CHAPTER 1

Why Life Science?

While there are many directions that those with a technical inclination and a passion for data can pursue, few areas can match the fundamental impact of biomedical research. The advent of modern medicine has fundamentally changed the nature of human existence. Over the last 20 years, we have seen innovations that have transformed the lives of countless individuals. When it first appeared in 1981, HIV/AIDS was a largely fatal disease. Continued development of antiretroviral therapies has dramatically extended the life expectancy for patients in the developed world. Other diseases, such as hepatitis C, which was considered largely untreatable a decade ago, can now be cured. Advances in genetics are enabling the identification and, hopefully soon, the treatment of a wide array of diseases. Innovations in diagnostics and instrumentation have enabled physicians to specifically identify and target disease in the human body. Many of these breakthroughs have benefited from and will continue to be advanced by computational methods.

Why Deep Learning?

Machine learning algorithms are now a key component of everything from online shopping to social media. Teams of computer scientists are developing algorithms that enable digital assistants such as the Amazon Echo or Google Home to understand speech. Advances in machine learning have enabled routine on-the-fly translation of web pages between spoken languages. In addition to machine learning's impact on everyday life, it has impacted many areas of the physical and life sciences. Algorithms are being applied to everything from the detection of new galaxies from telescope images to the classification of subatomic interactions at the Large Hadron Collider.

One of the drivers of these technological advances has been the development of a class of machine learning methods known as deep neural networks. While the technological underpinnings of artificial neural networks were developed in the 1950s and refined in the 1980s, the true power of the technique wasn't fully realized until advances in computer hardware became available over the last 10 years. We will provide a more complete overview of deep neural networks in the next chapter, but it is important to acknowledge some of the advances that have occurred through the application of deep learning:

- Many of the developments in speech recognition that have become ubiquitous in cell phones, computers, televisions, and other internet-connected devices have been driven by deep learning.
- Image recognition is a key component of self-driving cars, internet search, and other applications. Many of the same developments in deep learning that drove consumer applications are now being used in biomedical research, for example, to classify tumor cells into different types.
- Recommender systems have become a key component of the online experience. Companies like Amazon use deep learning to drive their "customers who bought this also bought" approach to encouraging additional purchases. Netflix uses a similar approach to recommend movies that an individual may want to watch. Many of the ideas behind these recommender systems are being used to identify new molecules that may provide starting points for drug discovery efforts.
- Language translation was once the domain of very complex rule-based systems. Over the last few years, systems driven by deep learning have outperformed systems that had undergone years of manual curation. Many of the same ideas are now being used to extract concepts from the scientific literature and alert scientists to journal articles that they may have missed.

These are just a few of the innovations that have come about through the application of deep learning methods. We are at an interesting time when we have a convergence of widely available scientific data and methods for processing that data. Those with the ability to combine data with new methods for learning from patterns in that data can make significant scientific advances.

Contemporary Life Science Is About Data

As mentioned previously, the fundamental nature of life science has changed. The availability of robotics and miniaturized experiments has brought about dramatic increases in the amount of experimental data that can be generated. In the 1980s a biologist would perform a single experiment and generate a single result. This sort of data could typically be manipulated by hand with the possible assistance of a pocket calculator. If we fast-forward to today's biology, we have instrumentation that is capable of generating millions of experimental data points in a day or two. Experiments like gene sequencing, which can generate huge datasets, have become inexpensive and routine.

The advances in gene sequencing have led to the construction of databases that link an individual's genetic code to a multitude of health-related outcomes, including diabetes, cancer, and genetic diseases such as cystic fibrosis. By using computational techniques to analyze and mine this data, scientists are developing an understanding of the causes of these diseases and using this understanding to develop new treatments.

Disciplines that once relied primarily on human observation are now utilizing datasets that simply could not be analyzed manually. Machine learning is now routinely used to classify images of cells. The output of these machine learning models is used to identify and classify cancerous tumors and to evaluate the effects of potential disease treatments.

Advances in experimental techniques have led to the development of several databases that catalog the structures of chemicals and the effects that these chemicals have on a wide range of biological processes or activities. These structure–activity relationships (SARs) form the basis of a field known as chemical informatics, or *cheminformatics*. Scientists mine these large datasets and use the data to build predictive models that will drive the next generation of drug development.

With these large amounts of data comes a need for a new breed of scientist who is comfortable in both the scientific and computational domains. Those with these hybrid capabilities have the potential to unlock structure and trends in large datasets and to make the scientific discoveries of tomorrow.

What Will You Learn?

In the first few chapters of this book, we provide an overview of deep learning and how it can be applied in the life sciences. We begin with machine learning, which has been defined as "the science (and art) of programming computers so that they can learn from data."[1]

Chapter 2 provides a brief introduction to deep learning. We begin with an example of how this type of machine learning can be used to perform a simple task like linear regression, and progress to more sophisticated models that are commonly used to solve real-world problems in the life sciences. Machine learning typically proceeds by initially splitting a dataset into a training set that is used to generate a model and a test set that is used to assess the performance of the model. In Chapter 2 we discuss

1 Furbush, James. "Machine Learning: A Quick and Simple Definition." *https://www.oreilly.com/ideas/machine-learning-a-quick-and-simple-definition*. 2018.

CHAPTER 2

Introduction to Deep Learning

The goal of this chapter is to introduce the basic principles of deep learning. If you already have lots of experience with deep learning, you should feel free to skim this chapter and then go on to the next. If you have less experience, you should study this chapter carefully as the material it covers will be essential to understanding the rest of the book.

In most of the problems we will discuss, our task will be to create a mathematical function:

$$\mathbf{y} = f(\mathbf{x})$$

Notice that **x** and **y** are written in bold. This indicates they are vectors. The function might take many numbers as input, perhaps thousands or even millions, and it might produce many numbers as outputs. Here are some examples of functions you might want to create:

- **x** contains the colors of all the pixels in an image. $f(\mathbf{x})$ should equal 1 if the image contains a cat and 0 if it does not.
- The same as above, except $f(\mathbf{x})$ should be a vector of numbers. The first element indicates whether the image contains a cat, the second whether it contains a dog, the third whether it contains an airplane, and so on for thousands of types of objects.
- **x** contains the DNA sequence for a chromosome. **y** should be a vector whose length equals the number of bases in the chromosome. Each element should equal 1 if that base is part of a region that codes for a protein, or 0 if not.
- **x** describes the structure of a molecule. (We will discuss various ways of representing molecules in later chapters.) **y** should be a vector where each element

describes some physical property of the molecule: how easily it dissolves in water, how strongly it binds to some other molecule, and so on.

As you can see, $f(\mathbf{x})$ could be a very, very complicated function! It usually takes a long vector as input and tries to extract information from it that is not at all obvious just from looking at the input numbers.

The traditional approach to solving this problem is to design a function by hand. You would start by analyzing the problem. What patterns of pixels tend to indicate the presence of a cat? What patterns of DNA tend to distinguish coding regions from noncoding ones? You would write computer code to recognize particular types of features, then try to identify combinations of features that reliably produce the result you want. This process is slow and labor-intensive, and depends heavily on the expertise of the person carrying it out.

Machine learning takes a totally different approach. Instead of designing a function by hand, you allow the computer to learn its own function based on data. You collect thousands or millions of images, each labeled to indicate whether it includes a cat. You present all of this training data to the computer, and let it search for a function that is consistently close to 1 for the images with cats and close to 0 for the ones without.

What does it mean to "let the computer search for a function"? Generally speaking, you create a *model* that defines some large class of functions. The model includes *parameters*, variables that can take on any value. By choosing the values of the parameters, you select a particular function out of all the many functions in the class defined by the model. The computer's job is to select values for the parameters. It tries to find values such that, when your training data is used as input, the output is as close as possible to the corresponding targets.

Linear Models

One of the simplest models you might consider trying is a linear model:

$$\mathbf{y} = \mathbf{M}\mathbf{x} + \mathbf{b}$$

In this equation, **M** is a matrix (sometimes referred to as the "weights") and **b** is a vector (referred to as the "biases"). Their sizes are determined by the numbers of input and output values. If **x** has length T and you want **y** to have length S, then **M** will be an S × T matrix and **b** will be a vector of length S. Together, they make up the parameters of the model. This equation simply says that each output component is a linear combination of the input components. By setting the parameters (**M** and **b**), you can choose any linear combination you want for each component.

This was one of the very earliest machine learning models. It was introduced back in 1957 and was called a *perceptron*. The name is an amazing piece of marketing: it has a science fiction sound to it and seems to promise wonderful things, when in fact it is nothing more than a linear transform. In any case, the name has managed to stick for more than half a century.

The linear model is very easy to formulate in a completely generic way. It has exactly the same form no matter what problem you apply it to. The only differences between linear models are the lengths of the input and output vectors. From there, it is just a matter of choosing the parameter values, which can be done in a straightforward way with generic algorithms. That is exactly what we want for machine learning: a model and algorithms that are independent of what problem you are trying to solve. Just provide the training data, and parameters are automatically determined that transform the generic model into a function that solves your problem.

Unfortunately, linear models are also very limited. As demonstrated in Figure 2-1, a linear model (in one dimension, that means a straight line) simply cannot fit most real datasets. The problem becomes even worse when you move to very high-dimensional data. No linear combination of pixel values in an image will reliably identify whether the image contains a cat. The task requires a much more complicated nonlinear model. In fact, any model that solves that problem will necessarily be *very* complicated and *very* nonlinear. But how can we formulate it in a generic way? The space of all possible nonlinear functions is infinitely complex. How can we define a model such that, just by choosing values of parameters, we can create almost any nonlinear function we are ever likely to want?

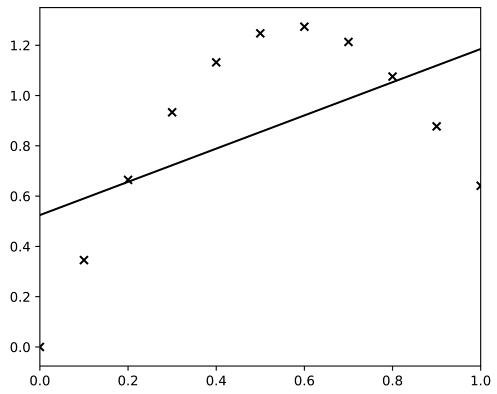

Figure 2-1. A linear model cannot fit data points that follow a curve. This requires a nonlinear model.

Multilayer Perceptrons

A simple approach is to stack multiple linear transforms, one after another. For example, we could write:

$$\mathbf{y} = \mathbf{M}_2\varphi(\mathbf{M}_1\mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$$

Look carefully at what we have done here. We start with an ordinary linear transform, $\mathbf{M}_1\mathbf{x} + \mathbf{b}_1$. We then pass the result through a nonlinear function $\varphi(x)$, and then apply a second linear transform to the result. The function $\varphi(x)$, which is known as the *activation function*, is an essential part of what makes this work. Without it, the model would still be linear, and no more powerful than the previous one. A linear combination of linear combinations is itself nothing more than a linear combination of the original inputs! By inserting a nonlinearity, we enable the model to learn a much wider range of functions.

We don't need to stop at two linear transforms. We can stack as many as we want on top of each other:

$$\mathbf{h}_1 = \varphi_1(\mathbf{M}_1\mathbf{x} + \mathbf{b}_1)$$

$$\mathbf{h}_2 = \varphi_2(\mathbf{M}_2\mathbf{h}_1 + \mathbf{b}_2)$$

$$\ldots$$

$$\mathbf{h}_{n-1} = \varphi_{n-1}(\mathbf{M}_{n-1}\mathbf{h}_{n-2} + \mathbf{b}_{n-1})$$

$$\mathbf{y} = \varphi_n(\mathbf{M}_n\mathbf{h}_{n-1} + \mathbf{b}_n)$$

This model is called a *multilayer perceptron*, or MLP for short. The middle steps h_i are called *hidden layers*. The name refers to the fact that they are neither inputs nor outputs, just intermediate values used in the process of calculating the result. Also notice that we have added a subscript to each $\varphi(x)$. This indicates that different layers might use different nonlinearities.

You can visualize this calculation as a stack of layers, as shown in Figure 2-2. Each layer corresponds to a linear transformation followed by a nonlinearity. Information flows from one layer to another, the output of one layer becoming the input to the next. Each layer has its own set of parameters that determine how its output is calculated from its input.

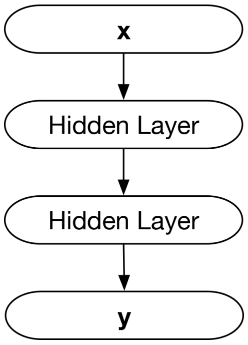

Figure 2-2. A multilayer perceptron, viewed as a stack of layers with information flowing from one layer to the next.

Multilayer perceptrons and their variants are also sometimes called *neural networks*. The name reflects the parallels between machine learning and neurobiology. A biological neuron connects to many other neurons. It receives signals from them, adds the signals together, and then sends out its own signals based on the result. As a very rough approximation, you can think of MLPs as working the same way as the neurons in your brain!

What should the activation function $\varphi(\mathbf{x})$ be? The surprising answer is that it mostly doesn't matter. Of course, that is not entirely true. It obviously does matter, but not as much as you might expect. Nearly any reasonable function (monotonic, reasonably smooth) can work. Lots of different functions have been tried over the years, and although some work better than others, nearly all of them can produce decent results.

The most popular activation function today is probably the *rectified linear unit* (ReLU), $\varphi(x) = \max(0, x)$. If you aren't sure what function to use, this is probably a good default. Other common choices include the *hyperbolic tangent*, $\tanh(x)$, and the *logistic sigmoid*, $\varphi(x) = 1/\left(1 + e^{-x}\right)$. All of these functions are shown in Figure 2-3.

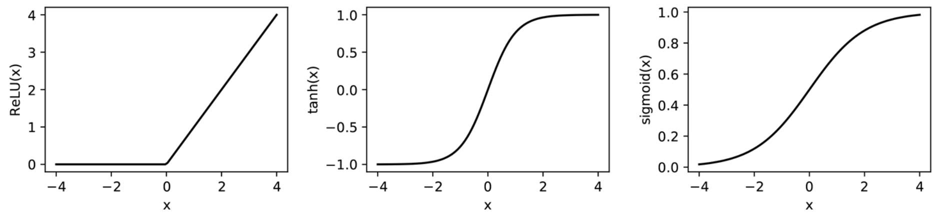

Figure 2-3. Three common activation functions: the rectified linear unit, hyperbolic tangent, and logistic sigmoid.

We also must choose two other properties for an MLP: its *width* and its *depth*. With the simple linear model, we had no choices to make. Given the lengths of **x** and **y**, the sizes of **M** and **b** were completely determined. Not so with hidden layers. Width refers to the size of the hidden layers. We can choose each $\mathbf{h}_i$ to have any length we want. Depending on the problem, you might want them to be much larger or much smaller than the input and output vectors.

Depth refers to the number of layers in the model. A model with only one hidden layer is described as *shallow*. A model with many hidden layers is described as *deep*. This is, in fact, the origin of the term "deep learning"; it simply means "machine learning using models with lots of layers."

Choosing the number and widths of layers in your model involves as much art as science. Or, to put it more formally, "This is still an active field of research." Often it just comes down to trying lots of combinations and seeing what works. There are a few principles that may provide guidance, however, or at least help you understand your results in hindsight:

1. An MLP with one hidden layer is a *universal approximator*.

   This means it can approximate any function at all (within certain fairly reasonable limits). In a sense, you never need more than one hidden layer. That is already enough to reproduce any function you are ever likely to want. Unfortunately, this result comes with a major caveat: the accuracy of the approximation depends on the width of the hidden layer, and you may need a very wide layer to get sufficient accuracy for a given problem. This brings us to the second principle.

2. Deep models tend to require fewer parameters than shallow ones.

   This statement is intentionally somewhat vague. More rigorous statements can be proven for particular special cases, but it does still apply as a general guideline. Here is perhaps a better way of stating it: every problem requires a model with a certain depth to efficiently achieve acceptable accuracy. At shallower depths, the required widths of the layers (and hence the total number of parameters) increase rapidly. This makes it sound like you should always prefer deep models over shallow ones. Unfortunately, it is partly contradicted by the third principle.

3. Deep models tend to be harder to train than shallow ones.

   Until about 2007, most machine learning models were shallow. The theoretical advantages of deep models were known, but researchers were usually unsuccessful at training them. Since then, a series of advances has gradually improved the usefulness of deep models. These include better training algorithms, new types of models that are easier to train, and of course faster computers combined with larger datasets on which to train the models. These advances gave rise to "deep learning" as a field. Yet despite the improvements, the general principle remains true: deeper models tend to be harder to train than shallower ones.

Training Models

This brings us to the next subject: just how do we train a model anyway? MLPs provide us with a (mostly) generic model that can be used for any problem. (We will discuss other, more specialized types of models a little later.) Now we want a similarly generic algorithm to find the optimal values of the model's parameters for a given problem. How do we do that?

The first thing you need, of course, is a collection of data to train it on. This dataset is known as the *training set*. It should consist of a large number of ($\mathbf{x}$,$\mathbf{y}$) pairs, also known as *samples*. Each sample specifies an input to the model, and what you want the model's output to be when given that input. For example, the training set could be a collection of images, along with labels indicating whether or not each image contains a cat.

Next you need to define a loss function $L(\mathbf{y}, \widehat{\mathbf{y}})$, where $\mathbf{y}$ is the actual output from the model and $\widehat{\mathbf{y}}$ is the target value specified in the training set. This is how you measure whether the model is doing a good job of reproducing the training data. It is then averaged over every sample in the training set:

$$\text{average loss} = \frac{1}{N} \sum_{i=1}^{N} L(\mathbf{y}_i, \widehat{\mathbf{y}}_i)$$

$L(\mathbf{y}, \widehat{\mathbf{y}})$ should be small when its arguments are close together and large when they are far apart. In other words, we take every sample in the training set, try using each one as an input to the model, and see how close the output is to the target value. Then we average this over the whole training set.

An appropriate loss function needs to be chosen for each problem. A common choice is the Euclidean distance (also known as the L_2 distance), $L(\mathbf{y}, \widehat{\mathbf{y}}) = \sqrt{\Sigma_i (y_i - \widehat{y}_i)^2}$. (In this expression, y_i means the i'th component of the vector $\mathbf{y}$.) When $\mathbf{y}$ represents a probability distribution, a popular choice is the cross entropy, $L(\mathbf{y}, \widehat{\mathbf{y}}) = -\Sigma_i y_i \log \widehat{y}_i$. Other choices are also possible, and there is no universal "best" choice. It depends on the details of your problem.

Now that we have a way to measure how well the model works, we need a way to improve it. We want to search for the parameter values that minimize the average loss over the training set. There are many ways to do this, but most work in deep learning uses some variant of the *gradient descent* algorithm. Let θ represent the set of all parameters in the model. Gradient descent involves taking a series of small steps:

$$\theta \leftarrow \theta - \epsilon \frac{\partial}{\partial \theta} \langle L \rangle$$

where $\langle L \rangle$ is the average loss over the training set. Each step moves a tiny distance in the "downhill" direction. It changes each of the model's parameters by a little bit, with the goal of causing the average loss to decrease. If all the stars align and the phase of the moon is just right, this will eventually produce parameters that do a good job of solving your problem. ϵ is called the *learning rate*, and it determines how much the parameters change on each step. It needs to be chosen very carefully: too small a value will cause learning to be very slow, while too large a value will prevent the algorithm from learning at all.

This algorithm really does work, but it has a serious problem. For every step of gradient descent, we need to loop over every sample in the training set. That means the time required to train the model is proportional to the size of the training set! Suppose that you have one million samples in the training set, that computing the gradient of the loss for one sample requires one million operations, and that it takes one million steps to find a good model. (All of these numbers are fairly typical of real deep learning applications.) Training will then require *one quintillion* operations. That takes quite a long time, even on a fast computer.

Fortunately, there is a better solution: estimate $\langle L \rangle$ by averaging over a much smaller number of samples. This is the basis of the *stochastic gradient descent* (SGD) algorithm. For every step, we take a small set of samples (known as a *batch*) from the training set and compute the gradient of the loss function, averaged over only the samples in the batch. We can view this as an estimate of what we would have gotten if we had averaged over the entire training set, although it may be a very noisy estimate. We perform a single step of gradient descent, then select a new batch of samples for the next step.

This algorithm tends to be much faster. The time required for each step depends only on the size of each batch, which can be quite small (often on the order of 100 samples) and is independent of the size of the training set. The disadvantage is that each step does a less good job of reducing the loss, because it is based on a noisy estimate of the gradient rather than the true gradient. Still, it leads to a much shorter training time overall.

Most optimization algorithms used in deep learning are based on SGD, but there are many variations that improve on it in different ways. Fortunately, you can usually treat these algorithms as black boxes and trust them to do the right thing without understanding all the details of how they work. Two of the most popular algorithms used today are called Adam and RMSProp. If you are in doubt about what algorithm to use, either one of those will probably be a reasonable choice.

Validation

Suppose you have done everything described so far. You collected a large set of training data. You selected a model, then ran a training algorithm until the loss became very small. Congratulations, you now have a function that solves your problem!

Right?

Sorry, it's not that simple! All you really know for sure is that the function works well *on the training data*. You might hope it will also work well on other data, but you certainly can't count on it. Now you need to validate the model to see whether it works on data that it hasn't been specifically trained on.

To do this you need a second dataset, called the *test set*. It has exactly the same form as the training set, a collection of $(\mathbf{x}, \mathbf{y})$ pairs, but the two should have no samples in common. You train the model on the training set, then test it on the test set. This brings us to one of the most important principles in machine learning:

- You must not use the test set in any way while designing or training the model.

In fact, it is best if you never even look at the data in the test set. Test set data is only for testing the fully trained model to find out how well it works. If you allow the test set to influence the model in any way, you risk getting a model that works better on the test set than on other data that was not involved in creating the model. It ceases to be a true test set, and becomes just another type of training set.

This is connected to the mathematical concept of *overfitting*. The training data is supposed to be representative of a much larger data distribution, the set of all inputs you might ever want to use the model on. But you can't train it on all possible inputs. You can only create a finite set of training samples, train the model on those, and hope it learns general strategies that work equally well on other samples. Overfitting is what happens when the training picks up on specific features of the training samples, such that the model works better on them than it does on other samples.

Regularization

Overfitting is a major problem for anyone who uses machine learning. Given that, you won't be surprised to learn that lots of techniques have been developed for avoiding it. These techniques are collectively known as *regularization*. The goal of any regularization technique is to avoid overfitting and produce a trained model that works well on any input, not just the particular inputs that were used for training.

Before we discuss particular regularization techniques, there are two very important points to understand about it.

First, the best way to avoid overfitting is *almost always* to get more training data. The bigger your training set, the better it represents the "true" data distribution, and the less likely the learning algorithm is to overfit. Of course, that is sometimes impossible: maybe you simply have no way to get more data, or the data may be very expensive to collect. In that case, you just have to do the best you can with the data you have, and if overfitting is a problem, you will have to use regularization to avoid it. But more data will probably lead to a better result than regularization.

Second, there is no universally "best" way to do regularization. It all depends on the problem. After all, the training algorithm doesn't know that it's overfitting. All it knows about is the training data. It doesn't know how the true data distribution differs from the training data, so the best it can do is produce a model that works well on the training set. If that isn't what you want, it's up to you to tell it.

That is the essence of any regularization method: biasing the training process to prefer certain types of models over others. You make assumptions about what properties a "good" model should have, and how it differs from an overfit one, and then you tell the training algorithm to prefer models with those properties. Of course, those assumptions are often implicit rather than explicit. It may not be obvious what assumptions you are making by choosing a particular regularization method. But they are always there.

One of the simplest regularization methods is just to train the model for fewer steps. Early in training, it tends to pick up on coarse properties of the training data that likely apply to the true distribution. The longer it runs, the more likely it is to start picking up on fine details of particular training samples. By limiting the number of training steps, you give it less opportunity to overfit. More formally, you are really assuming that "good" parameter values should not be too different from whatever values you start training from.

Another method is to restrict the magnitude of the parameters in the model. For example, you might add a term to the loss function that is proportional to $|\theta|^2$, where θ is a vector containing all of the model's parameters. By doing this, you are assuming that "good" parameter values should not be any larger than necessary. It reflects the fact that overfitting often (though not always) involves some parameters becoming very large.

A very popular method of regularization is called *dropout*. It involves doing something that at first seems ridiculous, but actually works surprisingly well. For each hidden layer in the model, you randomly select a subset of elements in the output vector h_i and set them to 0. On every step of gradient descent, you pick a different random subset of elements. This might seem like it would just break the model: how can you expect it to work when internal calculations keep randomly getting set to 0? The mathematical theory for why dropout works is a bit complicated. Very roughly speaking, by using dropout you are assuming that no individual calculation within the model should be too important. You should be able to randomly remove any individual calculation, and the rest of the model should continue to work without it. This forces it to learn redundant, highly distributed representations of data that make overfitting unlikely. If you are unsure of what regularization method to use, dropout is a good first thing to try.

Hyperparameter Optimization

By now you have probaly noticed that there are a lot of choices to make, even when using a supposedly generic model with a "generic" learning algorithm. Examples include:

- The number of layers in the model
- The width of each layer
- The number of training steps to perform
- The learning rate to use during training
- The fraction of elements to set to 0 when using dropout

These options are called *hyperparameters*. A hyperparameter is any aspect of the model or training algorithm that must be set in advance rather than being learned by the training algorithm. But how are you supposed to choose them—and isn't the whole point of machine learning to select settings automatically based on data?

This brings us to the subject of *hyperparameter optimization*. The simplest way of doing it is just to try lots of values for each hyperparameter and see what works best. This becomes very expensive when you want to try lots of values for lots of hyperparameters, so there are more sophisticated approaches, but the basic idea remains the same: try different combinations and see what works best.

But how can you tell what works best? The simplest answer would be to just see what produces the lowest value of the loss function (or some other measure of accuracy) on the training set. But remember, that isn't what we really care about. We want to minimize error on the test set, not the training set. This is especially important for hyperparameters that affect regularization, such as the dropout rate. A low training set error might just mean the model is overfitting, optimizing for the precise details of the training data. So instead we want to try lots of hyperparameter values, then use the ones that minimize the loss on the test set.

But we mustn't do that! Remember: you must not use the test set in any way while designing or training the model. Its job is to tell you how well the model is likely to work on new data it has never seen before. Just because a particular set of hyperparameters happens to work best on the test set doesn't guarantee those values will always work best. We must not allow the test set to influence the model, or it is no longer an unbiased test set.

The solution is to create yet another dataset, which is called the *validation set*. It must not share any samples with either the training set or the test set. The full procedure now works as follows:

1. For each set of hyperparameter values, train the model on the training set, then compute the loss on the validation set.
2. Whichever set of hyperparameters give the lowest loss on the validation set, accept them as your final model.
3. Evaluate that final model on the test set to get an unbiased measure of how well it works.

Other Types of Models

This still leaves one more decision you need to make, and it is a huge subject in itself: what kind of model to use. Earlier in this chapter we introduced multilayer perceptrons. They have the advantage of being a generic class of models that can be applied to many different problems. Unfortunately, they also have serious disadvantages. They require a huge number of parameters, which makes them very susceptible to overfitting. They become difficult to train when they have more than one or two hidden layers. In many cases, you can get a better result by using a less generic model that takes advantage of specific features of your problem.

Much of the content of this book consists of discussing particular types of models that are especially useful in the life sciences. Those can wait until later chapters. But for the purposes of this introduction, there are two very important classes of models we should discuss that are widely used in many different fields. They are called convolutional neural networks and recurrent neural networks.

Convolutional Neural Networks

Convolutional neural networks (CNNs for short) were one of the very first classes of deep models to be widely used. They were developed for use in image processing and computer vision. They remain an excellent choice for many kinds of problems that involve continuous data sampled on a rectangular grid: audio signals (1D), images (2D), volumetric MRI data (3D), and so on.

They are also a class of models that truly justify the term "neural network." The design of CNNs was originally inspired by the workings of the feline visual cortex. (Cats have played a central role in deep learning from the dawn of the field.) Research performed from the 1950s to the 1980s revealed that vision is processed through a series of layers. Each neuron in the first layer takes input from a small region of the visual field (its *receptive field*). Different neurons are specialized to detect particular local patterns or features, such as vertical or horizontal lines. Cells in the second layer take input from local clusters of cells in the first layer, combining their signals to detect more complicated patterns over a larger receptive field. Each layer can be viewed as a new representation of the original image, described in terms of larger and more abstract patterns than the ones in the previous layer.

CNNs mirror this design, sending an input image through a series of layers. In that sense, they are just like MLPs, but the structure of each layer is very different. MLPs use *fully connected layers*. Every element of the output vector depends on every element of the input vector. CNNs use *convolutional layers* that take advantage of spatial locality. Each output element corresponds to a small region of the image, and only depends on the input values in that region. This enormously reduces the number of parameters defining each layer. In effect, it assumes that most elements of the weight matrix $\mathbf{M}_i$ are 0, since each output element only depends on a small number of input elements.

Convolutional layers take this a step further: they assume the parameters are the same for *every local region of the image*. If a layer uses one set of parameters to detect horizontal lines at one location in the image, it also uses exactly the same parameters to detect horizontal lines everywhere else in the image. This makes the number of parameters for the layer independent of the size of the image. All it has to learn is a single *convolutional kernel* that defines how output features are computed from any local region of the image. That local region is often very small, perhaps 5 by 5 pixels. In that case, the number of parameters to learn is only 25 times the number of output features for each region. This is tiny compared to the number in a fully connected layer, making CNNs much easier to train and much less susceptible to overfitting than MLPs.

Recurrent Neural Networks

Recurrent neural networks (RNNs for short) are a bit different. They are normally used to process data that takes the form of a sequence of elements: words in a text document, bases in a DNA molecule, etc. The elements in the sequence are fed into the network's input one at a time. But then the network does something very different: the output from each layer is fed back into its own input on the next step! This allows RNNs to have a sort of memory. When an element (word, DNA base, etc.) from the sequence is fed into the network, the input to each layer depends on that element, but also on all of the previous elements (Figure 2-4).

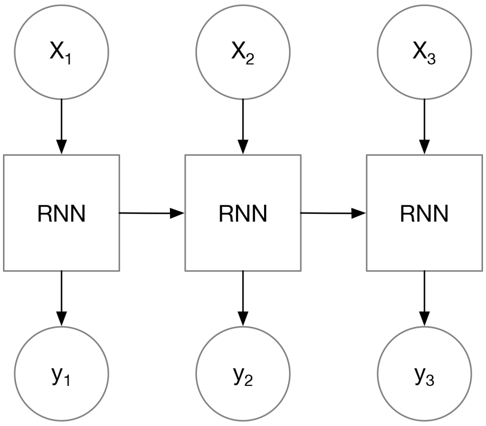

Figure 2-4. A recurrent neural network. As each element (x_1, x_2, ...) of the sequence is fed into the input, the output (y_1, y_2, ...) depends both on the input element and on the RNN's own output during the previous step.

So, the input to a recurrent layer has two parts: the regular input (that is, the output from the previous layer in the network) and the recurrent input (which equals its own output from the previous step). It then needs to calculate a new output based on those inputs. In principle you could use a fully connected layer, but in practice that usually doesn't work very well. Researchers have developed other types of layers that work much better in RNNs. The two most popular ones are called the *gated recurrent unit* (GRU) and the *long short-term memory* (LSTM). Don't worry about the details for now; just remember that if you are creating an RNN, you should usually build it out of one of those types of layers.

Having memory makes RNNs fundamentally different from the other models we have discussed. With a CNN or MLP, you simply feed a value into the network's input and get a different value out. The output is entirely determined by the input. Not so with an RNN. The model has its own internal state, composed of the outputs of all its layers from the most recent step. Each time you feed a new value into the model, the output depends not just on the input value but also on the internal state. Likewise, the internal state is altered by each new input value. This makes RNNs very powerful, and allows them to be used for lots of different applications.