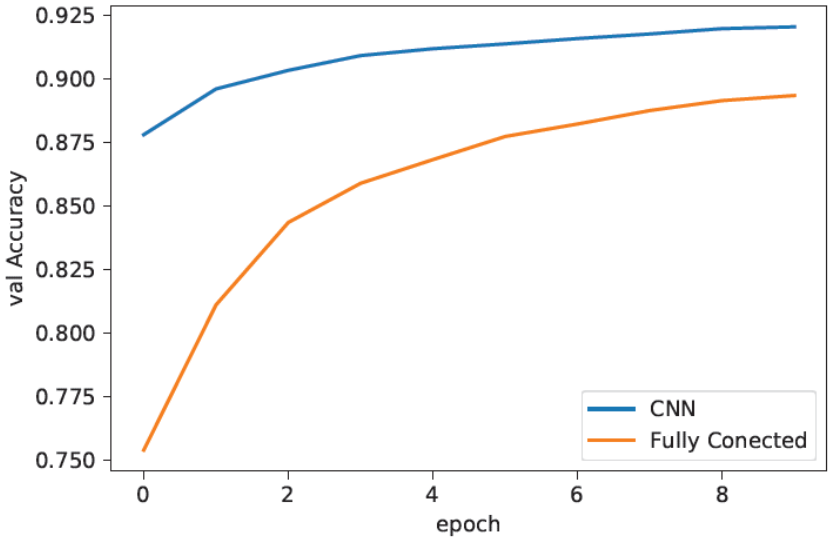
set of predictions for each class. Thats why the `nn.Flatten()` module occurs right before the call to `nn.Linear`

So does a CNN perform better? Lets find out. We can train both a CNN and a fully-connected model, measure accuracy on the test set, and look at the accuracy after each epoch.

```
loss_func = nn.CrossEntropyLoss()
cnn_results = train_simple_network(model_cnn, loss_func, mnist_train_loader,
       val_loader=mnist_test_loader, score_funcs={'Accuracy': accuracy_score}, device=device,
       epochs=10)
fc_results = train_simple_network(model_linear, loss_func, mnist_train_loader,
       val_loader=mnist_test_loader, score_funcs={'Accuracy': accuracy_score}, device=device,
       epochs=10)
```

```
sns.lineplot(x='epoch', y='val Accuracy', data=cnn_results, label='CNN')
sns.lineplot(x='epoch', y='val Accuracy', data=fc_results, label='Fully Conected')
```

```
[20]: <AxesSubplot:xlabel='epoch', ylabel='val Accuracy'>
```



One epoch of training out CNN has *better accuracy* than our fully connected network ever achieves. While our CNN is a about $1.046\times$ slower to train, the results are well worth it. Why did it perform so much better? Because we have given the network information about the problem (convolutions) via the structure of the domain (data are images). This does not mean CNNs are always better. If the assumptions of a CNN are not true, or not accurate, they will not perform well. Remember that the prior belief convolutions impart is that things located near each other are related, but far apart things are not related.

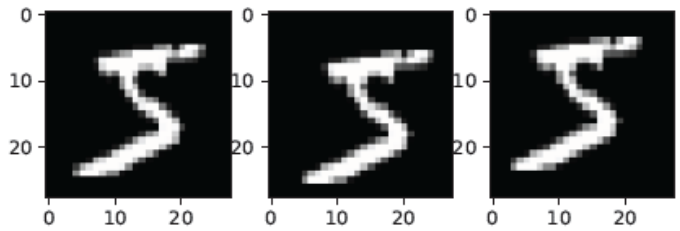
3.5 Pooling And Object Location

Just like with our feed-forward networks, we can make our convolutional networks more powerful by stacking more and more layers with non-linearities inserted inbetween. But before we do that, there is a special type of layer we like to use with CNNs called a *pooling* layer.

Pooling helps us solve a problem, that we aren't fully exploiting the spatial nature of our data. That may seem a confusing statement. We just significantly increased our accuracy with a simple switch to `nn.Conv2d`, and we spent a lot of time talking about how convolutions encode this spatial prior, that they slide a set of weights over the input, applying them at every location. The problem is that we *eventually* switch to using a fully connected layer, at which does not understand the spatial nature of the data. For this reason, the `nn.Linear` layer will learn to look for values at *very specific* locations.

This is not a huge problem for MNIST because all of the digits are aligned so that they are in the center of the image. But imagine you have a digit that is not perfectly center-aligned with your image. This is a very real potential problem. Pooling helps us solve this problem. Lets quickly grab an image from the MNIST dataset, and create two altered versions by just *ever so slightly* moving the content up/down by one pixel.

```
img_indx =0
img, correct_class = mnist_data_train[img_indx]
img = img[0,:]
#move to the lower right, then upper left
img_lr = np.roll(np.roll(img,1, axis=1),1, axis=0)
img_ul = np.roll(np.roll(img, -1, axis=1), -1, axis=0)
#plot the images
f, axarr = plt.subplots(1,3)
axarr[0].imshow(img, cmap='gray')
axarr[1].imshow(img_lr, cmap='gray')
axarr[2].imshow(img_ul, cmap='gray')
plt.show()
```



Clearly, all three versions of the above image are the same digit. It does not matter that we shifted the content up or down, left or right, by just a few pixels. *But our model does not know this*. If we classify different versions of the above image, there is a good chance we get it wrong.

Lets quickly put this model into `eval()` mode, and write a quick function to get the predictions for a single image. That happens with the below `pred` function, that takes an image as input.

```
#eval mode since we are not training
model = model_cnn.cpu().eval()

def pred(model, img):
    with torch.no_grad():#Always turn off gradients when evaluating
        w, h = img.shape#Whats the width/height of the image
        if not isinstance(img, torch.Tensor):
            img = torch.tensor(img)
        x = img.reshape(1,-1,w,h)#reshape it as (B, C, W, H)
        logits = model(x) #Get the logits
        y_hat = F.softmax(logits, dim=1)#Turn into probabilities
        return y_hat.numpy().flatten()#convert prediction to numpy array.
```

This is a simple way to apply a model to *single* images at a time. PyTorch always expects things to be in batches, so we needed to reshape the input to have a batch dimension, which is equal to 1 since there are no other images at a time. The `if not isinstance` check is some defensive code you can add to make sure your code works for both NumPy and PyTorch input tensors. Also remember that the `CrossEntropy` loss we used handles softmax implicitly. So when we use a model trained with `CrossEntropy`, we need to call `F.softmax` ourselves to transform the outputs into probabilities.

With that our of the way, we can get predictions for all three images to see if tiny changes to the image can have a big change on the network's prediction. Remember, each image differs by shifting the image to the lower right or upper left by one pixel. So intuitively, we would expect very little to change.

```
img_pred = pred(model, img)
img_lr_pred = pred(model, img_lr)
img_ul_pred = pred(model, img_ul)

print("Org Img Class  {} Prob:         ".format(correct_class) , img_pred[correct_class])
print("Lower Right Img Class  {} Prob: ".format(correct_class) , img_lr_pred[correct_class])
print("Uper Left Img Class  {} Prob:   ".format(correct_class) , img_ul_pred[correct_class])
```

```
Org Img Class5 Prob:          0.74373084
Lower Right Img Class5 Prob:  0.43279952
Uper Left Img Class5 Prob:    0.24676102
```

Clearly, we want all three of the above examples to receive the same classification. They are *essentially* the same image, yet the outputs swing from a reasonably confident and correct 74.4% down to an erroneous 24.7%. The problem is that a small shift or "translation" of causes the predictions to change significantly.

What we desire is a property called *translation invariance*. To be invariant to property X means that our output does not change based on changes to X. We do not want translations (shifting up/down) to change our decisions. So we want to be translation invariant.

We can obtain some partial translation invariance with an operation known as *pooling*. Specifically, we will look at *max pooling*. What is max pooling? Similar to convolution, we will apply the same function to multiple locations in an image. We will generally stick to even sized pooling filters. As the name would imply, we slide the `max` function function around the image. You could again describe this as having a kernel size K, which is the size of the window to select the maximum from. The big difference here is that we move the `max` function by K pixels at a time, where we did only 1 pixel at a time when performing a convolution. Lets see how that works in Figure 3.14.

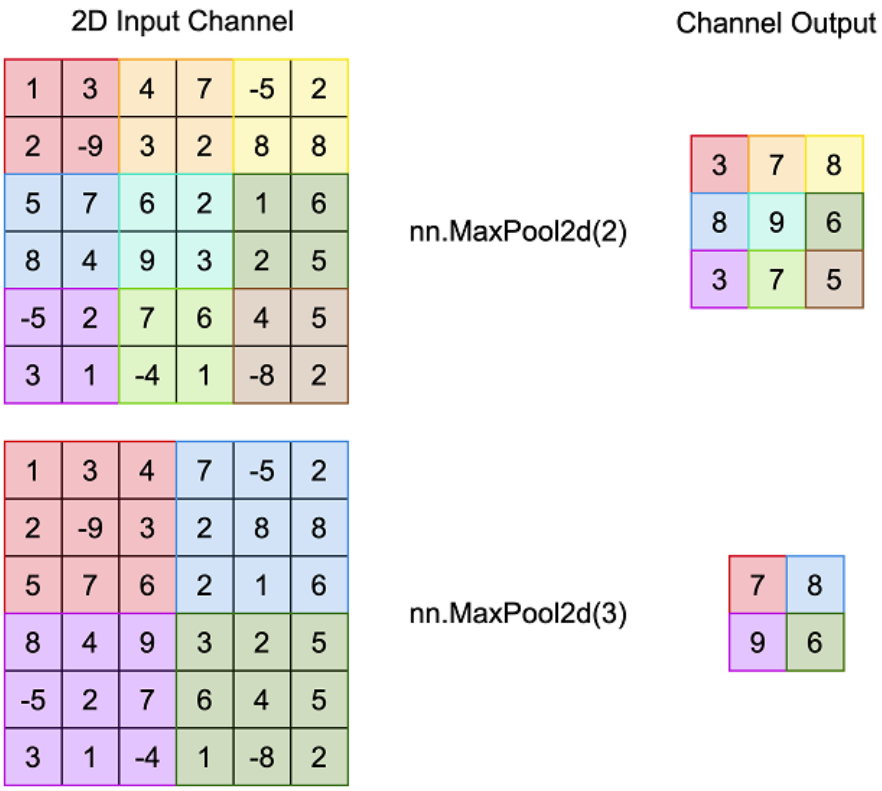


Figure 3.14: Example of max pooling for ($K = 2$) (top) and ($K = 3$) (bottom). Each region of the input (left) has a color indicating the group of pixels participating in the pooling, and the output (right) shows which value was selected from the input region. Note that the output is smaller than the input by a factor of K.

The choice on how many pixels to "slide" by is called the *stride*. By default, practitioners tend to use `stride=1` for convolutions so that *every* possible location is evaluated. We use `stride=K` for pooling so that the input is shrunk by a factor of K. For any operation, if you use a `stride=Z` for any (positive integer) value of Z, the result will be smaller by a factor of Z along each dimension.

The intuition behind pooling is that it gives us more robustness to slight changes in values. Consider the first image above, if your shifted every value over to the right by one position. Five of the nice output values*would not change*, giving it a minor degree of *invariance* to *translating* the image by one pixel. This is not perfect, but it helps reduce the impact of such alterations. If we accumulate this effect through multiple rounds of pooling, we can make the effect stronger.

Just like before, PyTorch provides a `nn.MaxPool1d`, `nn.MaxPool2d`, and `nn.MaxPool3d`, for almost all your needs. It takes the kernel size as input, which is also the stride. Having a stride of K means we shrink the size of each shape dimension by a factor of K. So if our input had a shape of (B, C, W, H), the output of `nn.MaxPool2d(K)` will be a shape of $(B, C, W/K, H/K)$. Since C remains the same, that means We apply this simple operation to every channel independently. If we do max-pooling with a 2×2 filter (the norm for most applications), we end up with an image 1/4 the size (half as many rows, and half as many columns).

3.5.1 CNNs with Max Pooling

Its easy to add pooling into our model's defintion. Just insert `nn.MaxPool2d(2)` into the `nn.Sequential` and it will be in there. But *where* exactly should we use max pooling? First, lets talk bout *how many times* you should apply max pooling. Every time pooling is applied, we shrink the width (and height if 2D) by a factor of K. That means n rounds of pooling means shrinking by a factor or K^n, so that will make the image very small very quickly. For MNIST, we only have 28 pixels of width, so we can do at most 4 rounds of pooling with a size of $K = 2$. Thats because 5 rounds would give us $28/2^5 = 28/32$, which is *less than a pixel of output*.

So does 4 rounds of pooling make more sense? Try to imagine it was a problem *you* were asked to solve. 4 rounds of pooling means shrinking the image to just $28/2^4 = 28/16 = 1.75$ pixels tall. If you could not hazard a guess at what digit was being represent with 1.75 pixels, your CNN probably can't either. Visually shrinking your data down is a good way to estimate the maximum amount of pooling you should apply to most problems. Using two-three rounds of pooling is a good initial lower bound / estimate for images up to 256×256 pixels.

NOTE **Most modern applications of CNNs are on images smaller than 256×256. Thats very large for us to process with modern GPUs and techniques. In practice if your images are larger than that, the first step is to re-size them to have at most 256 pixels along any dimension. If you really *need* to be processing at a larger resolution, you probably want to have someone on your team with more detailed experience. As the tricks to working at that scale are more unique, and often require very expensive hardware.**

Every application of pooling is shrinking the image by a factor of K. This also means the network has less data to process after every round of pooling. If you are working with very large images, this can be used to help reduce the time it takes to train larger models, and reduce the memory costs of training. If neither of these are problems, common practice is to increase the number of filters by $K\times$ after every round of pooling. This is so that the total amount of computation being done at every layer remains roughly the same (i.e., twice as many filters on half as many rows/columns about balances out).

Lets quickly try this out on our MNIST data. The bellow code will define a deeper CNN with multiple layers of convolution, and two rounds of max pooling.

```
model_cnn_pool = nn.Sequential(
  nn.Conv2d(C, filters,3, padding=3//2),
  nn.Tanh(),
  nn.Conv2d(filters, filters,3, padding=3//2),
  nn.Tanh(),
  nn.Conv2d(filters, filters,3, padding=3//2),
  nn.Tanh(),
  nn.MaxPool2d(2),
  nn.Conv2d(filters,2*filters,3, padding=3//2),
  nn.Tanh(),
  nn.Conv2d(2*filters,2*filters,3, padding=3//2),
  nn.Tanh(),
  nn.Conv2d(2*filters,2*filters,3, padding=3//2),
  nn.Tanh(),
  nn.MaxPool2d(2),

  nn.Flatten(),
  #Why did we reduce the number of units into the Linear layer by a factor of 4^2? Because
       pooling a 2x2 grid down to one value means we go from 4 values, down to 1, and we did
       this two times.
```

```
  nn.Linear(2*filters*D//(4**2), classes),
)

cnn_results_with_pool = train_simple_network(model_cnn_pool, loss_func, mnist_train_loader,
       val_loader=mnist_test_loader, score_funcs={'Accuracy': accuracy_score}, device=device,
       epochs=10)
```

Now if we run the same shifted test image through our model, we should see some different results. Max pooling is not a *perfect* solution to the translation problem, so the scores for each shifted version of the image still change. But they don't change *as much*. That is overall a good thing, because it makes our model more *robust* to real life issues. Data will not always be perfectly aligned, so we want to help make the model resilient to the kinds of issues we expect to see in real life test data.

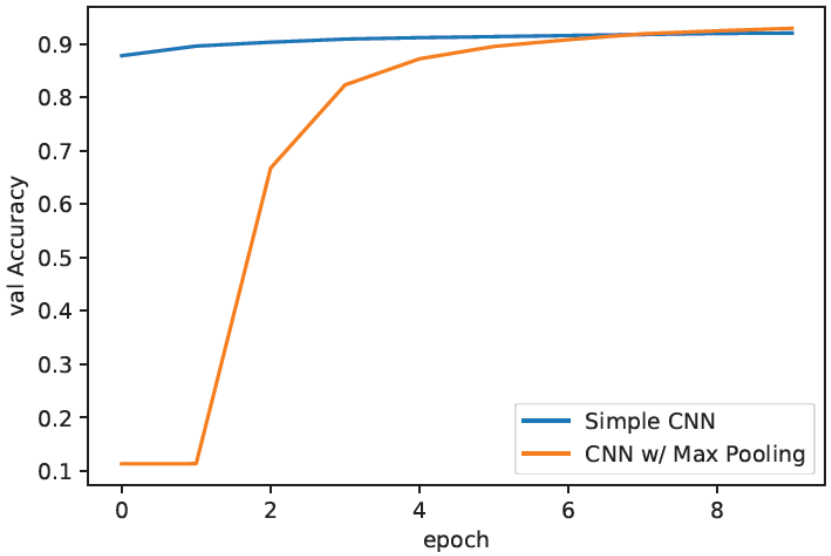
```
model = model_cnn_pool.cpu().eval()
img_pred = pred(model, img)
img_lr_pred = pred(model, img_lr)
img_ul_pred = pred(model, img_ul)

print("Org Img Class  {} Prob:         ".format(correct_class) , img_pred[correct_class])
print("Lower Right Img Class  {} Prob: ".format(correct_class) , img_lr_pred[correct_class])
print("Uper Left Img Class  {} Prob:   ".format(correct_class) , img_ul_pred[correct_class])
```

```
Org Img Class5 Prob:          0.59679925
Lower Right Img Class5 Prob:  0.71586144
Uper Left Img Class5 Prob:    0.31253654
```

Last, we can look at the accuracy of this new and larger network as we trained, which is shown in the below plot. Adding more layers is causing it to take *much* longer for our network to converge, but once it does, it is able to eke out a little better in terms of accuracy.

```
sns.lineplot(x='epoch', y='val Accuracy', data=cnn_results, label='Simple CNN')
sns.lineplot(x='epoch', y='val Accuracy', data=cnn_results_with_pool, label='CNN w/ Max
       Pooling')
```

```
[27]: <AxesSubplot:xlabel='epoch', ylabel='val Accuracy'>
```



It is common for more layers to cause training to take longer to converge *and* longer to process for each epoch. This double whammy is only offset by the fact that more layers, making the models *deeper*, is how we tend to obtain the best possible accuracy. If you continued to train this deeper model for more epochs, you will see it continue to climb a little higher than we could achieve with our first model containing only one `nn.Conv2d` layer.

As we continue further into this book, we will learn about newer and better approaches that help resolve these issues. Making us converge faster and better. But I want to take you through this slower and somewhat more painful path, so that you understand *why* these newer techniques were developed and what problems they are solving for us. This deeper understanding will help you be better prepared for the more advanced techniques we will then be able to tackle by the time you finish this book.

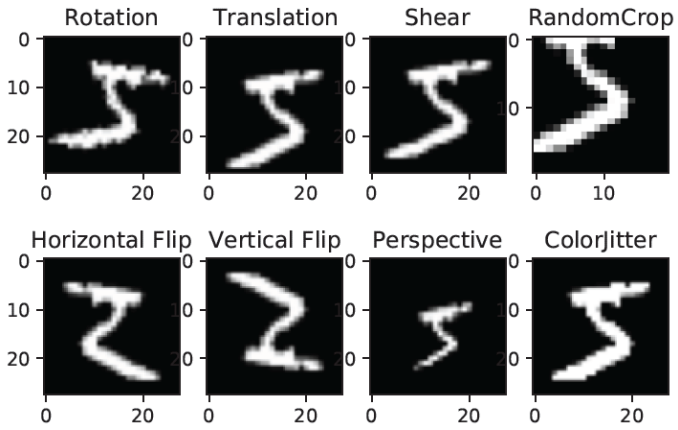
3.6 Data Augmentation

It may seem a little anticlimactic, but you do now know everything you need to start training and building your own CNNs for new problems. Applying really was just replacing `nn.Linear` layers with `nn.Conv2d`, followed by a `nn.Flatten` just before the end. But there is one more big secret to applying CNNs in practice. Thats to use *data augmentation*. In general, neural networks are "data hungry", meaning that they learn best when you have a *huge* amount of *diverse* data. Since data takes time to get, we will instead *augmented* our real data by creating new "fake" data based on our real data.

The idea is simple. If we are working with 2D images, we can do a lot of transforms we could apply to an image that do not change the meaning of it's content, but do alter the pixels themselves. For example, we could rotate the image by a few degrees without really altering what the content means. PyTorch provides a number of transforms in the `torchvision.transforms` package, lets take a look at some of them.

```
#Several built-in transformations, given some agressive values to make their impact more
      obvious.
```

```
sample_transforms = {
    "Rotation" : transforms.RandomAffine(degrees=45),
    "Translation" : transforms.RandomAffine(degrees=0, translate=(0.1,0.1)),
    "Shear": transforms.RandomAffine(degrees=0, shear=45),
    "RandomCrop" : transforms.RandomCrop((20,20)),
    "Horizontal Flip" : transforms.RandomHorizontalFlip(p=1.0),
    "Vertical Flip": transforms.RandomVerticalFlip(p=1.0),
    "Perspective": transforms.RandomPerspective(p=1.0),
    "ColorJitter" : transforms.ColorJitter(brightness=0.9, contrast=0.9)
}
#Convert the Tensor image back to a PIL image using a transform
pil_img = transforms.ToPILImage()(img)
#Plot a randomy application of each transform
f, axarr = plt.subplots(2,4)
for count, (name, t) in enumerate(sample_transforms.items()):
    row = count %4
    col = count //4
    axarr[col,row].imshow(t(pil_img), cmap='gray')
    axarr[col,row].set_title(name)
plt.show()
```



First thing you should notice is that transforms are almost always *randomized*, and every time we apply one it gives a different result. These new results are our augmented data. For example specifying the `degrees=45` is saying that the maximum rotation will be $\pm$ 45° degrees, and the amount applied will be a randomly chosen value in that range. This is done to increase the *diversity* of inputs our model sees. Some transforms do not always apply themselves, and offer the `p` argument to control the probability they are chosen. We set these as `p=1.0` so that you would definitely see them have an impact on the test image. For real use, you probably want to pick a value of `p=0.5` or `p=0.15`. Like many things, the specific value you want to use will depend on your data.

Not every transform should *always* be used. You need to make sure that your transforms preserve the "essence" or "meaning" of your data. For example, we can clearly see that horizontal and vertical flips are not a good idea for the MNIST dataset. A vertical flip applied to the digit 9 could turn it into a 6, and that *changes the meaning of the image*. Your best bet to selecting a good set of transforms is to apply them to data and *look at the results yourself*, if you cant tell what the correct answer is anymore, chances are your CNN can't either.

But once you select a set of transforms you are comfortable with, it is a simple and powerful approach to improved your model's accuracy. Below is a short example of how you can combine the `Compose` transform to create a sequence of transforms in a larger pipeline, which we can apply to augment our training data on the fly. All of the image based datasets PyTorch provides have the `transform` argument so that you can perform these alterations.

```
train_transform = transforms.Compose([
    transforms.RandomAffine(degrees=15, translate=(0.05, 0.05), scale=(0.95, 1.05)),
    transforms.ToTensor(),
])

test_transform = transforms.ToTensor()

mnist_train_t = torchvision.datasets.MNIST("./data", train=True, transform=train_transform)
mnist_test_t = torchvision.datasets.MNIST("./data", train=False, transform=test_transform)
mnist_train_loader_t = DataLoader(mnist_train_t, shuffle=True,  batch_size=B, num_workers=5)
mnist_test_loader_t = DataLoader(mnist_test_t, batch_size=B, num_workers=5)
```

NOTE **A new and important optional argument has been specified in the `DataLoader` class. The `num_workers` flag controls how many threads are used to *pre-load* batches of data for training. While the GPU is busy crunching on a batch of data, each thread can be preparing the next batch so that it is ready to go once the GPU is done. You should always be using this flag because it helps use your GPU more efficiently. Its especially important when you start using transforms, because they CPU will have to spend time processing the images which keeps your GPU idly waiting for longer.**

Now we can redefine the exact same network we used to show max pooling, and call the exact same training method. The data augmentation happens automatically for us by just defining these new data loaders. A simple `ToTensor` transform is all that we use for the test set, because we want the test set to be *deterministic*. That means if we run the same model five times on the test set, we get the same answer five times.

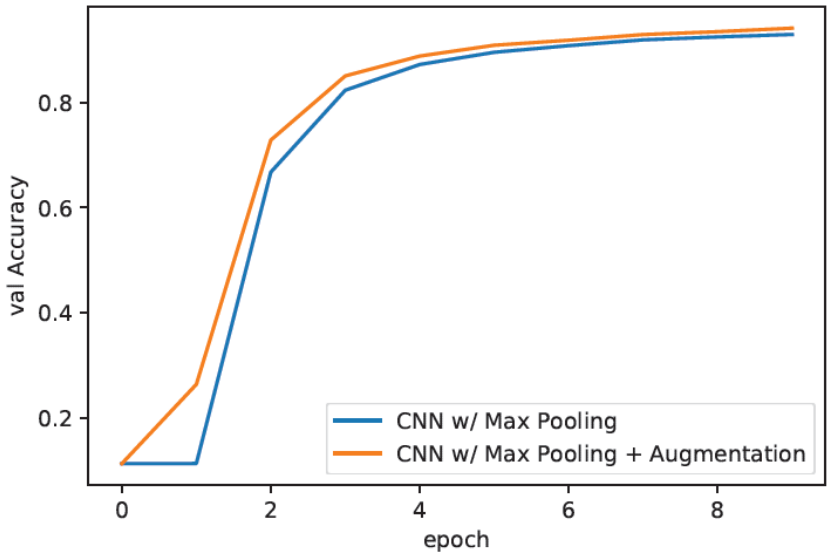
```
model_cnn_pool = nn.Sequential(
  nn.Conv2d(C, filters,3, padding=3//2),
  nn.Tanh(),
  nn.Conv2d(filters, filters,3, padding=3//2),
  nn.Tanh(),
  nn.Conv2d(filters, filters,3, padding=3//2),
  nn.Tanh(),
  nn.MaxPool2d(2),
  nn.Conv2d(filters,2*filters,3, padding=3//2),
  nn.Tanh(),
  nn.Conv2d(2*filters,2*filters,3, padding=3//2),
  nn.Tanh(),
  nn.Conv2d(2*filters,2*filters,3, padding=3//2),
  nn.Tanh(),
  nn.MaxPool2d(2),
  nn.Flatten(),
  nn.Linear(2*filters*D//(4**2), classes),
)

cnn_results_with_pool_augmented = train_simple_network(model_cnn_pool, loss_func,
       mnist_train_loader_t, val_loader=mnist_test_loader_t, score_funcs={'Accuracy':
       accuracy_score}, device=device, epochs=10)
```

Now we can plot the result showing the difference of validation accuracy. With a careful choice of augmentation, we actually helped our model learn faster and converge to a better quality solution, 94% accuracy instead of 93%.

```
sns.lineplot(x='epoch', y='val Accuracy', data=cnn_results_with_pool, label='CNN w/ Max
```

```
        Pooling')
sns.lineplot(x='epoch', y='val Accuracy', data=cnn_results_with_pool_augmented, label='CNN w/
        Max Pooling + Augmentation')
```

```
[31]: <AxesSubplot:xlabel='epoch', ylabel='val Accuracy'>
```



Designing good data augmentation pipelines is the "feature engineering" counterpart to deep learning. If you do it well, it can have a massive impact on your results, and be the difference between success and failure. Because PyTorch uses PIL images as its foundation, you can also write your own custom transforms to add into the pipeline. This is where you can import other tools like `sciktimage` that provide more advanced computer visions transforms that you could apply. The impact of good data augmentation will also grow as we learn how to build more sophisticated networks, and when you work on more complex datasets than MNIST. Data augmentation also increases the value of training for more epochs. Without augmentation, each epoch is revisiting the exact same data. With augmentation, your model sees a different variant of the data that helps it generalize to new data better. Our 10 epochs isn't enough to see the full benefit.

Despite the importance of good data augmentation, you cans till get far in deep learning without it. For the sake of simplifying our examples, we are not going to use data augmentation for most of this book. Part of the reasons for this is to keep examples simpler, without having to explain the choices of data augmentation pipelines for every new dataset. The other reason is that augmentation is *domain specific*. What works for images *only* works for images. You need to come up with a new set of transforms for audio data, and text is a difficult realm to perform augmentation in. Where most of the techniques we will learn in this book can be applied to a fairly broad classes of problems.

3.7 Exercises

Note that these exercises are intentionally *exploratory* in nature. The goal is for you to learn about and discover a number of common trends/properties of working with CNNs through your own code and experience.

1. Try training all of the networks in this chapter for 40 epochs instead of 10. What happens?
2. Load the CIFAR10 dataset from `torchvision`, and try to build your own CNN. Try using 2-10 layers of convolutions, and 0-2 rounds of max-pooling. What seemed to work best?
3. Go through the transforms provided and see which make sense for CIFAR10 via visual inspection. Do any transforms work for CIFAR10 that did not make sense for MNIST?
4. Training a new CIFAR10 model with the transforms you selected. What impact does this have on your accuracy?
5. Try altering the size of the convolutional filters in CIFAR10 and MNIST, what impact does it have?
6. Create a new custom `Shuffle` transform that applies the same, fixed, re-ordering of the pixels in an image. How does using this transform impact your CIFAR10 model? Hint: look at the `Lambda` transfrom to help you implement this.

3.8 Chapter summary

1. To represent images in a PyTorch tensor, we use multiple "channels", where channel describes something of a different nature about the image (e.g., red, green, and blue channels).
2. Convolution is a mathematical operation that takes a kernel, a small tensor, and applies a function over every location in a larger input tensor, to produce an output. This makes them *spatial* in nature.
3. A convolutional layer is learning multiple different kernels to apply to the input, to create multiple different ouptputs.
4. Convolutions don't capture translation (shifts up/down) invariance, and we can use pooling to make our models more robust to translations.
5. Both convolutions and pooling have a "stride" option, which dictates how many pixels we move when "sliding" across the image.
6. When our data is spatial (e.g., images) we can embed a prior (convolutions are spatial) into our network to learn faster and better solutions.
7. We can augment our training data by selecting a set of transformations to apply to the data, which can improve our model's accuracy.