Chapter 5

Machine Learning Basics

Deep learning is a specific kind of machine learning. In order to understand deep learning well, one must have a solid understanding of the basic principles of machine learning. This chapter provides a brief course in the most important general principles that will be applied throughout the rest of the book. Novice readers or those who want a wider perspective are encouraged to consider machine learning textbooks with a more comprehensive coverage of the fundamentals, such as Murphy (2012) or Bishop (2006). If you are already familiar with machine learning basics, feel free to skip ahead to section 5.11. That section covers some perspectives on traditional machine learning techniques that have strongly influenced the development of deep learning algorithms.

We begin with a definition of what a learning algorithm is, and present an example: the linear regression algorithm. We then proceed to describe how the challenge of fitting the training data differs from the challenge of finding patterns that generalize to new data. Most machine learning algorithms have settings called hyperparameters that must be determined external to the learning algorithm itself; we discuss how to set these using additional data. Machine learning is essentially a form of applied statistics with increased emphasis on the use of computers to statistically estimate complicated functions and a decreased emphasis on proving confidence intervals around these functions; we therefore present the two central approaches to statistics: frequentist estimators and Bayesian inference. Most machine learning algorithms can be divided into the categories of supervised learning and unsupervised learning; we describe these categories and give some examples of simple learning algorithms from each category. Most deep learning algorithms are based on an optimization algorithm called stochastic gradient descent. We describe how to combine various algorithm components such as

an optimization algorithm, a cost function, a model, and a dataset to build a machine learning algorithm. Finally, in section 5.11, we describe some of the factors that have limited the ability of traditional machine learning to generalize. These challenges have motivated the development of deep learning algorithms that overcome these obstacles.

5.1 Learning Algorithms

A machine learning algorithm is an algorithm that is able to learn from data. But what do we mean by learning? Mitchell (1997) provides the definition "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." One can imagine a very wide variety of experiences E, tasks T, and performance measures P, and we do not make any attempt in this book to provide a formal definition of what may be used for each of these entities. Instead, the following sections provide intuitive descriptions and examples of the different kinds of tasks, performance measures and experiences that can be used to construct machine learning algorithms.

5.1.1 The Task, T

Machine learning allows us to tackle tasks that are too difficult to solve with fixed programs written and designed by human beings. From a scientific and philosophical point of view, machine learning is interesting because developing our understanding of machine learning entails developing our understanding of the principles that underlie intelligence.

In this relatively formal definition of the word "task," the process of learning itself is not the task. Learning is our means of attaining the ability to perform the task. For example, if we want a robot to be able to walk, then walking is the task. We could program the robot to learn to walk, or we could attempt to directly write a program that specifies how to walk manually.

Machine learning tasks are usually described in terms of how the machine learning system should process an **example**. An example is a collection of **features** that have been quantitatively measured from some object or event that we want the machine learning system to process. We typically represent an example as a vector $\boldsymbol{x} \in \mathbb{R}^n$ where each entry x_i of the vector is another feature. For example, the features of an image are usually the values of the pixels in the image.

Many kinds of tasks can be solved with machine learning. Some of the most common machine learning tasks include the following:

- **Classification**: In this type of task, the computer program is asked to specify which of k categories some input belongs to. To solve this task, the learning algorithm is usually asked to produce a function $f : \mathbb{R}^n \to \{1, \ldots, k\}$. When $y = f(\boldsymbol{x})$, the model assigns an input described by vector $\boldsymbol{x}$ to a category identified by numeric code y. There are other variants of the classification task, for example, where f outputs a probability distribution over classes. An example of a classification task is object recognition, where the input is an image (usually described as a set of pixel brightness values), and the output is a numeric code identifying the object in the image. For example, the Willow Garage PR2 robot is able to act as a waiter that can recognize different kinds of drinks and deliver them to people on command (Goodfellow *et al.*, 2010). Modern object recognition is best accomplished with deep learning (Krizhevsky *et al.*, 2012; Ioffe and Szegedy, 2015). Object recognition is the same basic technology that allows computers to recognize faces (Taigman *et al.*, 2014), which can be used to automatically tag people in photo collections and allow computers to interact more naturally with their users.

- **Classification with missing inputs**: Classification becomes more challenging if the computer program is not guaranteed that every measurement in its input vector will always be provided. In order to solve the classification task, the learning algorithm only has to define a *single* function mapping from a vector input to a categorical output. When some of the inputs may be missing, rather than providing a single classification function, the learning algorithm must learn a *set* of functions. Each function corresponds to classifying $\boldsymbol{x}$ with a different subset of its inputs missing. This kind of situation arises frequently in medical diagnosis, because many kinds of medical tests are expensive or invasive. One way to efficiently define such a large set of functions is to learn a probability distribution over all of the relevant variables, then solve the classification task by marginalizing out the missing variables. With n input variables, we can now obtain all 2^n different classification functions needed for each possible set of missing inputs, but we only need to learn a single function describing the joint probability distribution. See Goodfellow *et al.* (2013b) for an example of a deep probabilistic model applied to such a task in this way. Many of the other tasks described in this section can also be generalized to work with missing inputs; classification with missing inputs is just one example of what machine learning can do.

- **Regression**: In this type of task, the computer program is asked to predict a numerical value given some input. To solve this task, the learning algorithm is asked to output a function $f : \mathbb{R}^n \to \mathbb{R}$. This type of task is similar to classification, except that the format of output is different. An example of a regression task is the prediction of the expected claim amount that an insured person will make (used to set insurance premiums), or the prediction of future prices of securities. These kinds of predictions are also used for algorithmic trading.

- **Transcription**: In this type of task, the machine learning system is asked to observe a relatively unstructured representation of some kind of data and transcribe it into discrete, textual form. For example, in optical character recognition, the computer program is shown a photograph containing an image of text and is asked to return this text in the form of a sequence of characters (e.g., in ASCII or Unicode format). Google Street View uses deep learning to process address numbers in this way (Goodfellow *et al.*, 2014d). Another example is speech recognition, where the computer program is provided an audio waveform and emits a sequence of characters or word ID codes describing the words that were spoken in the audio recording. Deep learning is a crucial component of modern speech recognition systems used at major companies including Microsoft, IBM and Google (Hinton *et al.*, 2012b).

- **Machine translation**: In a machine translation task, the input already consists of a sequence of symbols in some language, and the computer program must convert this into a sequence of symbols in another language. This is commonly applied to natural languages, such as translating from English to French. Deep learning has recently begun to have an important impact on this kind of task (Sutskever *et al.*, 2014; Bahdanau *et al.*, 2015).

- **Structured output**: Structured output tasks involve any task where the output is a vector (or other data structure containing multiple values) with important relationships between the different elements. This is a broad category, and subsumes the transcription and translation tasks described above, but also many other tasks. One example is parsing—mapping a natural language sentence into a tree that describes its grammatical structure and tagging nodes of the trees as being verbs, nouns, or adverbs, and so on. See Collobert (2011) for an example of deep learning applied to a parsing task. Another example is pixel-wise segmentation of images, where the computer program assigns every pixel in an image to a specific category. For

example, deep learning can be used to annotate the locations of roads in aerial photographs (Mnih and Hinton, 2010). The output need not have its form mirror the structure of the input as closely as in these annotation-style tasks. For example, in image captioning, the computer program observes an image and outputs a natural language sentence describing the image (Kiros *et al.*, 2014a,b; Mao *et al.*, 2015; Vinyals *et al.*, 2015b; Donahue *et al.*, 2014; Karpathy and Li, 2015; Fang *et al.*, 2015; Xu *et al.*, 2015). These tasks are called structured output tasks because the program must output several values that are all tightly inter-related. For example, the words produced by an image captioning program must form a valid sentence.

- **Anomaly detection**: In this type of task, the computer program sifts through a set of events or objects, and flags some of them as being unusual or atypical. An example of an anomaly detection task is credit card fraud detection. By modeling your purchasing habits, a credit card company can detect misuse of your cards. If a thief steals your credit card or credit card information, the thief's purchases will often come from a different probability distribution over purchase types than your own. The credit card company can prevent fraud by placing a hold on an account as soon as that card has been used for an uncharacteristic purchase. See Chandola *et al.* (2009) for a survey of anomaly detection methods.

- **Synthesis and sampling**: In this type of task, the machine learning algorithm is asked to generate new examples that are similar to those in the training data. Synthesis and sampling via machine learning can be useful for media applications where it can be expensive or boring for an artist to generate large volumes of content by hand. For example, video games can automatically generate textures for large objects or landscapes, rather than requiring an artist to manually label each pixel (Luo *et al.*, 2013). In some cases, we want the sampling or synthesis procedure to generate some specific kind of output given the input. For example, in a speech synthesis task, we provide a written sentence and ask the program to emit an audio waveform containing a spoken version of that sentence. This is a kind of structured output task, but with the added qualification that there is no single correct output for each input, and we explicitly desire a large amount of variation in the output, in order for the output to seem more natural and realistic.

- **Imputation of missing values**: In this type of task, the machine learning algorithm is given a new example $\boldsymbol{x} \in \mathbb{R}^n$, but with some entries x_i of $\boldsymbol{x}$ missing. The algorithm must provide a prediction of the values of the missing entries.

- **Denoising**: In this type of task, the machine learning algorithm is given in input a *corrupted example* $\tilde{\boldsymbol{x}} \in \mathbb{R}^n$ obtained by an unknown corruption process from a *clean example* $\boldsymbol{x} \in \mathbb{R}^n$. The learner must predict the clean example $\boldsymbol{x}$ from its corrupted version $\tilde{\boldsymbol{x}}$, or more generally predict the conditional probability distribution $p(\boldsymbol{x} \mid \tilde{\boldsymbol{x}})$.

- **Density estimation** or **probability mass function estimation**: In the density estimation problem, the machine learning algorithm is asked to learn a function $p_{\text{model}} : \mathbb{R}^n \rightarrow \mathbb{R}$, where $p_{\text{model}}(\boldsymbol{x})$ can be interpreted as a probability density function (if $\mathbf{x}$ is continuous) or a probability mass function (if $\mathbf{x}$ is discrete) on the space that the examples were drawn from. To do such a task well (we will specify exactly what that means when we discuss performance measures P), the algorithm needs to learn the structure of the data it has seen. It must know where examples cluster tightly and where they are unlikely to occur. Most of the tasks described above require the learning algorithm to at least implicitly capture the structure of the probability distribution. Density estimation allows us to explicitly capture that distribution. In principle, we can then perform computations on that distribution in order to solve the other tasks as well. For example, if we have performed density estimation to obtain a probability distribution $p(\boldsymbol{x})$, we can use that distribution to solve the missing value imputation task. If a value x_i is missing and all of the other values, denoted $\boldsymbol{x}_{-i}$, are given, then we know the distribution over it is given by $p(x_i \mid \boldsymbol{x}_{-i})$. In practice, density estimation does not always allow us to solve all of these related tasks, because in many cases the required operations on $p(\boldsymbol{x})$ are computationally intractable.

Of course, many other tasks and types of tasks are possible. The types of tasks we list here are intended only to provide examples of what machine learning can do, not to define a rigid taxonomy of tasks.

5.1.2 The Performance Measure, P

In order to evaluate the abilities of a machine learning algorithm, we must design a quantitative measure of its performance. Usually this performance measure P is specific to the task T being carried out by the system.

For tasks such as classification, classification with missing inputs, and transcription, we often measure the **accuracy** of the model. Accuracy is just the proportion of examples for which the model produces the correct output. We can

also obtain equivalent information by measuring the **error rate**, the proportion of examples for which the model produces an incorrect output. We often refer to the error rate as the expected 0-1 loss. The 0-1 loss on a particular example is 0 if it is correctly classified and 1 if it is not. For tasks such as density estimation, it does not make sense to measure accuracy, error rate, or any other kind of 0-1 loss. Instead, we must use a different performance metric that gives the model a continuous-valued score for each example. The most common approach is to report the average log-probability the model assigns to some examples.

Usually we are interested in how well the machine learning algorithm performs on data that it has not seen before, since this determines how well it will work when deployed in the real world. We therefore evaluate these performance measures using a **test set** of data that is separate from the data used for training the machine learning system.

The choice of performance measure may seem straightforward and objective, but it is often difficult to choose a performance measure that corresponds well to the desired behavior of the system.

In some cases, this is because it is difficult to decide what should be measured. For example, when performing a transcription task, should we measure the accuracy of the system at transcribing entire sequences, or should we use a more fine-grained performance measure that gives partial credit for getting some elements of the sequence correct? When performing a regression task, should we penalize the system more if it frequently makes medium-sized mistakes or if it rarely makes very large mistakes? These kinds of design choices depend on the application.

In other cases, we know what quantity we would ideally like to measure, but measuring it is impractical. For example, this arises frequently in the context of density estimation. Many of the best probabilistic models represent probability distributions only implicitly. Computing the actual probability value assigned to a specific point in space in many such models is intractable. In these cases, one must design an alternative criterion that still corresponds to the design objectives, or design a good approximation to the desired criterion.

5.1.3 The Experience, E

Machine learning algorithms can be broadly categorized as **unsupervised** or **supervised** by what kind of experience they are allowed to have during the learning process.

Most of the learning algorithms in this book can be understood as being allowed to experience an entire **dataset**. A dataset is a collection of many examples, as

defined in section 5.1.1. Sometimes we will also call examples **data points**.

One of the oldest datasets studied by statisticians and machine learning researchers is the Iris dataset (Fisher, 1936). It is a collection of measurements of different parts of 150 iris plants. Each individual plant corresponds to one example. The features within each example are the measurements of each of the parts of the plant: the sepal length, sepal width, petal length and petal width. The dataset also records which species each plant belonged to. Three different species are represented in the dataset.

Unsupervised learning algorithms experience a dataset containing many features, then learn useful properties of the structure of this dataset. In the context of deep learning, we usually want to learn the entire probability distribution that generated a dataset, whether explicitly as in density estimation or implicitly for tasks like synthesis or denoising. Some other unsupervised learning algorithms perform other roles, like clustering, which consists of dividing the dataset into clusters of similar examples.

Supervised learning algorithms experience a dataset containing features, but each example is also associated with a **label** or **target**. For example, the Iris dataset is annotated with the species of each iris plant. A supervised learning algorithm can study the Iris dataset and learn to classify iris plants into three different species based on their measurements.

Roughly speaking, unsupervised learning involves observing several examples of a random vector $\mathbf{x}$, and attempting to implicitly or explicitly learn the probability distribution $p(\mathbf{x})$, or some interesting properties of that distribution, while supervised learning involves observing several examples of a random vector $\mathbf{x}$ and an associated value or vector $\mathbf{y}$, and learning to predict $\mathbf{y}$ from $\mathbf{x}$, usually by estimating $p(\mathbf{y} \mid \mathbf{x})$. The term **supervised learning** originates from the view of the target $\mathbf{y}$ being provided by an instructor or teacher who shows the machine learning system what to do. In unsupervised learning, there is no instructor or teacher, and the algorithm must learn to make sense of the data without this guide.

Unsupervised learning and supervised learning are not formally defined terms. The lines between them are often blurred. Many machine learning technologies can be used to perform both tasks. For example, the chain rule of probability states that for a vector $\mathbf{x} \in \mathbb{R}^n$, the joint distribution can be decomposed as

$$p(\mathbf{x}) = \prod_{i=1}^{n} p(\mathrm{x}_i \mid \mathrm{x}_1, \ldots, \mathrm{x}_{i-1}). \tag{5.1}$$

This decomposition means that we can solve the ostensibly unsupervised problem of modeling $p(\mathbf{x})$ by splitting it into n supervised learning problems. Alternatively, we

can solve the supervised learning problem of learning $p(y \mid \mathbf{x})$ by using traditional unsupervised learning technologies to learn the joint distribution $p(\mathbf{x}, y)$ and inferring

$$p(y \mid \mathbf{x}) = \frac{p(\mathbf{x}, y)}{\sum_{y'} p(\mathbf{x}, y')}. \tag{5.2}$$

Though unsupervised learning and supervised learning are not completely formal or distinct concepts, they do help to roughly categorize some of the things we do with machine learning algorithms. Traditionally, people refer to regression, classification and structured output problems as supervised learning. Density estimation in support of other tasks is usually considered unsupervised learning.

Other variants of the learning paradigm are possible. For example, in semi-supervised learning, some examples include a supervision target but others do not. In multi-instance learning, an entire collection of examples is labeled as containing or not containing an example of a class, but the individual members of the collection are not labeled. For a recent example of multi-instance learning with deep models, see Kotzias *et al.* (2015).

Some machine learning algorithms do not just experience a fixed dataset. For example, **reinforcement learning** algorithms interact with an environment, so there is a feedback loop between the learning system and its experiences. Such algorithms are beyond the scope of this book. Please see Sutton and Barto (1998) or Bertsekas and Tsitsiklis (1996) for information about reinforcement learning, and Mnih *et al.* (2013) for the deep learning approach to reinforcement learning.

Most machine learning algorithms simply experience a dataset. A dataset can be described in many ways. In all cases, a dataset is a collection of examples, which are in turn collections of features.

One common way of describing a dataset is with a **design matrix**. A design matrix is a matrix containing a different example in each row. Each column of the matrix corresponds to a different feature. For instance, the Iris dataset contains 150 examples with four features for each example. This means we can represent the dataset with a design matrix $\boldsymbol{X} \in \mathbb{R}^{150 \times 4}$, where $X_{i,1}$ is the sepal length of plant i, $X_{i,2}$ is the sepal width of plant i, etc. We will describe most of the learning algorithms in this book in terms of how they operate on design matrix datasets.

Of course, to describe a dataset as a design matrix, it must be possible to describe each example as a vector, and each of these vectors must be the same size. This is not always possible. For example, if you have a collection of photographs with different widths and heights, then different photographs will contain different numbers of pixels, so not all of the photographs may be described with the same length of vector. Section 9.7 and chapter 10 describe how to handle different

types of such heterogeneous data. In cases like these, rather than describing the dataset as a matrix with m rows, we will describe it as a set containing m elements: $\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(m)}\}$. This notation does not imply that any two example vectors $\boldsymbol{x}^{(i)}$ and $\boldsymbol{x}^{(j)}$ have the same size.

In the case of supervised learning, the example contains a label or target as well as a collection of features. For example, if we want to use a learning algorithm to perform object recognition from photographs, we need to specify which object appears in each of the photos. We might do this with a numeric code, with 0 signifying a person, 1 signifying a car, 2 signifying a cat, etc. Often when working with a dataset containing a design matrix of feature observations $\boldsymbol{X}$, we also provide a vector of labels $\boldsymbol{y}$, with y_i providing the label for example i.

Of course, sometimes the label may be more than just a single number. For example, if we want to train a speech recognition system to transcribe entire sentences, then the label for each example sentence is a sequence of words.

Just as there is no formal definition of supervised and unsupervised learning, there is no rigid taxonomy of datasets or experiences. The structures described here cover most cases, but it is always possible to design new ones for new applications.

5.1.4 Example: Linear Regression

Our definition of a machine learning algorithm as an algorithm that is capable of improving a computer program's performance at some task via experience is somewhat abstract. To make this more concrete, we present an example of a simple machine learning algorithm: **linear regression**. We will return to this example repeatedly as we introduce more machine learning concepts that help to understand its behavior.

As the name implies, linear regression solves a regression problem. In other words, the goal is to build a system that can take a vector $\boldsymbol{x} \in \mathbb{R}^n$ as input and predict the value of a scalar $y \in \mathbb{R}$ as its output. In the case of linear regression, the output is a linear function of the input. Let $\hat{y}$ be the value that our model predicts y should take on. We define the output to be

$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x} \tag{5.3}$$

where $\boldsymbol{w} \in \mathbb{R}^n$ is a vector of **parameters**.

Parameters are values that control the behavior of the system. In this case, w_i is the coefficient that we multiply by feature x_i before summing up the contributions from all the features. We can think of $\boldsymbol{w}$ as a set of **weights** that determine how each feature affects the prediction. If a feature x_i receives a positive weight w_i,

then increasing the value of that feature increases the value of our prediction $\hat{y}$. If a feature receives a negative weight, then increasing the value of that feature decreases the value of our prediction. If a feature's weight is large in magnitude, then it has a large effect on the prediction. If a feature's weight is zero, it has no effect on the prediction.

We thus have a definition of our task T: to predict y from $\boldsymbol{x}$ by outputting $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$. Next we need a definition of our performance measure, P.

Suppose that we have a design matrix of m example inputs that we will not use for training, only for evaluating how well the model performs. We also have a vector of regression targets providing the correct value of y for each of these examples. Because this dataset will only be used for evaluation, we call it the **test set**. We refer to the design matrix of inputs as $\boldsymbol{X}^{(\text{test})}$ and the vector of regression targets as $\boldsymbol{y}^{(\text{test})}$.

One way of measuring the performance of the model is to compute the **mean squared error** of the model on the test set. If $\hat{\boldsymbol{y}}^{(\text{test})}$ gives the predictions of the model on the test set, then the mean squared error is given by

$$\text{MSE}_{\text{test}} = \frac{1}{m} \sum_i (\hat{\boldsymbol{y}}^{(\text{test})} - \boldsymbol{y}^{(\text{test})})_i^2. \tag{5.4}$$

Intuitively, one can see that this error measure decreases to 0 when $\hat{\boldsymbol{y}}^{(\text{test})} = \boldsymbol{y}^{(\text{test})}$. We can also see that

$$\text{MSE}_{\text{test}} = \frac{1}{m} ||\hat{\boldsymbol{y}}^{(\text{test})} - \boldsymbol{y}^{(\text{test})}||_2^2, \tag{5.5}$$

so the error increases whenever the Euclidean distance between the predictions and the targets increases.

To make a machine learning algorithm, we need to design an algorithm that will improve the weights $\boldsymbol{w}$ in a way that reduces MSE_{test} when the algorithm is allowed to gain experience by observing a training set $(\boldsymbol{X}^{(\text{train})}, \boldsymbol{y}^{(\text{train})})$. One intuitive way of doing this (which we will justify later, in section 5.5.1) is just to minimize the mean squared error on the training set, $\text{MSE}_{\text{train}}$.

To minimize $\text{MSE}_{\text{train}}$, we can simply solve for where its gradient is $\boldsymbol{0}$:

$$\nabla_{\boldsymbol{w}} \text{MSE}_{\text{train}} = 0 \tag{5.6}$$

$$\Rightarrow \nabla_{\boldsymbol{w}} \frac{1}{m} ||\hat{\boldsymbol{y}}^{(\text{train})} - \boldsymbol{y}^{(\text{train})}||_2^2 = 0 \tag{5.7}$$

$$\Rightarrow \frac{1}{m} \nabla_{\boldsymbol{w}} ||\boldsymbol{X}^{(\text{train})} \boldsymbol{w} - \boldsymbol{y}^{(\text{train})}||_2^2 = 0 \tag{5.8}$$

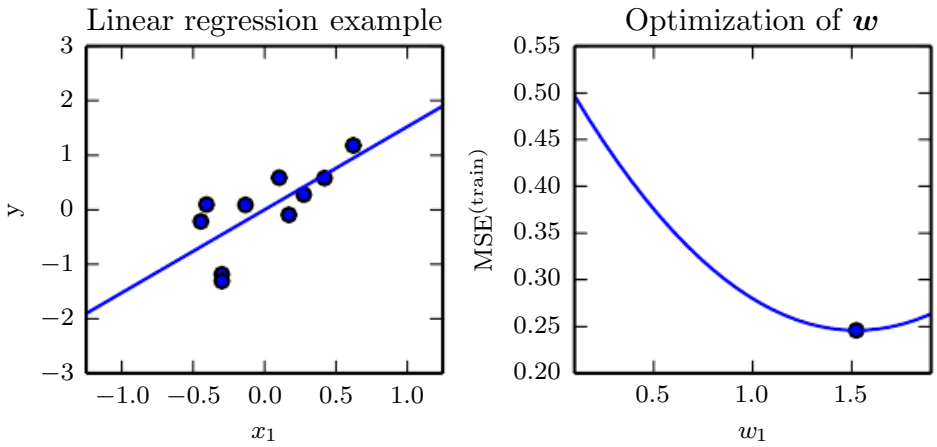

Figure 5.1: A linear regression problem, with a training set consisting of ten data points, each containing one feature. Because there is only one feature, the weight vector $\boldsymbol{w}$ contains only a single parameter to learn, w_1. *(Left)*Observe that linear regression learns to set w_1 such that the line $y = w_1 x$ comes as close as possible to passing through all the training points. *(Right)*The plotted point indicates the value of w_1 found by the normal equations, which we can see minimizes the mean squared error on the training set.

$$\Rightarrow \nabla_{\boldsymbol{w}} \left(\boldsymbol{X}^{(\text{train})} \boldsymbol{w} - \boldsymbol{y}^{(\text{train})} \right)^\top \left(\boldsymbol{X}^{(\text{train})} \boldsymbol{w} - \boldsymbol{y}^{(\text{train})} \right) = 0 \tag{5.9}$$

$$\Rightarrow \nabla_{\boldsymbol{w}} \left(\boldsymbol{w}^\top \boldsymbol{X}^{(\text{train})\top} \boldsymbol{X}^{(\text{train})} \boldsymbol{w} - 2\boldsymbol{w}^\top \boldsymbol{X}^{(\text{train})\top} \boldsymbol{y}^{(\text{train})} + \boldsymbol{y}^{(\text{train})\top} \boldsymbol{y}^{(\text{train})} \right) = 0 \tag{5.10}$$

$$\Rightarrow 2\boldsymbol{X}^{(\text{train})\top} \boldsymbol{X}^{(\text{train})} \boldsymbol{w} - 2\boldsymbol{X}^{(\text{train})\top} \boldsymbol{y}^{(\text{train})} = 0 \tag{5.11}$$

$$\Rightarrow \boldsymbol{w} = \left(\boldsymbol{X}^{(\text{train})\top} \boldsymbol{X}^{(\text{train})} \right)^{-1} \boldsymbol{X}^{(\text{train})\top} \boldsymbol{y}^{(\text{train})} \tag{5.12}$$

The system of equations whose solution is given by equation 5.12 is known as the **normal equations**. Evaluating equation 5.12 constitutes a simple learning algorithm. For an example of the linear regression learning algorithm in action, see figure 5.1.

It is worth noting that the term **linear regression** is often used to refer to a slightly more sophisticated model with one additional parameter—an intercept term b. In this model

$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x} + b \tag{5.13}$$

so the mapping from parameters to predictions is still a linear function but the mapping from features to predictions is now an affine function. This extension to affine functions means that the plot of the model's predictions still looks like a line, but it need not pass through the origin. Instead of adding the bias parameter

b, one can continue to use the model with only weights but augment $\boldsymbol{x}$ with an extra entry that is always set to 1. The weight corresponding to the extra 1 entry plays the role of the bias parameter. We will frequently use the term "linear" when referring to affine functions throughout this book.

The intercept term b is often called the **bias** parameter of the affine transformation. This terminology derives from the point of view that the output of the transformation is biased toward being b in the absence of any input. This term is different from the idea of a statistical bias, in which a statistical estimation algorithm's expected estimate of a quantity is not equal to the true quantity.

Linear regression is of course an extremely simple and limited learning algorithm, but it provides an example of how a learning algorithm can work. In the subsequent sections we will describe some of the basic principles underlying learning algorithm design and demonstrate how these principles can be used to build more complicated learning algorithms.

5.2 Capacity, Overfitting and Underfitting

The central challenge in machine learning is that we must perform well on *new, previously unseen* inputs—not just those on which our model was trained. The ability to perform well on previously unobserved inputs is called **generalization**.

Typically, when training a machine learning model, we have access to a training set, we can compute some error measure on the training set called the **training error**, and we reduce this training error. So far, what we have described is simply an optimization problem. What separates machine learning from optimization is that we want the **generalization error**, also called the **test error**, to be low as well. The generalization error is defined as the expected value of the error on a new input. Here the expectation is taken across different possible inputs, drawn from the distribution of inputs we expect the system to encounter in practice.

We typically estimate the generalization error of a machine learning model by measuring its performance on a **test set** of examples that were collected separately from the training set.

In our linear regression example, we trained the model by minimizing the training error,

$$\frac{1}{m^{(\text{train})}}||\boldsymbol{X}^{(\text{train})}\boldsymbol{w} - \boldsymbol{y}^{(\text{train})}||_2^2, \tag{5.14}$$

but we actually care about the test error, $\frac{1}{m^{(\text{test})}}||\boldsymbol{X}^{(\text{test})}\boldsymbol{w} - \boldsymbol{y}^{(\text{test})}||_2^2$.

How can we affect performance on the test set when we get to observe only the

training set? The field of **statistical learning theory** provides some answers. If the training and the test set are collected arbitrarily, there is indeed little we can do. If we are allowed to make some assumptions about how the training and test set are collected, then we can make some progress.

The train and test data are generated by a probability distribution over datasets called the **data generating process**. We typically make a set of assumptions known collectively as the **i.i.d. assumptions**. These assumptions are that the examples in each dataset are **independent** from each other, and that the train set and test set are **identically distributed**, drawn from the same probability distribution as each other. This assumption allows us to describe the data generating process with a probability distribution over a single example. The same distribution is then used to generate every train example and every test example. We call that shared underlying distribution the **data generating distribution**, denoted p_{data}. This probabilistic framework and the i.i.d. assumptions allow us to mathematically study the relationship between training error and test error.

One immediate connection we can observe between the training and test error is that the expected training error of a randomly selected model is equal to the expected test error of that model. Suppose we have a probability distribution $p(\boldsymbol{x}, y)$ and we sample from it repeatedly to generate the train set and the test set. For some fixed value $\boldsymbol{w}$, the expected training set error is exactly the same as the expected test set error, because both expectations are formed using the same dataset sampling process. The only difference between the two conditions is the name we assign to the dataset we sample.

Of course, when we use a machine learning algorithm, we do not fix the parameters ahead of time, then sample both datasets. We sample the training set, then use it to choose the parameters to reduce training set error, then sample the test set. Under this process, the expected test error is greater than or equal to the expected value of training error. The factors determining how well a machine learning algorithm will perform are its ability to:

1. Make the training error small.

2. Make the gap between training and test error small.

These two factors correspond to the two central challenges in machine learning: **underfitting** and **overfitting**. Underfitting occurs when the model is not able to obtain a sufficiently low error value on the training set. Overfitting occurs when the gap between the training error and test error is too large.

We can control whether a model is more likely to overfit or underfit by altering its **capacity**. Informally, a model's capacity is its ability to fit a wide variety of

functions. Models with low capacity may struggle to fit the training set. Models with high capacity can overfit by memorizing properties of the training set that do not serve them well on the test set.

One way to control the capacity of a learning algorithm is by choosing its **hypothesis space**, the set of functions that the learning algorithm is allowed to select as being the solution. For example, the linear regression algorithm has the set of all linear functions of its input as its hypothesis space. We can generalize linear regression to include polynomials, rather than just linear functions, in its hypothesis space. Doing so increases the model's capacity.

A polynomial of degree one gives us the linear regression model with which we are already familiar, with prediction

$$\hat{y} = b + wx. \tag{5.15}$$

By introducing x^2 as another feature provided to the linear regression model, we can learn a model that is quadratic as a function of x:

$$\hat{y} = b + w_1 x + w_2 x^2 . \tag{5.16}$$

Though this model implements a quadratic function of its *input*, the output is still a linear function of the *parameters*, so we can still use the normal equations to train the model in closed form. We can continue to add more powers of x as additional features, for example to obtain a polynomial of degree 9:

$$\hat{y} = b + \sum_{i=1}^{9} w_i x^i . \tag{5.17}$$

Machine learning algorithms will generally perform best when their capacity is appropriate for the true complexity of the task they need to perform and the amount of training data they are provided with. Models with insufficient capacity are unable to solve complex tasks. Models with high capacity can solve complex tasks, but when their capacity is higher than needed to solve the present task they may overfit.

Figure 5.2 shows this principle in action. We compare a linear, quadratic and degree-9 predictor attempting to fit a problem where the true underlying function is quadratic. The linear function is unable to capture the curvature in the true underlying problem, so it underfits. The degree-9 predictor is capable of representing the correct function, but it is also capable of representing infinitely many other functions that pass exactly through the training points, because we

have more parameters than training examples. We have little chance of choosing a solution that generalizes well when so many wildly different solutions exist. In this example, the quadratic model is perfectly matched to the true structure of the task so it generalizes well to new data.

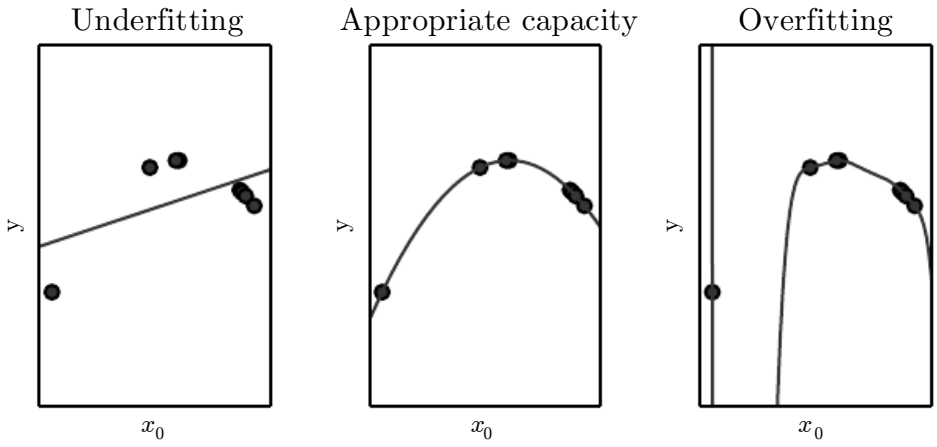

Figure 5.2: We fit three models to this example training set. The training data was generated synthetically, by randomly sampling x values and choosing y deterministically by evaluating a quadratic function. *(Left)*A linear function fit to the data suffers from underfitting—it cannot capture the curvature that is present in the data. *(Center)*A quadratic function fit to the data generalizes well to unseen points. It does not suffer from a significant amount of overfitting or underfitting. *(Right)*A polynomial of degree 9 fit to the data suffers from overfitting. Here we used the Moore-Penrose pseudoinverse to solve the underdetermined normal equations. The solution passes through all of the training points exactly, but we have not been lucky enough for it to extract the correct structure. It now has a deep valley in between two training points that does not appear in the true underlying function. It also increases sharply on the left side of the data, while the true function decreases in this area.

So far we have described only one way of changing a model's capacity: by changing the number of input features it has, and simultaneously adding new parameters associated with those features. There are in fact many ways of changing a model's capacity. Capacity is not determined only by the choice of model. The model specifies which family of functions the learning algorithm can choose from when varying the parameters in order to reduce a training objective. This is called the **representational capacity** of the model. In many cases, finding the best function within this family is a very difficult optimization problem. In practice, the learning algorithm does not actually find the best function, but merely one that significantly reduces the training error. These additional limitations, such as

the imperfection of the optimization algorithm, mean that the learning algorithm's **effective capacity** may be less than the representational capacity of the model family.

Our modern ideas about improving the generalization of machine learning models are refinements of thought dating back to philosophers at least as early as Ptolemy. Many early scholars invoke a principle of parsimony that is now most widely known as **Occam's razor** (c. 1287-1347). This principle states that among competing hypotheses that explain known observations equally well, one should choose the "simplest" one. This idea was formalized and made more precise in the 20th century by the founders of statistical learning theory (Vapnik and Chervonenkis, 1971; Vapnik, 1982; Blumer *et al.*, 1989; Vapnik, 1995).

Statistical learning theory provides various means of quantifying model capacity. Among these, the most well-known is the **Vapnik-Chervonenkis dimension**, or VC dimension. The VC dimension measures the capacity of a binary classifier. The VC dimension is defined as being the largest possible value of m for which there exists a training set of m different $\boldsymbol{x}$ points that the classifier can label arbitrarily.

Quantifying the capacity of the model allows statistical learning theory to make quantitative predictions. The most important results in statistical learning theory show that the discrepancy between training error and generalization error is bounded from above by a quantity that grows as the model capacity grows but shrinks as the number of training examples increases (Vapnik and Chervonenkis, 1971; Vapnik, 1982; Blumer *et al.*, 1989; Vapnik, 1995). These bounds provide intellectual justification that machine learning algorithms can work, but they are rarely used in practice when working with deep learning algorithms. This is in part because the bounds are often quite loose and in part because it can be quite difficult to determine the capacity of deep learning algorithms. The problem of determining the capacity of a deep learning model is especially difficult because the effective capacity is limited by the capabilities of the optimization algorithm, and we have little theoretical understanding of the very general non-convex optimization problems involved in deep learning.

We must remember that while simpler functions are more likely to generalize (to have a small gap between training and test error) we must still choose a sufficiently complex hypothesis to achieve low training error. Typically, training error decreases until it asymptotes to the minimum possible error value as model capacity increases (assuming the error measure has a minimum value). Typically, generalization error has a U-shaped curve as a function of model capacity. This is illustrated in figure 5.3.

To reach the most extreme case of arbitrarily high capacity, we introduce

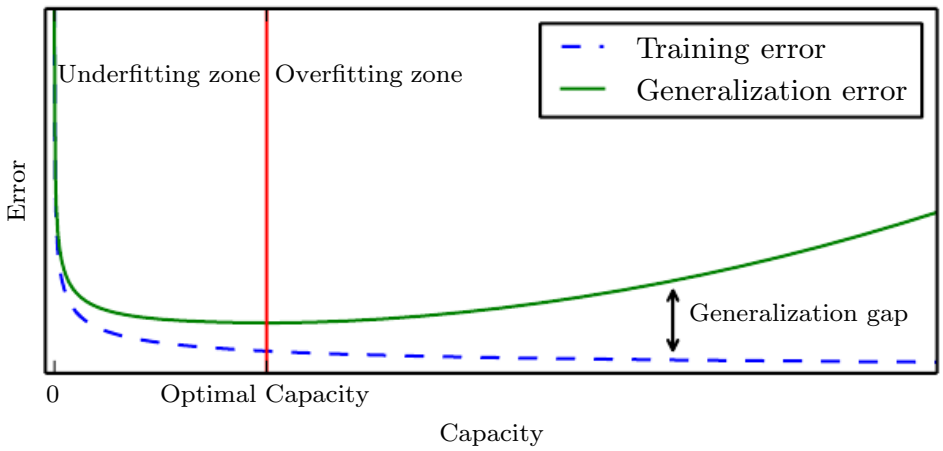

Figure 5.3: Typical relationship between capacity and error. Training and test error behave differently. At the left end of the graph, training error and generalization error are both high. This is the **underfitting regime**. As we increase capacity, training error decreases, but the gap between training and generalization error increases. Eventually, the size of this gap outweighs the decrease in training error, and we enter the **overfitting regime**, where capacity is too large, above the **optimal capacity**.

the concept of **non-parametric** models. So far, we have seen only parametric models, such as linear regression. Parametric models learn a function described by a parameter vector whose size is finite and fixed before any data is observed. Non-parametric models have no such limitation.

Sometimes, non-parametric models are just theoretical abstractions (such as an algorithm that searches over all possible probability distributions) that cannot be implemented in practice. However, we can also design practical non-parametric models by making their complexity a function of the training set size. One example of such an algorithm is **nearest neighbor regression**. Unlike linear regression, which has a fixed-length vector of weights, the nearest neighbor regression model simply stores the $\boldsymbol{X}$ and $\boldsymbol{y}$ from the training set. When asked to classify a test point $\boldsymbol{x}$, the model looks up the nearest entry in the training set and returns the associated regression target. In other words, $\hat{y} = y_i$ where $i = \arg\min ||\boldsymbol{X}_{i,:} - \boldsymbol{x}||_2^2$. The algorithm can also be generalized to distance metrics other than the L^2 norm, such as learned distance metrics (Goldberger *et al.*, 2005). If the algorithm is allowed to break ties by averaging the y_i values for all $\boldsymbol{X}_{i,:}$ that are tied for nearest, then this algorithm is able to achieve the minimum possible training error (which might be greater than zero, if two identical inputs are associated with different outputs) on any regression dataset.

Finally, we can also create a non-parametric learning algorithm by wrapping a

parametric learning algorithm inside another algorithm that increases the number of parameters as needed. For example, we could imagine an outer loop of learning that changes the degree of the polynomial learned by linear regression on top of a polynomial expansion of the input.

The ideal model is an oracle that simply knows the true probability distribution that generates the data. Even such a model will still incur some error on many problems, because there may still be some noise in the distribution. In the case of supervised learning, the mapping from $\boldsymbol{x}$ to y may be inherently stochastic, or y may be a deterministic function that involves other variables besides those included in $\boldsymbol{x}$. The error incurred by an oracle making predictions from the true distribution $p(\boldsymbol{x}, y)$ is called the **Bayes error**.

Training and generalization error vary as the size of the training set varies. Expected generalization error can never increase as the number of training examples increases. For non-parametric models, more data yields better generalization until the best possible error is achieved. Any fixed parametric model with less than optimal capacity will asymptote to an error value that exceeds the Bayes error. See figure 5.4 for an illustration. Note that it is possible for the model to have optimal capacity and yet still have a large gap between training and generalization error. In this situation, we may be able to reduce this gap by gathering more training examples.

5.2.1 The No Free Lunch Theorem

Learning theory claims that a machine learning algorithm can generalize well from a finite training set of examples. This seems to contradict some basic principles of logic. Inductive reasoning, or inferring general rules from a limited set of examples, is not logically valid. To logically infer a rule describing every member of a set, one must have information about every member of that set.

In part, machine learning avoids this problem by offering only probabilistic rules, rather than the entirely certain rules used in purely logical reasoning. Machine learning promises to find rules that are *probably* correct about *most* members of the set they concern.

Unfortunately, even this does not resolve the entire problem. The **no free lunch theorem** for machine learning (Wolpert, 1996) states that, averaged over all possible data generating distributions, every classification algorithm has the same error rate when classifying previously unobserved points. In other words, in some sense, no machine learning algorithm is universally any better than any other. The most sophisticated algorithm we can conceive of has the same average

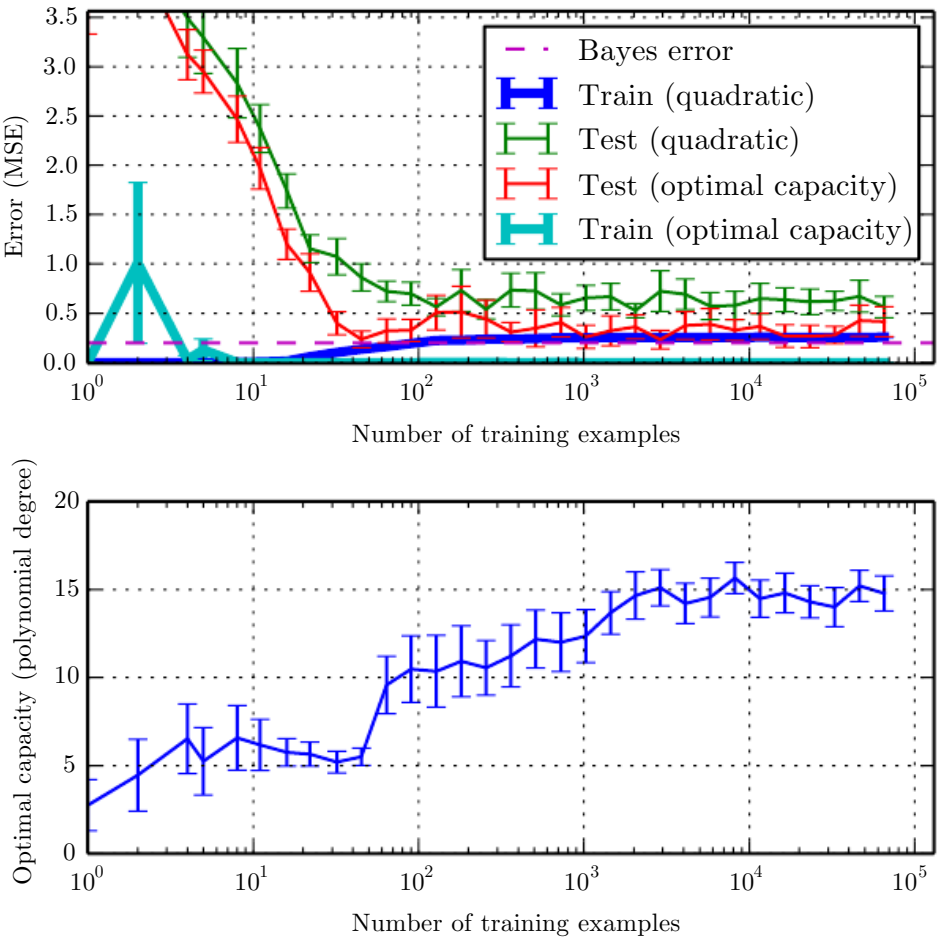

Figure 5.4: The effect of the training dataset size on the train and test error, as well as on the optimal model capacity. We constructed a synthetic regression problem based on adding a moderate amount of noise to a degree-5 polynomial, generated a single test set, and then generated several different sizes of training set. For each size, we generated 40 different training sets in order to plot error bars showing 95 percent confidence intervals. *(Top)*The MSE on the training and test set for two different models: a quadratic model, and a model with degree chosen to minimize the test error. Both are fit in closed form. For the quadratic model, the training error increases as the size of the training set increases. This is because larger datasets are harder to fit. Simultaneously, the test error decreases, because fewer incorrect hypotheses are consistent with the training data. The quadratic model does not have enough capacity to solve the task, so its test error asymptotes to a high value. The test error at optimal capacity asymptotes to the Bayes error. The training error can fall below the Bayes error, due to the ability of the training algorithm to memorize specific instances of the training set. As the training size increases to infinity, the training error of any fixed-capacity model (here, the quadratic model) must rise to at least the Bayes error. *(Bottom)*As the training set size increases, the optimal capacity (shown here as the degree of the optimal polynomial regressor) increases. The optimal capacity plateaus after reaching sufficient complexity to solve the task.

performance (over all possible tasks) as merely predicting that every point belongs to the same class.

Fortunately, these results hold only when we average over *all* possible data generating distributions. If we make assumptions about the kinds of probability distributions we encounter in real-world applications, then we can design learning algorithms that perform well on these distributions.

This means that the goal of machine learning research is not to seek a universal learning algorithm or the absolute best learning algorithm. Instead, our goal is to understand what kinds of distributions are relevant to the "real world" that an AI agent experiences, and what kinds of machine learning algorithms perform well on data drawn from the kinds of data generating distributions we care about.

5.2.2 Regularization

The no free lunch theorem implies that we must design our machine learning algorithms to perform well on a specific task. We do so by building a set of preferences into the learning algorithm. When these preferences are aligned with the learning problems we ask the algorithm to solve, it performs better.

So far, the only method of modifying a learning algorithm that we have discussed concretely is to increase or decrease the model's representational capacity by adding or removing functions from the hypothesis space of solutions the learning algorithm is able to choose. We gave the specific example of increasing or decreasing the degree of a polynomial for a regression problem. The view we have described so far is oversimplified.

The behavior of our algorithm is strongly affected not just by how large we make the set of functions allowed in its hypothesis space, but by the specific identity of those functions. The learning algorithm we have studied so far, linear regression, has a hypothesis space consisting of the set of linear functions of its input. These linear functions can be very useful for problems where the relationship between inputs and outputs truly is close to linear. They are less useful for problems that behave in a very nonlinear fashion. For example, linear regression would not perform very well if we tried to use it to predict $\sin(x)$ from x. We can thus control the performance of our algorithms by choosing what kind of functions we allow them to draw solutions from, as well as by controlling the amount of these functions.

We can also give a learning algorithm a preference for one solution in its hypothesis space to another. This means that both functions are eligible, but one is preferred. The unpreferred solution will be chosen only if it fits the training

data significantly better than the preferred solution.

For example, we can modify the training criterion for linear regression to include **weight decay**. To perform linear regression with weight decay, we minimize a sum comprising both the mean squared error on the training and a criterion $J(\boldsymbol{w})$ that expresses a preference for the weights to have smaller squared L^2 norm. Specifically,

$$J(\boldsymbol{w}) = \text{MSE}_{\text{train}} + \lambda \boldsymbol{w}^\top \boldsymbol{w}, \tag{5.18}$$

where λ is a value chosen ahead of time that controls the strength of our preference for smaller weights. When $\lambda = 0$, we impose no preference, and larger λ forces the weights to become smaller. Minimizing $J(\boldsymbol{w})$ results in a choice of weights that make a tradeoff between fitting the training data and being small. This gives us solutions that have a smaller slope, or put weight on fewer of the features. As an example of how we can control a model's tendency to overfit or underfit via weight decay, we can train a high-degree polynomial regression model with different values of λ. See figure 5.5 for the results.

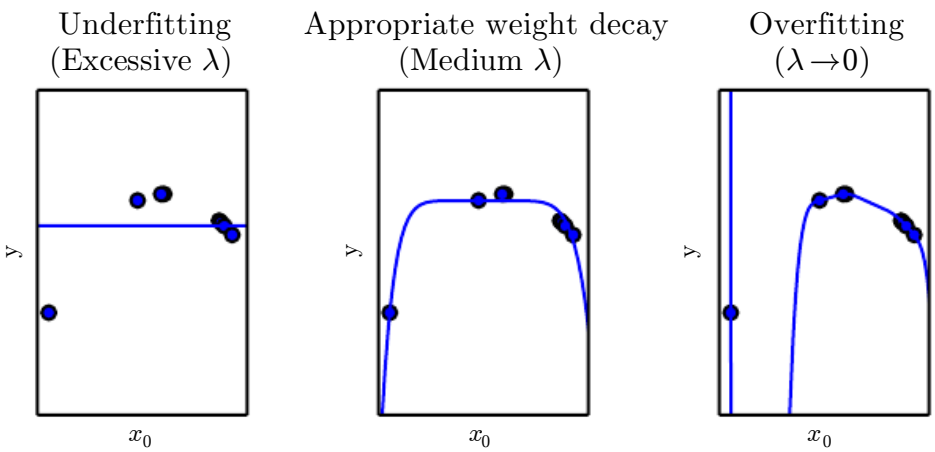

Figure 5.5: We fit a high-degree polynomial regression model to our example training set from figure 5.2. The true function is quadratic, but here we use only models with degree 9. We vary the amount of weight decay to prevent these high-degree models from overfitting. *(Left)*With very large λ, we can force the model to learn a function with no slope at all. This underfits because it can only represent a constant function. *(Center)*With a medium value of λ, the learning algorithm recovers a curve with the right general shape. Even though the model is capable of representing functions with much more complicated shape, weight decay has encouraged it to use a simpler function described by smaller coefficients. *(Right)*With weight decay approaching zero (i.e., using the Moore-Penrose pseudoinverse to solve the underdetermined problem with minimal regularization), the degree-9 polynomial overfits significantly, as we saw in figure 5.2.

More generally, we can regularize a model that learns a function $f(\boldsymbol{x}; \boldsymbol{\theta})$ by adding a penalty called a **regularizer** to the cost function. In the case of weight decay, the regularizer is $\Omega(\boldsymbol{w}) = \boldsymbol{w}^\top \boldsymbol{w}$. In chapter 7, we will see that many other regularizers are possible.

Expressing preferences for one function over another is a more general way of controlling a model's capacity than including or excluding members from the hypothesis space. We can think of excluding a function from a hypothesis space as expressing an infinitely strong preference against that function.

In our weight decay example, we expressed our preference for linear functions defined with smaller weights explicitly, via an extra term in the criterion we minimize. There are many other ways of expressing preferences for different solutions, both implicitly and explicitly. Together, these different approaches are known as **regularization**. *Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.* Regularization is one of the central concerns of the field of machine learning, rivaled in its importance only by optimization.

The no free lunch theorem has made it clear that there is no best machine learning algorithm, and, in particular, no best form of regularization. Instead we must choose a form of regularization that is well-suited to the particular task we want to solve. The philosophy of deep learning in general and this book in particular is that a very wide range of tasks (such as all of the intellectual tasks that people can do) may all be solved effectively using very general-purpose forms of regularization.

5.3 Hyperparameters and Validation Sets

Most machine learning algorithms have several settings that we can use to control the behavior of the learning algorithm. These settings are called **hyperparameters**. The values of hyperparameters are not adapted by the learning algorithm itself (though we can design a nested learning procedure where one learning algorithm learns the best hyperparameters for another learning algorithm).

In the polynomial regression example we saw in figure 5.2, there is a single hyperparameter: the degree of the polynomial, which acts as a **capacity** hyperparameter. The λ value used to control the strength of weight decay is another example of a hyperparameter.

Sometimes a setting is chosen to be a hyperparameter that the learning algorithm does not learn because it is difficult to optimize. More frequently, the

setting must be a hyperparameter because it is not appropriate to learn that hyperparameter on the training set. This applies to all hyperparameters that control model capacity. If learned on the training set, such hyperparameters would always choose the maximum possible model capacity, resulting in overfitting (refer to figure 5.3). For example, we can always fit the training set better with a higher degree polynomial and a weight decay setting of $\lambda = 0$ than we could with a lower degree polynomial and a positive weight decay setting.

To solve this problem, we need a **validation set** of examples that the training algorithm does not observe.

Earlier we discussed how a held-out test set, composed of examples coming from the same distribution as the training set, can be used to estimate the generalization error of a learner, after the learning process has completed. It is important that the test examples are not used in any way to make choices about the model, including its hyperparameters. For this reason, no example from the test set can be used in the validation set. Therefore, we always construct the validation set from the *training* data. Specifically, we split the training data into two disjoint subsets. One of these subsets is used to learn the parameters. The other subset is our validation set, used to estimate the generalization error during or after training, allowing for the hyperparameters to be updated accordingly. The subset of data used to learn the parameters is still typically called the training set, even though this may be confused with the larger pool of data used for the entire training process. The subset of data used to guide the selection of hyperparameters is called the validation set. Typically, one uses about 80% of the training data for training and 20% for validation. Since the validation set is used to "train" the hyperparameters, the validation set error will underestimate the generalization error, though typically by a smaller amount than the training error. After all hyperparameter optimization is complete, the generalization error may be estimated using the test set.

In practice, when the same test set has been used repeatedly to evaluate performance of different algorithms over many years, and especially if we consider all the attempts from the scientific community at beating the reported state-of-the-art performance on that test set, we end up having optimistic evaluations with the test set as well. Benchmarks can thus become stale and then do not reflect the true field performance of a trained system. Thankfully, the community tends to move on to new (and usually more ambitious and larger) benchmark datasets.

5.3.1 Cross-Validation

Dividing the dataset into a fixed training set and a fixed test set can be problematic if it results in the test set being small. A small test set implies statistical uncertainty around the estimated average test error, making it difficult to claim that algorithm A works better than algorithm B on the given task.

When the dataset has hundreds of thousands of examples or more, this is not a serious issue. When the dataset is too small, are alternative procedures enable one to use all of the examples in the estimation of the mean test error, at the price of increased computational cost. These procedures are based on the idea of repeating the training and testing computation on different randomly chosen subsets or splits of the original dataset. The most common of these is the k-fold cross-validation procedure, shown in algorithm 5.1, in which a partition of the dataset is formed by splitting it into k non-overlapping subsets. The test error may then be estimated by taking the average test error across k trials. On trial i, the i-th subset of the data is used as the test set and the rest of the data is used as the training set. One problem is that there exist no unbiased estimators of the variance of such average error estimators (Bengio and Grandvalet, 2004), but approximations are typically used.

5.4 Estimators, Bias and Variance

The field of statistics gives us many tools that can be used to achieve the machine learning goal of solving a task not only on the training set but also to generalize. Foundational concepts such as parameter estimation, bias and variance are useful to formally characterize notions of generalization, underfitting and overfitting.

5.4.1 Point Estimation

Point estimation is the attempt to provide the single "best" prediction of some quantity of interest. In general the quantity of interest can be a single parameter or a vector of parameters in some parametric model, such as the weights in our linear regression example in section 5.1.4, but it can also be a whole function.

In order to distinguish estimates of parameters from their true value, our convention will be to denote a point estimate of a parameter $\boldsymbol{\theta}$ by $\hat{\boldsymbol{\theta}}$.

Let $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ be a set of m independent and identically distributed

Algorithm 5.1 The k-fold cross-validation algorithm. It can be used to estimate generalization error of a learning algorithm A when the given dataset $\mathbb{D}$ is too small for a simple train/test or train/valid split to yield accurate estimation of generalization error, because the mean of a loss L on a small test set may have too high variance. The dataset $\mathbb{D}$ contains as elements the abstract examples $\boldsymbol{z}^{(i)}$ (for the i-th example), which could stand for an (input,target) pair $\boldsymbol{z}^{(i)} = (\boldsymbol{x}^{(i)}, y^{(i)})$ in the case of supervised learning, or for just an input $\boldsymbol{z}^{(i)} = \boldsymbol{x}^{(i)}$ in the case of unsupervised learning. The algorithm returns the vector of errors $\boldsymbol{e}$ for each example in $\mathbb{D}$, whose mean is the estimated generalization error. The errors on individual examples can be used to compute a confidence interval around the mean (equation 5.47). While these confidence intervals are not well-justified after the use of cross-validation, it is still common practice to use them to declare that algorithm A is better than algorithm B only if the confidence interval of the error of algorithm A lies below and does not intersect the confidence interval of algorithm B.

Define `KFoldXV`$(\mathbb{D}, A, L, k)$:
Require: $\mathbb{D}$, the given dataset, with elements $\boldsymbol{z}^{(i)}$
Require: A, the learning algorithm, seen as a function that takes a dataset as input and outputs a learned function
Require: L, the loss function, seen as a function from a learned function f and an example $\boldsymbol{z}^{(i)} \in \mathbb{D}$ to a scalar $\in \mathbb{R}$
Require: k, the number of folds
  Split $\mathbb{D}$ into k mutually exclusive subsets $\mathbb{D}_i$, whose union is $\mathbb{D}$.
  for i from 1 to k **do**
    $f_i = A(\mathbb{D} \backslash \mathbb{D}_i)$
    for $\boldsymbol{z}^{(j)}$ in $\mathbb{D}_i$ **do**
      $e_j = L(f_i, \boldsymbol{z}^{(j)})$
    end for
  end for
  Return $\boldsymbol{e}$

(i.i.d.) data points. A **point estimator** or **statistic** is any function of the data:

$$\hat{\boldsymbol{\theta}}_m = g(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}). \tag{5.19}$$

The definition does not require that g return a value that is close to the true $\boldsymbol{\theta}$ or even that the range of g is the same as the set of allowable values of $\boldsymbol{\theta}$. This definition of a point estimator is very general and allows the designer of an estimator great flexibility. While almost any function thus qualifies as an estimator, a good estimator is a function whose output is close to the true underlying $\boldsymbol{\theta}$ that generated the training data.

For now, we take the frequentist perspective on statistics. That is, we assume that the true parameter value $\boldsymbol{\theta}$ is fixed but unknown, while the point estimate $\hat{\boldsymbol{\theta}}$ is a function of the data. Since the data is drawn from a random process, any function of the data is random. Therefore $\hat{\boldsymbol{\theta}}$ is a random variable.

Point estimation can also refer to the estimation of the relationship between input and target variables. We refer to these types of point estimates as function estimators.

Function Estimation As we mentioned above, sometimes we are interested in performing function estimation (or function approximation). Here we are trying to predict a variable $\boldsymbol{y}$ given an input vector $\boldsymbol{x}$. We assume that there is a function $f(\boldsymbol{x})$ that describes the approximate relationship between $\boldsymbol{y}$ and $\boldsymbol{x}$. For example, we may assume that $\boldsymbol{y} = f(\boldsymbol{x}) + \boldsymbol{\epsilon}$, where $\boldsymbol{\epsilon}$ stands for the part of $\boldsymbol{y}$ that is not predictable from $\boldsymbol{x}$. In function estimation, we are interested in approximating f with a model or estimate $\hat{f}$. Function estimation is really just the same as estimating a parameter $\boldsymbol{\theta}$; the function estimator $\hat{f}$ is simply a point estimator in function space. The linear regression example (discussed above in section 5.1.4) and the polynomial regression example (discussed in section 5.2) are both examples of scenarios that may be interpreted either as estimating a parameter $\boldsymbol{w}$ or estimating a function $\hat{f}$ mapping from $\boldsymbol{x}$ to y.

We now review the most commonly studied properties of point estimators and discuss what they tell us about these estimators.

5.4.2 Bias

The bias of an estimator is defined as:

$$\text{bias}(\hat{\boldsymbol{\theta}}_m) = \mathbb{E}(\hat{\boldsymbol{\theta}}_m) - \boldsymbol{\theta} \tag{5.20}$$

where the expectation is over the data (seen as samples from a random variable) and $\boldsymbol{\theta}$ is the true underlying value of $\boldsymbol{\theta}$ used to define the data generating distribution. An estimator $\hat{\boldsymbol{\theta}}_m$ is said to be **unbiased** if $\text{bias}(\hat{\boldsymbol{\theta}}_m) = \mathbf{0}$, which implies that $\mathbb{E}(\hat{\boldsymbol{\theta}}_m) = \boldsymbol{\theta}$. An estimator $\hat{\boldsymbol{\theta}}_m$ is said to be **asymptotically unbiased** if $\lim_{m\to\infty} \text{bias}(\hat{\boldsymbol{\theta}}_m) = \mathbf{0}$, which implies that $\lim_{m\to\infty} \mathbb{E}(\hat{\boldsymbol{\theta}}_m) = \boldsymbol{\theta}$.

Example: Bernoulli Distribution Consider a set of samples $\{x^{(1)}, \ldots, x^{(m)}\}$ that are independently and identically distributed according to a Bernoulli distribution with mean θ:

$$P(x^{(i)}; \theta) = \theta^{x^{(i)}}(1-\theta)^{(1-x^{(i)})}. \tag{5.21}$$

A common estimator for the θ parameter of this distribution is the mean of the training samples:

$$\hat{\theta}_m = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}. \tag{5.22}$$

To determine whether this estimator is biased, we can substitute equation 5.22 into equation 5.20:

$$\begin{aligned}
\text{bias}(\hat{\theta}_m) &= \mathbb{E}[\hat{\theta}_m] - \theta && (5.23)\\
&= \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right] - \theta && (5.24)\\
&= \frac{1}{m}\sum_{i=1}^{m} \mathbb{E}\left[x^{(i)}\right] - \theta && (5.25)\\
&= \frac{1}{m}\sum_{i=1}^{m} \sum_{x^{(i)}=0}^{1} \left(x^{(i)}\theta^{x^{(i)}}(1-\theta)^{(1-x^{(i)})}\right) - \theta && (5.26)\\
&= \frac{1}{m}\sum_{i=1}^{m} (\theta) - \theta && (5.27)\\
&= \theta - \theta = 0 && (5.28)
\end{aligned}$$

Since $\text{bias}(\hat{\theta}) = 0$, we say that our estimator $\hat{\theta}$ is unbiased.

Example: Gaussian Distribution Estimator of the Mean Now, consider a set of samples $\{x^{(1)}, \ldots, x^{(m)}\}$ that are independently and identically distributed according to a Gaussian distribution $p(x^{(i)}) = \mathcal{N}(x^{(i)}; \mu, \sigma^2)$, where $i \in \{1, \ldots, m\}$.

Recall that the Gaussian probability density function is given by

$$p(x^{(i)}; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\frac{(x^{(i)} - \mu)^2}{\sigma^2}\right). \tag{5.29}$$

A common estimator of the Gaussian mean parameter is known as the **sample mean**:

$$\hat{\mu}_m = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \tag{5.30}$$

To determine the bias of the sample mean, we are again interested in calculating its expectation:

$$\begin{aligned}
\text{bias}(\hat{\mu}_m) &= \mathbb{E}[\hat{\mu}_m] - \mu && (5.31)\\
&= \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right] - \mu && (5.32)\\
&= \left(\frac{1}{m}\sum_{i=1}^{m} \mathbb{E}\left[x^{(i)}\right]\right) - \mu && (5.33)\\
&= \left(\frac{1}{m}\sum_{i=1}^{m} \mu\right) - \mu && (5.34)\\
&= \mu - \mu = 0 && (5.35)
\end{aligned}$$

Thus we find that the sample mean is an unbiased estimator of Gaussian mean parameter.

Example: Estimators of the Variance of a Gaussian Distribution As an example, we compare two different estimators of the variance parameter σ^2 of a Gaussian distribution. We are interested in knowing if either estimator is biased.

The first estimator of σ^2 we consider is known as the **sample variance**:

$$\hat{\sigma}_m^2 = \frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)} - \hat{\mu}_m\right)^2, \tag{5.36}$$

where $\hat{\mu}_m$ is the sample mean, defined above. More formally, we are interested in computing

$$\text{bias}(\hat{\sigma}_m^2) = \mathbb{E}[\hat{\sigma}_m^2] - \sigma^2 \tag{5.37}$$

We begin by evaluating the term $\mathbb{E}[\hat{\sigma}_m^2]$:

$$\mathbb{E}[\hat{\sigma}_m^2] = \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)} - \hat{\mu}_m\right)^2\right] \tag{5.38}$$

$$= \frac{m-1}{m}\sigma^2 \tag{5.39}$$

Returning to equation 5.37, we conclude that the bias of $\hat{\sigma}_m^2$ is $-\sigma^2/m$. Therefore, the sample variance is a biased estimator.

The **unbiased sample variance** estimator

$$\tilde{\sigma}_m^2 = \frac{1}{m-1}\sum_{i=1}^{m}\left(x^{(i)} - \hat{\mu}_m\right)^2 \tag{5.40}$$

provides an alternative approach. As the name suggests this estimator is unbiased. That is, we find that $\mathbb{E}[\tilde{\sigma}_m^2] = \sigma^2$:

$$\mathbb{E}[\tilde{\sigma}_m^2] = \mathbb{E}\left[\frac{1}{m-1}\sum_{i=1}^{m}\left(x^{(i)} - \hat{\mu}_m\right)^2\right] \tag{5.41}$$

$$= \frac{m}{m-1}\mathbb{E}[\hat{\sigma}_m^2] \tag{5.42}$$

$$= \frac{m}{m-1}\left(\frac{m-1}{m}\sigma^2\right) \tag{5.43}$$

$$= \sigma^2. \tag{5.44}$$

We have two estimators: one is biased and the other is not. While unbiased estimators are clearly desirable, they are not always the "best" estimators. As we will see we often use biased estimators that possess other important properties.

5.4.3 Variance and Standard Error

Another property of the estimator that we might want to consider is how much we expect it to vary as a function of the data sample. Just as we computed the expectation of the estimator to determine its bias, we can compute its variance. The **variance** of an estimator is simply the variance

$$\mathrm{Var}(\hat{\theta}) \tag{5.45}$$

where the random variable is the training set. Alternately, the square root of the variance is called the **standard error**, denoted $\mathrm{SE}(\hat{\theta})$.

The variance or the standard error of an estimator provides a measure of how we would expect the estimate we compute from data to vary as we independently resample the dataset from the underlying data generating process. Just as we might like an estimator to exhibit low bias we would also like it to have relatively low variance.

When we compute any statistic using a finite number of samples, our estimate of the true underlying parameter is uncertain, in the sense that we could have obtained other samples from the same distribution and their statistics would have been different. The expected degree of variation in any estimator is a source of error that we want to quantify.

The standard error of the mean is given by

$$\mathrm{SE}(\hat{\mu}_m) = \sqrt{\mathrm{Var}\left[\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right]} = \frac{\sigma}{\sqrt{m}}, \tag{5.46}$$

where σ^2 is the true variance of the samples x^i. The standard error is often estimated by using an estimate of σ. Unfortunately, neither the square root of the sample variance nor the square root of the unbiased estimator of the variance provide an unbiased estimate of the standard deviation. Both approaches tend to underestimate the true standard deviation, but are still used in practice. The square root of the unbiased estimator of the variance is less of an underestimate. For large m, the approximation is quite reasonable.

The standard error of the mean is very useful in machine learning experiments. We often estimate the generalization error by computing the sample mean of the error on the test set. The number of examples in the test set determines the accuracy of this estimate. Taking advantage of the central limit theorem, which tells us that the mean will be approximately distributed with a normal distribution, we can use the standard error to compute the probability that the true expectation falls in any chosen interval. For example, the 95% confidence interval centered on the mean $\hat{\mu}_m$ is

$$(\hat{\mu}_m - 1.96\mathrm{SE}(\hat{\mu}_m), \hat{\mu}_m + 1.96\mathrm{SE}(\hat{\mu}_m)), \tag{5.47}$$

under the normal distribution with mean $\hat{\mu}_m$ and variance $\mathrm{SE}(\hat{\mu}_m)^2$. In machine learning experiments, it is common to say that algorithm A is better than algorithm B if the upper bound of the 95% confidence interval for the error of algorithm A is less than the lower bound of the 95% confidence interval for the error of algorithm B.

Example: Bernoulli Distribution We once again consider a set of samples $\{x^{(1)}, \ldots, x^{(m)}\}$ drawn independently and identically from a Bernoulli distribution (recall $P(x^{(i)}; \theta) = \theta^{x^{(i)}}(1-\theta)^{(1-x^{(i)})}$). This time we are interested in computing the variance of the estimator $\hat{\theta}_m = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$.

$$\text{Var}\left(\hat{\theta}_m\right) = \text{Var}\left(\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right) \tag{5.48}$$

$$= \frac{1}{m^2}\sum_{i=1}^{m} \text{Var}\left(x^{(i)}\right) \tag{5.49}$$

$$= \frac{1}{m^2}\sum_{i=1}^{m} \theta(1-\theta) \tag{5.50}$$

$$= \frac{1}{m^2} m\theta(1-\theta) \tag{5.51}$$

$$= \frac{1}{m}\theta(1-\theta) \tag{5.52}$$

The variance of the estimator decreases as a function of m, the number of examples in the dataset. This is a common property of popular estimators that we will return to when we discuss consistency (see section 5.4.5).

5.4.4 Trading off Bias and Variance to Minimize Mean Squared Error

Bias and variance measure two different sources of error in an estimator. Bias measures the expected deviation from the true value of the function or parameter. Variance on the other hand, provides a measure of the deviation from the expected estimator value that any particular sampling of the data is likely to cause.

What happens when we are given a choice between two estimators, one with more bias and one with more variance? How do we choose between them? For example, imagine that we are interested in approximating the function shown in figure 5.2 and we are only offered the choice between a model with large bias and one that suffers from large variance. How do we choose between them?

The most common way to negotiate this trade-off is to use cross-validation. Empirically, cross-validation is highly successful on many real-world tasks. Alternatively, we can also compare the **mean squared error** (MSE) of the estimates:

$$\text{MSE} = \mathbb{E}[(\hat{\theta}_m - \theta)^2] \tag{5.53}$$

$$= \text{Bias}(\hat{\theta}_m)^2 + \text{Var}(\hat{\theta}_m) \tag{5.54}$$

The MSE measures the overall expected deviation—in a squared error sense—between the estimator and the true value of the parameter θ. As is clear from equation 5.54, evaluating the MSE incorporates both the bias and the variance. Desirable estimators are those with small MSE and these are estimators that manage to keep both their bias and variance somewhat in check.

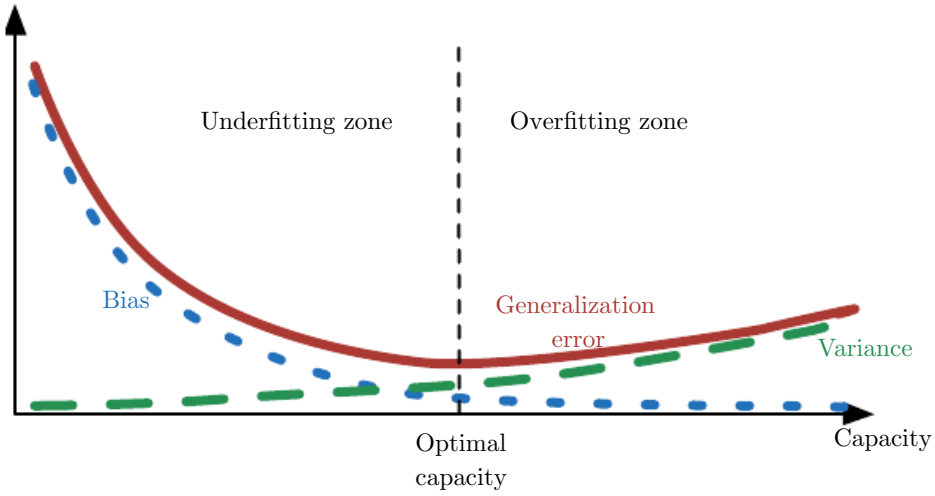

Figure 5.6: As capacity increases (x-axis), bias (dotted) tends to decrease and variance (dashed) tends to increase, yielding another U-shaped curve for generalization error (bold curve). If we vary capacity along one axis, there is an optimal capacity, with underfitting when the capacity is below this optimum and overfitting when it is above. This relationship is similar to the relationship between capacity, underfitting, and overfitting, discussed in section 5.2 and figure 5.3.

The relationship between bias and variance is tightly linked to the machine learning concepts of capacity, underfitting and overfitting. In the case where generalization error is measured by the MSE (where bias and variance are meaningful components of generalization error), increasing capacity tends to increase variance and decrease bias. This is illustrated in figure 5.6, where we see again the U-shaped curve of generalization error as a function of capacity.

5.4.5 Consistency

So far we have discussed the properties of various estimators for a training set of fixed size. Usually, we are also concerned with the behavior of an estimator as the amount of training data grows. In particular, we usually wish that, as the number of data points m in our dataset increases, our point estimates converge to the true

value of the corresponding parameters. More formally, we would like that

$$\text{plim}_{m\to\infty}\hat{\theta}_m = \theta. \tag{5.55}$$

The symbol plim indicates convergence in probability, meaning that for any $\epsilon > 0$, $P(|\hat{\theta}_m - \theta| > \epsilon) \to 0$ as $m \to \infty$. The condition described by equation 5.55 is known as **consistency**. It is sometimes referred to as weak consistency, with strong consistency referring to the **almost sure** convergence of $\hat{\theta}$ to θ. **Almost sure convergence** of a sequence of random variables $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots$ to a value $\boldsymbol{x}$ occurs when $p(\lim_{m\to\infty}\mathbf{x}^{(m)} = \boldsymbol{x}) = 1$.

Consistency ensures that the bias induced by the estimator diminishes as the number of data examples grows. However, the reverse is not true—asymptotic unbiasedness does not imply consistency. For example, consider estimating the mean parameter μ of a normal distribution $\mathcal{N}(x; \mu, \sigma^2)$, with a dataset consisting of m samples: $\{x^{(1)}, \ldots, x^{(m)}\}$. We could use the first sample $x^{(1)}$ of the dataset as an unbiased estimator: $\hat{\theta} = x^{(1)}$. In that case, $\mathbb{E}(\hat{\theta}_m) = \theta$ so the estimator is unbiased no matter how many data points are seen. This, of course, implies that the estimate is asymptotically unbiased. However, this is not a consistent estimator as it is *not* the case that $\hat{\theta}_m \to \theta$ as $m \to \infty$.

5.5 Maximum Likelihood Estimation

Previously, we have seen some definitions of common estimators and analyzed their properties. But where did these estimators come from? Rather than guessing that some function might make a good estimator and then analyzing its bias and variance, we would like to have some principle from which we can derive specific functions that are good estimators for different models.

The most common such principle is the maximum likelihood principle.

Consider a set of m examples $\mathbb{X} = \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ drawn independently from the true but unknown data generating distribution $p_{\text{data}}(\mathbf{x})$.

Let $p_{\text{model}}(\mathbf{x}; \boldsymbol{\theta})$ be a parametric family of probability distributions over the same space indexed by $\boldsymbol{\theta}$. In other words, $p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta})$ maps any configuration $\boldsymbol{x}$ to a real number estimating the true probability $p_{\text{data}}(\boldsymbol{x})$.

The maximum likelihood estimator for $\boldsymbol{\theta}$ is then defined as

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} p_{\text{model}}(\mathbb{X}; \boldsymbol{\theta}) \tag{5.56}$$

$$= \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{m} p_{\text{model}}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \tag{5.57}$$

This product over many probabilities can be inconvenient for a variety of reasons. For example, it is prone to numerical underflow. To obtain a more convenient but equivalent optimization problem, we observe that taking the logarithm of the likelihood does not change its arg max but does conveniently transform a product into a sum:

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\text{model}}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}). \tag{5.58}$$

Because the arg max does not change when we rescale the cost function, we can divide by m to obtain a version of the criterion that is expressed as an expectation with respect to the empirical distribution $\hat{p}_{\text{data}}$ defined by the training data:

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta}). \tag{5.59}$$

One way to interpret maximum likelihood estimation is to view it as minimizing the dissimilarity between the empirical distribution $\hat{p}_{\text{data}}$ defined by the training set and the model distribution, with the degree of dissimilarity between the two measured by the KL divergence. The KL divergence is given by

$$D_{\text{KL}}\left(\hat{p}_{\text{data}} \| p_{\text{model}}\right) = \mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}} \left[\log \hat{p}_{\text{data}}(\boldsymbol{x}) - \log p_{\text{model}}(\boldsymbol{x})\right]. \tag{5.60}$$

The term on the left is a function only of the data generating process, not the model. This means when we train the model to minimize the KL divergence, we need only minimize

$$- \mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}} \left[\log p_{\text{model}}(\boldsymbol{x})\right] \tag{5.61}$$

which is of course the same as the maximization in equation 5.59.

Minimizing this KL divergence corresponds exactly to minimizing the cross-entropy between the distributions. Many authors use the term "cross-entropy" to identify specifically the negative log-likelihood of a Bernoulli or softmax distribution, but that is a misnomer. Any loss consisting of a negative log-likelihood is a cross-entropy between the empirical distribution defined by the training set and the probability distribution defined by model. For example, mean squared error is the cross-entropy between the empirical distribution and a Gaussian model.

We can thus see maximum likelihood as an attempt to make the model distribution match the empirical distribution $\hat{p}_{\text{data}}$. Ideally, we would like to match the true data generating distribution p_{data}, but we have no direct access to this distribution.

While the optimal $\boldsymbol{\theta}$ is the same regardless of whether we are maximizing the likelihood or minimizing the KL divergence, the values of the objective functions

are different. In software, we often phrase both as minimizing a cost function. Maximum likelihood thus becomes minimization of the negative log-likelihood (NLL), or equivalently, minimization of the cross entropy. The perspective of maximum likelihood as minimum KL divergence becomes helpful in this case because the KL divergence has a known minimum value of zero. The negative log-likelihood can actually become negative when $\boldsymbol{x}$ is real-valued.

5.5.1 Conditional Log-Likelihood and Mean Squared Error

The maximum likelihood estimator can readily be generalized to the case where our goal is to estimate a conditional probability $P(\mathbf{y} \mid \mathbf{x}; \boldsymbol{\theta})$ in order to predict $\mathbf{y}$ given $\mathbf{x}$. This is actually the most common situation because it forms the basis for most supervised learning. If $\boldsymbol{X}$ represents all our inputs and $\boldsymbol{Y}$ all our observed targets, then the conditional maximum likelihood estimator is

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} P(\boldsymbol{Y} \mid \boldsymbol{X}; \boldsymbol{\theta}). \tag{5.62}$$

If the examples are assumed to be i.i.d., then this can be decomposed into

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log P(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}). \tag{5.63}$$

Example: Linear Regression as Maximum Likelihood Linear regression, introduced earlier in section 5.1.4, may be justified as a maximum likelihood procedure. Previously, we motivated linear regression as an algorithm that learns to take an input $\boldsymbol{x}$ and produce an output value $\hat{y}$. The mapping from $\boldsymbol{x}$ to $\hat{y}$ is chosen to minimize mean squared error, a criterion that we introduced more or less arbitrarily. We now revisit linear regression from the point of view of maximum likelihood estimation. Instead of producing a single prediction $\hat{y}$, we now think of the model as producing a conditional distribution $p(y \mid \boldsymbol{x})$. We can imagine that with an infinitely large training set, we might see several training examples with the same input value $\boldsymbol{x}$ but different values of y. The goal of the learning algorithm is now to fit the distribution $p(y \mid \boldsymbol{x})$ to all of those different y values that are all compatible with $\boldsymbol{x}$. To derive the same linear regression algorithm we obtained before, we define $p(y \mid \boldsymbol{x}) = \mathcal{N}(y; \hat{y}(\boldsymbol{x}; \boldsymbol{w}), \sigma^2)$. The function $\hat{y}(\boldsymbol{x}; \boldsymbol{w})$ gives the prediction of the mean of the Gaussian. In this example, we assume that the variance is fixed to some constant σ^2 chosen by the user. We will see that this choice of the functional form of $p(y \mid \boldsymbol{x})$ causes the maximum likelihood estimation procedure to yield the same learning algorithm as we developed before. Since the

examples are assumed to be i.i.d., the conditional log-likelihood (equation 5.63) is given by

$$\sum_{i=1}^{m} \log p(y^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \tag{5.64}$$

$$= -m \log \sigma - \frac{m}{2} \log(2\pi) - \sum_{i=1}^{m} \frac{\left\| \hat{y}^{(i)} - y^{(i)} \right\|^2}{2\sigma^2}, \tag{5.65}$$

where $\hat{y}^{(i)}$ is the output of the linear regression on the i-th input $\boldsymbol{x}^{(i)}$ and m is the number of the training examples. Comparing the log-likelihood with the mean squared error,

$$\text{MSE}_{\text{train}} = \frac{1}{m} \sum_{i=1}^{m} ||\hat{y}^{(i)} - y^{(i)}||^2, \tag{5.66}$$

we immediately see that maximizing the log-likelihood with respect to $\boldsymbol{w}$ yields the same estimate of the parameters $\boldsymbol{w}$ as does minimizing the mean squared error. The two criteria have different values but the same location of the optimum. This justifies the use of the MSE as a maximum likelihood estimation procedure. As we will see, the maximum likelihood estimator has several desirable properties.

5.5.2 Properties of Maximum Likelihood

The main appeal of the maximum likelihood estimator is that it can be shown to be the best estimator asymptotically, as the number of examples $m \to \infty$, in terms of its rate of convergence as m increases.

Under appropriate conditions, the maximum likelihood estimator has the property of consistency (see section 5.4.5 above), meaning that as the number of training examples approaches infinity, the maximum likelihood estimate of a parameter converges to the true value of the parameter. These conditions are:

- The true distribution p_{data} must lie within the model family $p_{\text{model}}(\cdot; \boldsymbol{\theta})$. Otherwise, no estimator can recover p_{data}.
- The true distribution p_{data} must correspond to exactly one value of $\boldsymbol{\theta}$. Otherwise, maximum likelihood can recover the correct p_{data}, but will not be able to determine which value of $\boldsymbol{\theta}$ was used by the data generating processing.

There are other inductive principles besides the maximum likelihood estimator, many of which share the property of being consistent estimators. However,

consistent estimators can differ in their **statistic efficiency**, meaning that one consistent estimator may obtain lower generalization error for a fixed number of samples m, or equivalently, may require fewer examples to obtain a fixed level of generalization error.

Statistical efficiency is typically studied in the **parametric case** (like in linear regression) where our goal is to estimate the value of a parameter (and assuming it is possible to identify the true parameter), not the value of a function. A way to measure how close we are to the true parameter is by the expected mean squared error, computing the squared difference between the estimated and true parameter values, where the expectation is over m training samples from the data generating distribution. That parametric mean squared error decreases as m increases, and for m large, the Cramér-Rao lower bound (Rao, 1945; Cramér, 1946) shows that no consistent estimator has a lower mean squared error than the maximum likelihood estimator.

For these reasons (consistency and efficiency), maximum likelihood is often considered the preferred estimator to use for machine learning. When the number of examples is small enough to yield overfitting behavior, regularization strategies such as weight decay may be used to obtain a biased version of maximum likelihood that has less variance when training data is limited.

5.6 Bayesian Statistics

So far we have discussed **frequentist statistics** and approaches based on estimating a single value of $\boldsymbol{\theta}$, then making all predictions thereafter based on that one estimate. Another approach is to consider all possible values of $\boldsymbol{\theta}$ when making a prediction. The latter is the domain of **Bayesian statistics**.

As discussed in section 5.4.1, the frequentist perspective is that the true parameter value $\boldsymbol{\theta}$ is fixed but unknown, while the point estimate $\hat{\boldsymbol{\theta}}$ is a random variable on account of it being a function of the dataset (which is seen as random).

The Bayesian perspective on statistics is quite different. The Bayesian uses probability to reflect degrees of certainty of states of knowledge. The dataset is directly observed and so is not random. On the other hand, the true parameter $\boldsymbol{\theta}$ is unknown or uncertain and thus is represented as a random variable.

Before observing the data, we represent our knowledge of $\boldsymbol{\theta}$ using the **prior probability distribution**, $p(\boldsymbol{\theta})$ (sometimes referred to as simply "the prior"). Generally, the machine learning practitioner selects a prior distribution that is quite broad (i.e. with high entropy) to reflect a high degree of uncertainty in the

value of $\boldsymbol{\theta}$ before observing any data. For example, one might assume *a priori* that $\boldsymbol{\theta}$ lies in some finite range or volume, with a uniform distribution. Many priors instead reflect a preference for "simpler" solutions (such as smaller magnitude coefficients, or a function that is closer to being constant).

Now consider that we have a set of data samples $\{x^{(1)}, \ldots, x^{(m)}\}$. We can recover the effect of data on our belief about $\boldsymbol{\theta}$ by combining the data likelihood $p(x^{(1)}, \ldots, x^{(m)} \mid \boldsymbol{\theta})$ with the prior via Bayes' rule:

$$p(\boldsymbol{\theta} \mid x^{(1)}, \ldots, x^{(m)}) = \frac{p(x^{(1)}, \ldots, x^{(m)} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta})}{p(x^{(1)}, \ldots, x^{(m)})} \tag{5.67}$$

In the scenarios where Bayesian estimation is typically used, the prior begins as a relatively uniform or Gaussian distribution with high entropy, and the observation of the data usually causes the posterior to lose entropy and concentrate around a few highly likely values of the parameters.

Relative to maximum likelihood estimation, Bayesian estimation offers two important differences. First, unlike the maximum likelihood approach that makes predictions using a point estimate of $\boldsymbol{\theta}$, the Bayesian approach is to make predictions using a full distribution over $\boldsymbol{\theta}$. For example, after observing m examples, the predicted distribution over the next data sample, $x^{(m+1)}$, is given by

$$p(x^{(m+1)} \mid x^{(1)}, \ldots, x^{(m)}) = \int p(x^{(m+1)} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta} \mid x^{(1)}, \ldots, x^{(m)}) \, d\boldsymbol{\theta}. \tag{5.68}$$

Here each value of $\boldsymbol{\theta}$ with positive probability density contributes to the prediction of the next example, with the contribution weighted by the posterior density itself. After having observed $\{x^{(1)}, \ldots, x^{(m)}\}$, if we are still quite uncertain about the value of $\boldsymbol{\theta}$, then this uncertainty is incorporated directly into any predictions we might make.

In section 5.4, we discussed how the frequentist approach addresses the uncertainty in a given point estimate of $\boldsymbol{\theta}$ by evaluating its variance. The variance of the estimator is an assessment of how the estimate might change with alternative samplings of the observed data. The Bayesian answer to the question of how to deal with the uncertainty in the estimator is to simply integrate over it, which tends to protect well against overfitting. This integral is of course just an application of the laws of probability, making the Bayesian approach simple to justify, while the frequentist machinery for constructing an estimator is based on the rather ad hoc decision to summarize all knowledge contained in the dataset with a single point estimate.

The second important difference between the Bayesian approach to estimation and the maximum likelihood approach is due to the contribution of the Bayesian

prior distribution. The prior has an influence by shifting probability mass density towards regions of the parameter space that are preferred *a priori*. In practice, the prior often expresses a preference for models that are simpler or more smooth. Critics of the Bayesian approach identify the prior as a source of subjective human judgment impacting the predictions.

Bayesian methods typically generalize much better when limited training data is available, but typically suffer from high computational cost when the number of training examples is large.

Example: Bayesian Linear Regression Here we consider the Bayesian estimation approach to learning the linear regression parameters. In linear regression, we learn a linear mapping from an input vector $\boldsymbol{x} \in \mathbb{R}^n$ to predict the value of a scalar $y \in \mathbb{R}$. The prediction is parametrized by the vector $\boldsymbol{w} \in \mathbb{R}^n$:

$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}. \tag{5.69}$$

Given a set of m training samples $(\boldsymbol{X}^{(\text{train})}, \boldsymbol{y}^{(\text{train})})$, we can express the prediction of y over the entire training set as:

$$\hat{\boldsymbol{y}}^{(\text{train})} = \boldsymbol{X}^{(\text{train})} \boldsymbol{w}. \tag{5.70}$$

Expressed as a Gaussian conditional distribution on $\boldsymbol{y}^{(\text{train})}$, we have

$$p(\boldsymbol{y}^{(\text{train})} \mid \boldsymbol{X}^{(\text{train})}, \boldsymbol{w}) = \mathcal{N}(\boldsymbol{y}^{(\text{train})}; \boldsymbol{X}^{(\text{train})} \boldsymbol{w}, \boldsymbol{I}) \tag{5.71}$$

$$\propto \exp\left(-\frac{1}{2} (\boldsymbol{y}^{(\text{train})} - \boldsymbol{X}^{(\text{train})} \boldsymbol{w})^\top (\boldsymbol{y}^{(\text{train})} - \boldsymbol{X}^{(\text{train})} \boldsymbol{w}) \right), \tag{5.72}$$

where we follow the standard MSE formulation in assuming that the Gaussian variance on y is one. In what follows, to reduce the notational burden, we refer to $(\boldsymbol{X}^{(\text{train})}, \boldsymbol{y}^{(\text{train})})$ as simply $(\boldsymbol{X}, \boldsymbol{y})$.

To determine the posterior distribution over the model parameter vector $\boldsymbol{w}$, we first need to specify a prior distribution. The prior should reflect our naive belief about the value of these parameters. While it is sometimes difficult or unnatural to express our prior beliefs in terms of the parameters of the model, in practice we typically assume a fairly broad distribution expressing a high degree of uncertainty about $\boldsymbol{\theta}$. For real-valued parameters it is common to use a Gaussian as a prior distribution:

$$p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \boldsymbol{\mu}_0, \boldsymbol{\Lambda}_0) \propto \exp\left(-\frac{1}{2} (\boldsymbol{w} - \boldsymbol{\mu}_0)^\top \boldsymbol{\Lambda}_0^{-1} (\boldsymbol{w} - \boldsymbol{\mu}_0) \right), \tag{5.73}$$

where $\boldsymbol{\mu}_0$ and $\boldsymbol{\Lambda}_0$ are the prior distribution mean vector and covariance matrix respectively.[1]

With the prior thus specified, we can now proceed in determining the **posterior** distribution over the model parameters.

$$p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}) \propto p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w}) p(\boldsymbol{w}) \tag{5.74}$$

$$\propto \exp\left(-\frac{1}{2}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})^\top (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})\right) \exp\left(-\frac{1}{2}(\boldsymbol{w} - \boldsymbol{\mu}_0)^\top \boldsymbol{\Lambda}_0^{-1}(\boldsymbol{w} - \boldsymbol{\mu}_0)\right) \tag{5.75}$$

$$\propto \exp\left(-\frac{1}{2}\left(-2\boldsymbol{y}^\top \boldsymbol{X}\boldsymbol{w} + \boldsymbol{w}^\top \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{w} + \boldsymbol{w}^\top \boldsymbol{\Lambda}_0^{-1}\boldsymbol{w} - 2\boldsymbol{\mu}_0^\top \boldsymbol{\Lambda}_0^{-1}\boldsymbol{w}\right)\right). \tag{5.76}$$

We now define $\boldsymbol{\Lambda}_m = \left(\boldsymbol{X}^\top \boldsymbol{X} + \boldsymbol{\Lambda}_0^{-1}\right)^{-1}$ and $\boldsymbol{\mu}_m = \boldsymbol{\Lambda}_m \left(\boldsymbol{X}^\top \boldsymbol{y} + \boldsymbol{\Lambda}_0^{-1}\boldsymbol{\mu}_0\right)$. Using these new variables, we find that the posterior may be rewritten as a Gaussian distribution:

$$p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}) \propto \exp\left(-\frac{1}{2}(\boldsymbol{w} - \boldsymbol{\mu}_m)^\top \boldsymbol{\Lambda}_m^{-1}(\boldsymbol{w} - \boldsymbol{\mu}_m) + \frac{1}{2}\boldsymbol{\mu}_m^\top \boldsymbol{\Lambda}_m^{-1}\boldsymbol{\mu}_m\right) \tag{5.77}$$

$$\propto \exp\left(-\frac{1}{2}(\boldsymbol{w} - \boldsymbol{\mu}_m)^\top \boldsymbol{\Lambda}_m^{-1}(\boldsymbol{w} - \boldsymbol{\mu}_m)\right). \tag{5.78}$$

All terms that do not include the parameter vector $\boldsymbol{w}$ have been omitted; they are implied by the fact that the distribution must be normalized to integrate to 1. Equation 3.23 shows how to normalize a multivariate Gaussian distribution.

Examining this posterior distribution allows us to gain some intuition for the effect of Bayesian inference. In most situations, we set $\boldsymbol{\mu}_0$ to $\mathbf{0}$. If we set $\boldsymbol{\Lambda}_0 = \frac{1}{\alpha}\boldsymbol{I}$, then μ_m gives the same estimate of $\boldsymbol{w}$ as does frequentist linear regression with a weight decay penalty of $\alpha \boldsymbol{w}^\top \boldsymbol{w}$. One difference is that the Bayesian estimate is undefined if α is set to zero—-we are not allowed to begin the Bayesian learning process with an infinitely wide prior on $\boldsymbol{w}$. The more important difference is that the Bayesian estimate provides a covariance matrix, showing how likely all the different values of $\boldsymbol{w}$ are, rather than providing only the estimate μ_m.

5.6.1 Maximum *A Posteriori* (MAP) Estimation

While the most principled approach is to make predictions using the full Bayesian posterior distribution over the parameter $\boldsymbol{\theta}$, it is still often desirable to have a

[1] Unless there is a reason to assume a particular covariance structure, we typically assume a diagonal covariance matrix $\boldsymbol{\Lambda}_0 = \text{diag}(\boldsymbol{\lambda}_0)$.

single point estimate. One common reason for desiring a point estimate is that most operations involving the Bayesian posterior for most interesting models are intractable, and a point estimate offers a tractable approximation. Rather than simply returning to the maximum likelihood estimate, we can still gain some of the benefit of the Bayesian approach by allowing the prior to influence the choice of the point estimate. One rational way to do this is to choose the **maximum a posteriori** (MAP) point estimate. The MAP estimate chooses the point of maximal posterior probability (or maximal probability density in the more common case of continuous $\boldsymbol{\theta}$):

$$\boldsymbol{\theta}_{\text{MAP}} = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta} \mid \boldsymbol{x}) = \arg\max_{\boldsymbol{\theta}} \log p(\boldsymbol{x} \mid \boldsymbol{\theta}) + \log p(\boldsymbol{\theta}). \tag{5.79}$$

We recognize, above on the right hand side, $\log p(\boldsymbol{x} \mid \boldsymbol{\theta})$, i.e. the standard log-likelihood term, and $\log p(\boldsymbol{\theta})$, corresponding to the prior distribution.

As an example, consider a linear regression model with a Gaussian prior on the weights $\boldsymbol{w}$. If this prior is given by $\mathcal{N}(\boldsymbol{w}; \mathbf{0}, \frac{1}{\lambda}\boldsymbol{I}^2)$, then the log-prior term in equation 5.79 is proportional to the familiar $\lambda \boldsymbol{w}^\top \boldsymbol{w}$ weight decay penalty, plus a term that does not depend on $\boldsymbol{w}$ and does not affect the learning process. MAP Bayesian inference with a Gaussian prior on the weights thus corresponds to weight decay.

As with full Bayesian inference, MAP Bayesian inference has the advantage of leveraging information that is brought by the prior and cannot be found in the training data. This additional information helps to reduce the variance in the MAP point estimate (in comparison to the ML estimate). However, it does so at the price of increased bias.

Many regularized estimation strategies, such as maximum likelihood learning regularized with weight decay, can be interpreted as making the MAP approximation to Bayesian inference. This view applies when the regularization consists of adding an extra term to the objective function that corresponds to $\log p(\boldsymbol{\theta})$. Not all regularization penalties correspond to MAP Bayesian inference. For example, some regularizer terms may not be the logarithm of a probability distribution. Other regularization terms depend on the data, which of course a prior probability distribution is not allowed to do.

MAP Bayesian inference provides a straightforward way to design complicated yet interpretable regularization terms. For example, a more complicated penalty term can be derived by using a mixture of Gaussians, rather than a single Gaussian distribution, as the prior (Nowlan and Hinton, 1992).

5.7 Supervised Learning Algorithms

Recall from section 5.1.3 that supervised learning algorithms are, roughly speaking, learning algorithms that learn to associate some input with some output, given a training set of examples of inputs $\boldsymbol{x}$ and outputs $\boldsymbol{y}$. In many cases the outputs $\boldsymbol{y}$ may be difficult to collect automatically and must be provided by a human "supervisor," but the term still applies even when the training set targets were collected automatically.

5.7.1 Probabilistic Supervised Learning

Most supervised learning algorithms in this book are based on estimating a probability distribution $p(y \mid \boldsymbol{x})$. We can do this simply by using maximum likelihood estimation to find the best parameter vector $\boldsymbol{\theta}$ for a parametric family of distributions $p(y \mid \boldsymbol{x}; \boldsymbol{\theta})$.

We have already seen that linear regression corresponds to the family

$$p(y \mid \boldsymbol{x}; \boldsymbol{\theta}) = \mathcal{N}(y; \boldsymbol{\theta}^\top \boldsymbol{x}, \boldsymbol{I}). \tag{5.80}$$

We can generalize linear regression to the classification scenario by defining a different family of probability distributions. If we have two classes, class 0 and class 1, then we need only specify the probability of one of these classes. The probability of class 1 determines the probability of class 0, because these two values must add up to 1.

The normal distribution over real-valued numbers that we used for linear regression is parametrized in terms of a mean. Any value we supply for this mean is valid. A distribution over a binary variable is slightly more complicated, because its mean must always be between 0 and 1. One way to solve this problem is to use the logistic sigmoid function to squash the output of the linear function into the interval (0, 1) and interpret that value as a probability:

$$p(y = 1 \mid \boldsymbol{x}; \boldsymbol{\theta}) = \sigma(\boldsymbol{\theta}^\top \boldsymbol{x}). \tag{5.81}$$

This approach is known as **logistic regression** (a somewhat strange name since we use the model for classification rather than regression).

In the case of linear regression, we were able to find the optimal weights by solving the normal equations. Logistic regression is somewhat more difficult. There is no closed-form solution for its optimal weights. Instead, we must search for them by maximizing the log-likelihood. We can do this by minimizing the negative log-likelihood (NLL) using gradient descent.

This same strategy can be applied to essentially any supervised learning problem, by writing down a parametric family of conditional probability distributions over the right kind of input and output variables.

5.7.2 Support Vector Machines

One of the most influential approaches to supervised learning is the support vector machine (Boser *et al.*, 1992; Cortes and Vapnik, 1995). This model is similar to logistic regression in that it is driven by a linear function $\boldsymbol{w}^\top \boldsymbol{x} + b$. Unlike logistic regression, the support vector machine does not provide probabilities, but only outputs a class identity. The SVM predicts that the positive class is present when $\boldsymbol{w}^\top \boldsymbol{x} + b$ is positive. Likewise, it predicts that the negative class is present when $\boldsymbol{w}^\top \boldsymbol{x} + b$ is negative.

One key innovation associated with support vector machines is the **kernel trick**. The kernel trick consists of observing that many machine learning algorithms can be written exclusively in terms of dot products between examples. For example, it can be shown that the linear function used by the support vector machine can be re-written as

$$\boldsymbol{w}^\top \boldsymbol{x} + b = b + \sum_{i=1}^{m} \alpha_i \boldsymbol{x}^\top \boldsymbol{x}^{(i)} \tag{5.82}$$

where $\boldsymbol{x}^{(i)}$ is a training example and $\boldsymbol{\alpha}$ is a vector of coefficients. Rewriting the learning algorithm this way allows us to replace $\boldsymbol{x}$ by the output of a given feature function $\phi(\boldsymbol{x})$ and the dot product with a function $k(\boldsymbol{x}, \boldsymbol{x}^{(i)}) = \phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{x}^{(i)})$ called a **kernel**. The $\cdot$ operator represents an inner product analogous to $\phi(\boldsymbol{x})^\top \phi(\boldsymbol{x}^{(i)})$. For some feature spaces, we may not use literally the vector inner product. In some infinite dimensional spaces, we need to use other kinds of inner products, for example, inner products based on integration rather than summation. A complete development of these kinds of inner products is beyond the scope of this book.

After replacing dot products with kernel evaluations, we can make predictions using the function

$$f(\boldsymbol{x}) = b + \sum_i \alpha_i k(\boldsymbol{x}, \boldsymbol{x}^{(i)}). \tag{5.83}$$

This function is nonlinear with respect to $\boldsymbol{x}$, but the relationship between $\phi(\boldsymbol{x})$ and $f(\boldsymbol{x})$ is linear. Also, the relationship between $\boldsymbol{\alpha}$ and $f(\boldsymbol{x})$ is linear. The kernel-based function is exactly equivalent to preprocessing the data by applying $\phi(\boldsymbol{x})$ to all inputs, then learning a linear model in the new transformed space.

The kernel trick is powerful for two reasons. First, it allows us to learn models that are nonlinear as a function of $\boldsymbol{x}$ using convex optimization techniques that are

guaranteed to converge efficiently. This is possible because we consider ϕ fixed and optimize only $\boldsymbol{\alpha}$, i.e., the optimization algorithm can view the decision function as being linear in a different space. Second, the kernel function k often admits an implementation that is significantly more computational efficient than naively constructing two $\phi(\boldsymbol{x})$ vectors and explicitly taking their dot product.

In some cases, $\phi(\boldsymbol{x})$ can even be infinite dimensional, which would result in an infinite computational cost for the naive, explicit approach. In many cases, $k(\boldsymbol{x}, \boldsymbol{x}')$ is a nonlinear, tractable function of $\boldsymbol{x}$ even when $\phi(\boldsymbol{x})$ is intractable. As an example of an infinite-dimensional feature space with a tractable kernel, we construct a feature mapping $\phi(x)$ over the non-negative integers x. Suppose that this mapping returns a vector containing x ones followed by infinitely many zeros. We can write a kernel function $k(x, x^{(i)}) = \min(x, x^{(i)})$ that is exactly equivalent to the corresponding infinite-dimensional dot product.

The most commonly used kernel is the **Gaussian kernel**

$$k(\boldsymbol{u}, \boldsymbol{v}) = \mathcal{N}(\boldsymbol{u} - \boldsymbol{v}; 0, \sigma^2 \boldsymbol{I}) \tag{5.84}$$

where $\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$ is the standard normal density. This kernel is also known as the **radial basis function** (RBF) kernel, because its value decreases along lines in $\boldsymbol{v}$ space radiating outward from $\boldsymbol{u}$. The Gaussian kernel corresponds to a dot product in an infinite-dimensional space, but the derivation of this space is less straightforward than in our example of the min kernel over the integers.

We can think of the Gaussian kernel as performing a kind of **template matching**. A training example $\boldsymbol{x}$ associated with training label y becomes a template for class y. When a test point $\boldsymbol{x}'$ is near $\boldsymbol{x}$ according to Euclidean distance, the Gaussian kernel has a large response, indicating that $\boldsymbol{x}'$ is very similar to the $\boldsymbol{x}$ template. The model then puts a large weight on the associated training label y. Overall, the prediction will combine many such training labels weighted by the similarity of the corresponding training examples.

Support vector machines are not the only algorithm that can be enhanced using the kernel trick. Many other linear models can be enhanced in this way. The category of algorithms that employ the kernel trick is known as **kernel machines** or **kernel methods** (Williams and Rasmussen, 1996; Schölkopf *et al.*, 1999).

A major drawback to kernel machines is that the cost of evaluating the decision function is linear in the number of training examples, because the i-th example contributes a term $\alpha_i k(\boldsymbol{x}, \boldsymbol{x}^{(i)})$ to the decision function. Support vector machines are able to mitigate this by learning an $\boldsymbol{\alpha}$ vector that contains mostly zeros. Classifying a new example then requires evaluating the kernel function only for the training examples that have non-zero α_i. These training examples are known

as **support vectors**.

Kernel machines also suffer from a high computational cost of training when the dataset is large. We will revisit this idea in section 5.9. Kernel machines with generic kernels struggle to generalize well. We will explain why in section 5.11. The modern incarnation of deep learning was designed to overcome these limitations of kernel machines. The current deep learning renaissance began when Hinton *et al.* (2006) demonstrated that a neural network could outperform the RBF kernel SVM on the MNIST benchmark.

5.7.3 Other Simple Supervised Learning Algorithms

We have already briefly encountered another non-probabilistic supervised learning algorithm, nearest neighbor regression. More generally, k-nearest neighbors is a family of techniques that can be used for classification or regression. As a non-parametric learning algorithm, k-nearest neighbors is not restricted to a fixed number of parameters. We usually think of the k-nearest neighbors algorithm as not having any parameters, but rather implementing a simple function of the training data. In fact, there is not even really a training stage or learning process. Instead, at test time, when we want to produce an output y for a new test input $\boldsymbol{x}$, we find the k-nearest neighbors to $\boldsymbol{x}$ in the training data $\boldsymbol{X}$. We then return the average of the corresponding y values in the training set. This works for essentially any kind of supervised learning where we can define an average over y values. In the case of classification, we can average over one-hot code vectors $\boldsymbol{c}$ with $c_y = 1$ and $c_i = 0$ for all other values of i. We can then interpret the average over these one-hot codes as giving a probability distribution over classes. As a non-parametric learning algorithm, k-nearest neighbor can achieve very high capacity. For example, suppose we have a multiclass classification task and measure performance with 0-1 loss. In this setting, 1-nearest neighbor converges to double the Bayes error as the number of training examples approaches infinity. The error in excess of the Bayes error results from choosing a single neighbor by breaking ties between equally distant neighbors randomly. When there is infinite training data, all test points $\boldsymbol{x}$ will have infinitely many training set neighbors at distance zero. If we allow the algorithm to use all of these neighbors to vote, rather than randomly choosing one of them, the procedure converges to the Bayes error rate. The high capacity of k-nearest neighbors allows it to obtain high accuracy given a large training set. However, it does so at high computational cost, and it may generalize very badly given a small, finite training set. One weakness of k-nearest neighbors is that it cannot learn that one feature is more discriminative than another. For example, imagine we have a regression task with $\boldsymbol{x} \in \mathbb{R}^{100}$ drawn from an isotropic Gaussian

distribution, but only a single variable x_1 is relevant to the output. Suppose further that this feature simply encodes the output directly, i.e. that $y = x_1$ in all cases. Nearest neighbor regression will not be able to detect this simple pattern. The nearest neighbor of most points $\boldsymbol{x}$ will be determined by the large number of features x_2 through x_{100}, not by the lone feature x_1. Thus the output on small training sets will essentially be random.

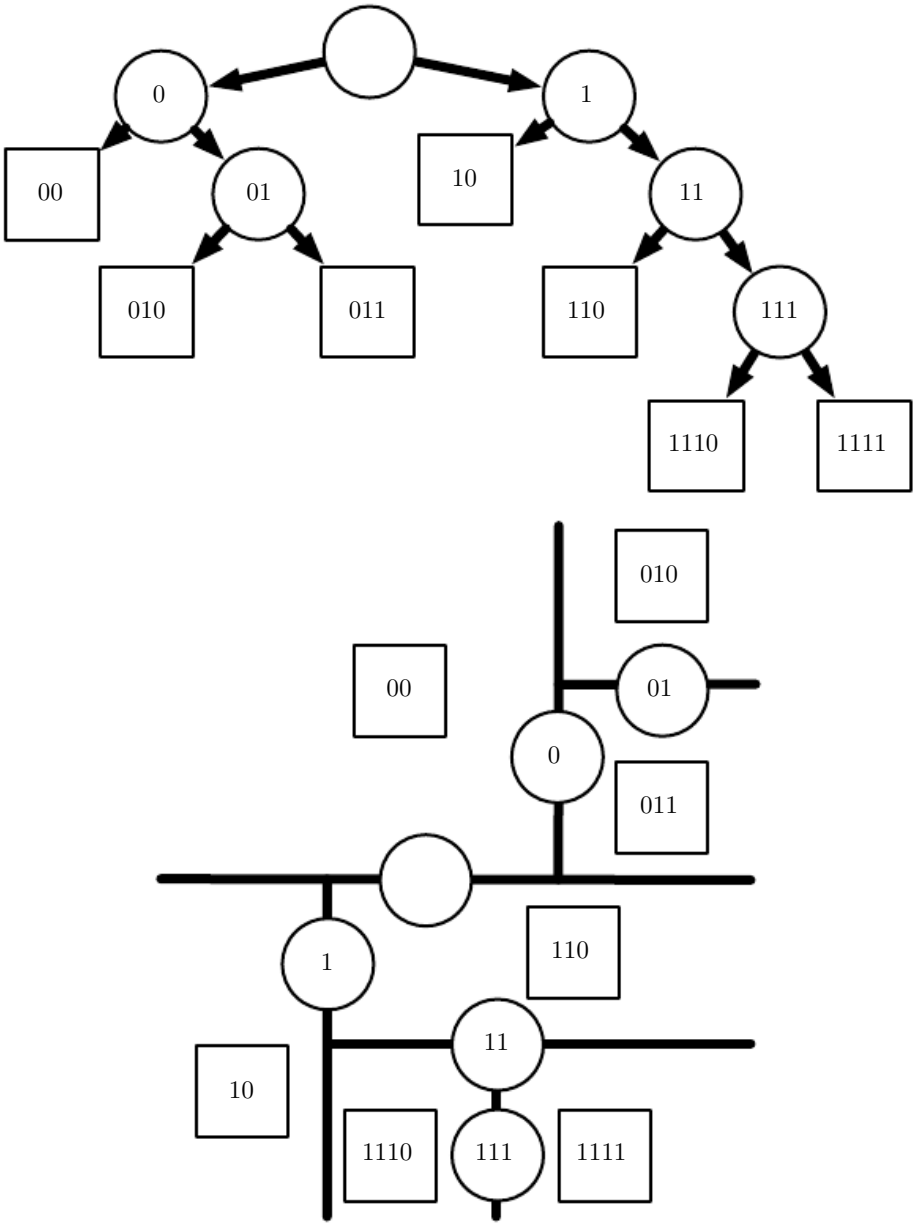

Figure 5.7: Diagrams describing how a decision tree works. *(Top)*Each node of the tree chooses to send the input example to the child node on the left (0) or or the child node on the right (1). Internal nodes are drawn as circles and leaf nodes as squares. Each node is displayed with a binary string identifier corresponding to its position in the tree, obtained by appending a bit to its parent identifier (0=choose left or top, 1=choose right or bottom). *(Bottom)*The tree divides space into regions. The 2D plane shows how a decision tree might divide $\mathbb{R}^2$. The nodes of the tree are plotted in this plane, with each internal node drawn along the dividing line it uses to categorize examples, and leaf nodes drawn in the center of the region of examples they receive. The result is a piecewise-constant function, with one piece per leaf. Each leaf requires at least one training example to define, so it is not possible for the decision tree to learn a function that has more local maxima than the number of training examples.

Another type of learning algorithm that also breaks the input space into regions and has separate parameters for each region is the **decision tree** (Breiman *et al.*, 1984) and its many variants. As shown in figure 5.7, each node of the decision tree is associated with a region in the input space, and internal nodes break that region into one sub-region for each child of the node (typically using an axis-aligned cut). Space is thus sub-divided into non-overlapping regions, with a one-to-one correspondence between leaf nodes and input regions. Each leaf node usually maps every point in its input region to the same output. Decision trees are usually trained with specialized algorithms that are beyond the scope of this book. The learning algorithm can be considered non-parametric if it is allowed to learn a tree of arbitrary size, though decision trees are usually regularized with size constraints that turn them into parametric models in practice. Decision trees as they are typically used, with axis-aligned splits and constant outputs within each node, struggle to solve some problems that are easy even for logistic regression. For example, if we have a two-class problem and the positive class occurs wherever $x_2 > x_1$, the decision boundary is not axis-aligned. The decision tree will thus need to approximate the decision boundary with many nodes, implementing a step function that constantly walks back and forth across the true decision function with axis-aligned steps.

As we have seen, nearest neighbor predictors and decision trees have many limitations. Nonetheless, they are useful learning algorithms when computational resources are constrained. We can also build intuition for more sophisticated learning algorithms by thinking about the similarities and differences between sophisticated algorithms and k-NN or decision tree baselines.

See Murphy (2012), Bishop (2006), Hastie *et al.* (2001) or other machine learning textbooks for more material on traditional supervised learning algorithms.

5.8 Unsupervised Learning Algorithms

Recall from section 5.1.3 that unsupervised algorithms are those that experience only "features" but not a supervision signal. The distinction between supervised and unsupervised algorithms is not formally and rigidly defined because there is no objective test for distinguishing whether a value is a feature or a target provided by a supervisor. Informally, unsupervised learning refers to most attempts to extract information from a distribution that do not require human labor to annotate examples. The term is usually associated with density estimation, learning to draw samples from a distribution, learning to denoise data from some distribution, finding a manifold that the data lies near, or clustering the data into groups of

related examples.

A classic unsupervised learning task is to find the "best" representation of the data. By 'best' we can mean different things, but generally speaking we are looking for a representation that preserves as much information about $\boldsymbol{x}$ as possible while obeying some penalty or constraint aimed at keeping the representation *simpler* or more accessible than $\boldsymbol{x}$ itself.

There are multiple ways of defining a *simpler* representation. Three of the most common include lower dimensional representations, sparse representations and independent representations. Low-dimensional representations attempt to compress as much information about x as possible in a smaller representation. Sparse representations (Barlow, 1989; Olshausen and Field, 1996; Hinton and Ghahramani, 1997) embed the dataset into a representation whose entries are mostly zeroes for most inputs. The use of sparse representations typically requires increasing the dimensionality of the representation, so that the representation becoming mostly zeroes does not discard too much information. This results in an overall structure of the representation that tends to distribute data along the axes of the representation space. Independent representations attempt to *disentangle* the sources of variation underlying the data distribution such that the dimensions of the representation are statistically independent.

Of course these three criteria are certainly not mutually exclusive. Low-dimensional representations often yield elements that have fewer or weaker dependencies than the original high-dimensional data. This is because one way to reduce the size of a representation is to find and remove redundancies. Identifying and removing more redundancy allows the dimensionality reduction algorithm to achieve more compression while discarding less information.

The notion of representation is one of the central themes of deep learning and therefore one of the central themes in this book. In this section, we develop some simple examples of representation learning algorithms. Together, these example algorithms show how to operationalize all three of the criteria above. Most of the remaining chapters introduce additional representation learning algorithms that develop these criteria in different ways or introduce other criteria.

5.8.1 Principal Components Analysis

In section 2.12, we saw that the principal components analysis algorithm provides a means of compressing data. We can also view PCA as an unsupervised learning algorithm that learns a representation of data. This representation is based on two of the criteria for a simple representation described above. PCA learns a

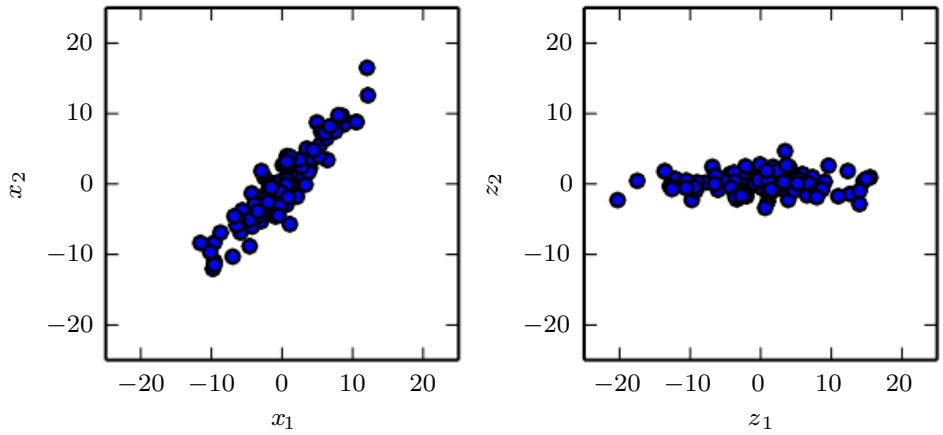

Figure 5.8: PCA learns a linear projection that aligns the direction of greatest variance with the axes of the new space. *(Left)*The original data consists of samples of $\boldsymbol{x}$. In this space, the variance might occur along directions that are not axis-aligned. *(Right)*The transformed data $\boldsymbol{z} = \boldsymbol{x}^\top \boldsymbol{W}$ now varies most along the axis z_1. The direction of second most variance is now along z_2.

representation that has lower dimensionality than the original input. It also learns a representation whose elements have no linear correlation with each other. This is a first step toward the criterion of learning representations whose elements are statistically independent. To achieve full independence, a representation learning algorithm must also remove the nonlinear relationships between variables.

PCA learns an orthogonal, linear transformation of the data that projects an input $\boldsymbol{x}$ to a representation $\boldsymbol{z}$ as shown in figure 5.8. In section 2.12, we saw that we could learn a one-dimensional representation that best reconstructs the original data (in the sense of mean squared error) and that this representation actually corresponds to the first principal component of the data. Thus we can use PCA as a simple and effective dimensionality reduction method that preserves as much of the information in the data as possible (again, as measured by least-squares reconstruction error). In the following, we will study how the PCA representation decorrelates the original data representation $\boldsymbol{X}$.

Let us consider the $m \times n$-dimensional design matrix $\boldsymbol{X}$. We will assume that the data has a mean of zero, $\mathbb{E}[\boldsymbol{x}] = \boldsymbol{0}$. If this is not the case, the data can easily be centered by subtracting the mean from all examples in a preprocessing step.

The unbiased sample covariance matrix associated with $\boldsymbol{X}$ is given by:

$$\text{Var}[\boldsymbol{x}] = \frac{1}{m-1} \boldsymbol{X}^\top \boldsymbol{X}. \tag{5.85}$$

PCA finds a representation (through linear transformation) $\boldsymbol{z} = \boldsymbol{x}^\top \boldsymbol{W}$ where $\text{Var}[\boldsymbol{z}]$ is diagonal.

In section 2.12, we saw that the principal components of a design matrix $\boldsymbol{X}$ are given by the eigenvectors of $\boldsymbol{X}^\top \boldsymbol{X}$. From this view,

$$\boldsymbol{X}^\top \boldsymbol{X} = \boldsymbol{W} \boldsymbol{\Lambda} \boldsymbol{W}^\top. \tag{5.86}$$

In this section, we exploit an alternative derivation of the principal components. The principal components may also be obtained via the singular value decomposition. Specifically, they are the right singular vectors of $\boldsymbol{X}$. To see this, let $\boldsymbol{W}$ be the right singular vectors in the decomposition $\boldsymbol{X} = \boldsymbol{U} \boldsymbol{\Sigma} \boldsymbol{W}^\top$. We then recover the original eigenvector equation with $\boldsymbol{W}$ as the eigenvector basis:

$$\boldsymbol{X}^\top \boldsymbol{X} = \left(\boldsymbol{U} \boldsymbol{\Sigma} \boldsymbol{W}^\top \right)^\top \boldsymbol{U} \boldsymbol{\Sigma} \boldsymbol{W}^\top = \boldsymbol{W} \boldsymbol{\Sigma}^2 \boldsymbol{W}^\top. \tag{5.87}$$

The SVD is helpful to show that PCA results in a diagonal $\text{Var}\,[z]$. Using the SVD of $\boldsymbol{X}$, we can express the variance of $\boldsymbol{X}$ as:

$$\begin{aligned}
\text{Var}[\boldsymbol{x}] &= \frac{1}{m-1} \boldsymbol{X}^\top \boldsymbol{X} && (5.88)\\
&= \frac{1}{m-1} (\boldsymbol{U} \boldsymbol{\Sigma} \boldsymbol{W}^\top)^\top \boldsymbol{U} \boldsymbol{\Sigma} \boldsymbol{W}^\top && (5.89)\\
&= \frac{1}{m-1} \boldsymbol{W} \boldsymbol{\Sigma}^\top \boldsymbol{U}^\top \boldsymbol{U} \boldsymbol{\Sigma} \boldsymbol{W}^\top && (5.90)\\
&= \frac{1}{m-1} \boldsymbol{W} \boldsymbol{\Sigma}^2 \boldsymbol{W}^\top, && (5.91)
\end{aligned}$$

where we use the fact that $\boldsymbol{U}^\top \boldsymbol{U} = \boldsymbol{I}$ because the $\boldsymbol{U}$ matrix of the singular value decomposition is defined to be orthogonal. This shows that if we take $\boldsymbol{z} = \boldsymbol{x}^\top \boldsymbol{W}$, we can ensure that the covariance of $\boldsymbol{z}$ is diagonal as required:

$$\begin{aligned}
\text{Var}[\boldsymbol{z}] &= \frac{1}{m-1} \boldsymbol{Z}^\top \boldsymbol{Z} && (5.92)\\
&= \frac{1}{m-1} \boldsymbol{W}^\top \boldsymbol{X}^\top \boldsymbol{X} \boldsymbol{W} && (5.93)\\
&= \frac{1}{m-1} \boldsymbol{W}^\top \boldsymbol{W} \boldsymbol{\Sigma}^2 \boldsymbol{W}^\top \boldsymbol{W} && (5.94)\\
&= \frac{1}{m-1} \boldsymbol{\Sigma}^2, && (5.95)
\end{aligned}$$

where this time we use the fact that $\boldsymbol{W}^\top \boldsymbol{W} = \boldsymbol{I}$, again from the definition of the SVD.

The above analysis shows that when we project the data $\boldsymbol{x}$ to $\boldsymbol{z}$, via the linear transformation $\boldsymbol{W}$, the resulting representation has a diagonal covariance matrix (as given by $\boldsymbol{\Sigma}^2$) which immediately implies that the individual elements of $\boldsymbol{z}$ are mutually uncorrelated.

This ability of PCA to transform data into a representation where the elements are mutually uncorrelated is a very important property of PCA. It is a simple example of a representation that attempts to *disentangle the unknown factors of variation* underlying the data. In the case of PCA, this disentangling takes the form of finding a rotation of the input space (described by $\boldsymbol{W}$) that aligns the principal axes of variance with the basis of the new representation space associated with $\boldsymbol{z}$.

While correlation is an important category of dependency between elements of the data, we are also interested in learning representations that disentangle more complicated forms of feature dependencies. For this, we will need more than what can be done with a simple linear transformation.

5.8.2 k-means Clustering

Another example of a simple representation learning algorithm is k-means clustering. The k-means clustering algorithm divides the training set into k different clusters of examples that are near each other. We can thus think of the algorithm as providing a k-dimensional one-hot code vector $\boldsymbol{h}$ representing an input $\boldsymbol{x}$. If $\boldsymbol{x}$ belongs to cluster i, then $h_i = 1$ and all other entries of the representation $\boldsymbol{h}$ are zero.

The one-hot code provided by k-means clustering is an example of a sparse representation, because the majority of its entries are zero for every input. Later, we will develop other algorithms that learn more flexible sparse representations, where more than one entry can be non-zero for each input $\boldsymbol{x}$. One-hot codes are an extreme example of sparse representations that lose many of the benefits of a distributed representation. The one-hot code still confers some statistical advantages (it naturally conveys the idea that all examples in the same cluster are similar to each other) and it confers the computational advantage that the entire representation may be captured by a single integer.

The k-means algorithm works by initializing k different centroids $\{\boldsymbol{\mu}^{(1)}, \ldots, \boldsymbol{\mu}^{(k)}\}$ to different values, then alternating between two different steps until convergence. In one step, each training example is assigned to cluster i, where i is the index of the nearest centroid $\boldsymbol{\mu}^{(i)}$. In the other step, each centroid $\boldsymbol{\mu}^{(i)}$ is updated to the mean of all training examples $\boldsymbol{x}^{(j)}$ assigned to cluster i.

One difficulty pertaining to clustering is that the clustering problem is inherently ill-posed, in the sense that there is no single criterion that measures how well a clustering of the data corresponds to the real world. We can measure properties of the clustering such as the average Euclidean distance from a cluster centroid to the members of the cluster. This allows us to tell how well we are able to reconstruct the training data from the cluster assignments. We do not know how well the cluster assignments correspond to properties of the real world. Moreover, there may be many different clusterings that all correspond well to some property of the real world. We may hope to find a clustering that relates to one feature but obtain a different, equally valid clustering that is not relevant to our task. For example, suppose that we run two clustering algorithms on a dataset consisting of images of red trucks, images of red cars, images of gray trucks, and images of gray cars. If we ask each clustering algorithm to find two clusters, one algorithm may find a cluster of cars and a cluster of trucks, while another may find a cluster of red vehicles and a cluster of gray vehicles. Suppose we also run a third clustering algorithm, which is allowed to determine the number of clusters. This may assign the examples to four clusters, red cars, red trucks, gray cars, and gray trucks. This new clustering now at least captures information about both attributes, but it has lost information about similarity. Red cars are in a different cluster from gray cars, just as they are in a different cluster from gray trucks. The output of the clustering algorithm does not tell us that red cars are more similar to gray cars than they are to gray trucks. They are different from both things, and that is all we know.

These issues illustrate some of the reasons that we may prefer a distributed representation to a one-hot representation. A distributed representation could have two attributes for each vehicle—one representing its color and one representing whether it is a car or a truck. It is still not entirely clear what the optimal distributed representation is (how can the learning algorithm know whether the two attributes we are interested in are color and car-versus-truck rather than manufacturer and age?) but having many attributes reduces the burden on the algorithm to guess which single attribute we care about, and allows us to measure similarity between objects in a fine-grained way by comparing many attributes instead of just testing whether one attribute matches.

5.9 Stochastic Gradient Descent

Nearly all of deep learning is powered by one very important algorithm: **stochastic gradient descent** or SGD. Stochastic gradient descent is an extension of the

gradient descent algorithm introduced in section 4.3.

A recurring problem in machine learning is that large training sets are necessary for good generalization, but large training sets are also more computationally expensive.

The cost function used by a machine learning algorithm often decomposes as a sum over training examples of some per-example loss function. For example, the negative conditional log-likelihood of the training data can be written as

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},\mathrm{y}\sim\hat{p}_{\mathrm{data}}} L(\boldsymbol{x}, y, \boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}) \tag{5.96}$$

where L is the per-example loss $L(\boldsymbol{x}, y, \boldsymbol{\theta}) = -\log p(y \mid \boldsymbol{x}; \boldsymbol{\theta})$.

For these additive cost functions, gradient descent requires computing

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}). \tag{5.97}$$

The computational cost of this operation is $O(m)$. As the training set size grows to billions of examples, the time to take a single gradient step becomes prohibitively long.

The insight of stochastic gradient descent is that the gradient is an expectation. The expectation may be approximately estimated using a small set of samples. Specifically, on each step of the algorithm, we can sample a **minibatch** of examples $\mathbb{B} = \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m')}\}$ drawn uniformly from the training set. The minibatch size m' is typically chosen to be a relatively small number of examples, ranging from 1 to a few hundred. Crucially, m' is usually held fixed as the training set size m grows. We may fit a training set with billions of examples using updates computed on only a hundred examples.

The estimate of the gradient is formed as

$$\boldsymbol{g} = \frac{1}{m'}\nabla_{\boldsymbol{\theta}}\sum_{i=1}^{m'} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}). \tag{5.98}$$

using examples from the minibatch $\mathbb{B}$. The stochastic gradient descent algorithm then follows the estimated gradient downhill:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \boldsymbol{g}, \tag{5.99}$$

where ϵ is the learning rate.

Gradient descent in general has often been regarded as slow or unreliable. In the past, the application of gradient descent to non-convex optimization problems was regarded as foolhardy or unprincipled. Today, we know that the machine learning models described in part II work very well when trained with gradient descent. The optimization algorithm may not be guaranteed to arrive at even a local minimum in a reasonable amount of time, but it often finds a very low value of the cost function quickly enough to be useful.

Stochastic gradient descent has many important uses outside the context of deep learning. It is the main way to train large linear models on very large datasets. For a fixed model size, the cost per SGD update does not depend on the training set size m. In practice, we often use a larger model as the training set size increases, but we are not forced to do so. The number of updates required to reach convergence usually increases with training set size. However, as m approaches infinity, the model will eventually converge to its best possible test error before SGD has sampled every example in the training set. Increasing m further will not extend the amount of training time needed to reach the model's best possible test error. From this point of view, one can argue that the asymptotic cost of training a model with SGD is $O(1)$ as a function of m.

Prior to the advent of deep learning, the main way to learn nonlinear models was to use the kernel trick in combination with a linear model. Many kernel learning algorithms require constructing an $m \times m$ matrix $G_{i,j} = k(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)})$. Constructing this matrix has computational cost $O(m^2)$, which is clearly undesirable for datasets with billions of examples. In academia, starting in 2006, deep learning was initially interesting because it was able to generalize to new examples better than competing algorithms when trained on medium-sized datasets with tens of thousands of examples. Soon after, deep learning garnered additional interest in industry, because it provided a scalable way of training nonlinear models on large datasets.

Stochastic gradient descent and many enhancements to it are described further in chapter 8.

5.10 Building a Machine Learning Algorithm

Nearly all deep learning algorithms can be described as particular instances of a fairly simple recipe: combine a specification of a dataset, a cost function, an optimization procedure and a model.

For example, the linear regression algorithm combines a dataset consisting of

$\boldsymbol{X}$ and $\boldsymbol{y}$, the cost function

$$J(\boldsymbol{w}, b) = -\mathbb{E}_{\mathbf{x},\mathrm{y}\sim\hat{p}_{\mathrm{data}}} \log p_{\mathrm{model}}(y \mid \boldsymbol{x}), \tag{5.100}$$

the model specification $p_{\mathrm{model}}(y \mid \boldsymbol{x}) = \mathcal{N}(y; \boldsymbol{x}^\top \boldsymbol{w} + b, 1)$, and, in most cases, the optimization algorithm defined by solving for where the gradient of the cost is zero using the normal equations.

By realizing that we can replace any of these components mostly independently from the others, we can obtain a very wide variety of algorithms.

The cost function typically includes at least one term that causes the learning process to perform statistical estimation. The most common cost function is the negative log-likelihood, so that minimizing the cost function causes maximum likelihood estimation.

The cost function may also include additional terms, such as regularization terms. For example, we can add weight decay to the linear regression cost function to obtain

$$J(\boldsymbol{w}, b) = \lambda ||\boldsymbol{w}||_2^2 - \mathbb{E}_{\mathbf{x},\mathrm{y}\sim\hat{p}_{\mathrm{data}}} \log p_{\mathrm{model}}(y \mid \boldsymbol{x}). \tag{5.101}$$

This still allows closed-form optimization.

If we change the model to be nonlinear, then most cost functions can no longer be optimized in closed form. This requires us to choose an iterative numerical optimization procedure, such as gradient descent.

The recipe for constructing a learning algorithm by combining models, costs, and optimization algorithms supports both supervised and unsupervised learning. The linear regression example shows how to support supervised learning. Unsupervised learning can be supported by defining a dataset that contains only $\boldsymbol{X}$ and providing an appropriate unsupervised cost and model. For example, we can obtain the first PCA vector by specifying that our loss function is

$$J(\boldsymbol{w}) = \mathbb{E}_{\mathbf{x}\sim\hat{p}_{\mathrm{data}}} ||\boldsymbol{x} - r(\boldsymbol{x}; \boldsymbol{w})||_2^2 \tag{5.102}$$

while our model is defined to have $\boldsymbol{w}$ with norm one and reconstruction function $r(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} \boldsymbol{w}$.

In some cases, the cost function may be a function that we cannot actually evaluate, for computational reasons. In these cases, we can still approximately minimize it using iterative numerical optimization so long as we have some way of approximating its gradients.

Most machine learning algorithms make use of this recipe, though it may not immediately be obvious. If a machine learning algorithm seems especially unique or

hand-designed, it can usually be understood as using a special-case optimizer. Some models such as decision trees or k-means require special-case optimizers because their cost functions have flat regions that make them inappropriate for minimization by gradient-based optimizers. Recognizing that most machine learning algorithms can be described using this recipe helps to see the different algorithms as part of a taxonomy of methods for doing related tasks that work for similar reasons, rather than as a long list of algorithms that each have separate justifications.

5.11 Challenges Motivating Deep Learning

The simple machine learning algorithms described in this chapter work very well on a wide variety of important problems. However, they have not succeeded in solving the central problems in AI, such as recognizing speech or recognizing objects.

The development of deep learning was motivated in part by the failure of traditional algorithms to generalize well on such AI tasks.

This section is about how the challenge of generalizing to new examples becomes exponentially more difficult when working with high-dimensional data, and how the mechanisms used to achieve generalization in traditional machine learning are insufficient to learn complicated functions in high-dimensional spaces. Such spaces also often impose high computational costs. Deep learning was designed to overcome these and other obstacles.

5.11.1 The Curse of Dimensionality

Many machine learning problems become exceedingly difficult when the number of dimensions in the data is high. This phenomenon is known as the **curse of dimensionality**. Of particular concern is that the number of possible distinct configurations of a set of variables increases exponentially as the number of variables increases.

Figure 5.9: As the number of relevant dimensions of the data increases (from left to right), the number of configurations of interest may grow exponentially. *(Left)*In this one-dimensional example, we have one variable for which we only care to distinguish 10 regions of interest. With enough examples falling within each of these regions (each region corresponds to a cell in the illustration), learning algorithms can easily generalize correctly. A straightforward way to generalize is to estimate the value of the target function within each region (and possibly interpolate between neighboring regions). *(Center)*With 2 dimensions it is more difficult to distinguish 10 different values of each variable. We need to keep track of up to 10×10=100 regions, and we need at least that many examples to cover all those regions. *(Right)*With 3 dimensions this grows to $10^3 = 1000$ regions and at least that many examples. For d dimensions and v values to be distinguished along each axis, we seem to need $O(v^d)$ regions and examples. This is an instance of the curse of dimensionality. Figure graciously provided by Nicolas Chapados.

The curse of dimensionality arises in many places in computer science, and especially so in machine learning.

One challenge posed by the curse of dimensionality is a statistical challenge. As illustrated in figure 5.9, a statistical challenge arises because the number of possible configurations of $\boldsymbol{x}$ is much larger than the number of training examples. To understand the issue, let us consider that the input space is organized into a grid, like in the figure. We can describe low-dimensional space with a low number of grid cells that are mostly occupied by the data. When generalizing to a new data point, we can usually tell what to do simply by inspecting the training examples that lie in the same cell as the new input. For example, if estimating the probability density at some point $\boldsymbol{x}$, we can just return the number of training examples in the same unit volume cell as $\boldsymbol{x}$, divided by the total number of training examples. If we wish to classify an example, we can return the most common class of training examples in the same cell. If we are doing regression we can average the target values observed over the examples in that cell. But what about the cells for which we have seen no example? Because in high-dimensional spaces the number of configurations is huge, much larger than our number of examples, a typical grid cell has no training example associated with it. How could we possibly say something

meaningful about these new configurations? Many traditional machine learning algorithms simply assume that the output at a new point should be approximately the same as the output at the nearest training point.

5.11.2 Local Constancy and Smoothness Regularization

In order to generalize well, machine learning algorithms need to be guided by prior beliefs about what kind of function they should learn. Previously, we have seen these priors incorporated as explicit beliefs in the form of probability distributions over parameters of the model. More informally, we may also discuss prior beliefs as directly influencing the *function* itself and only indirectly acting on the parameters via their effect on the function. Additionally, we informally discuss prior beliefs as being expressed implicitly, by choosing algorithms that are biased toward choosing some class of functions over another, even though these biases may not be expressed (or even possible to express) in terms of a probability distribution representing our degree of belief in various functions.

Among the most widely used of these implicit "priors" is the **smoothness prior** or **local constancy prior**. This prior states that the function we learn should not change very much within a small region.

Many simpler algorithms rely exclusively on this prior to generalize well, and as a result they fail to scale to the statistical challenges involved in solving AI-level tasks. Throughout this book, we will describe how deep learning introduces additional (explicit and implicit) priors in order to reduce the generalization error on sophisticated tasks. Here, we explain why the smoothness prior alone is insufficient for these tasks.

There are many different ways to implicitly or explicitly express a prior belief that the learned function should be smooth or locally constant. All of these different methods are designed to encourage the learning process to learn a function f^* that satisfies the condition

$$f^*(\boldsymbol{x}) \approx f^*(\boldsymbol{x} + \epsilon) \tag{5.103}$$

for most configurations $\boldsymbol{x}$ and small change ϵ. In other words, if we know a good answer for an input $\boldsymbol{x}$ (for example, if $\boldsymbol{x}$ is a labeled training example) then that answer is probably good in the neighborhood of $\boldsymbol{x}$. If we have several good answers in some neighborhood we would combine them (by some form of averaging or interpolation) to produce an answer that agrees with as many of them as much as possible.

An extreme example of the local constancy approach is the k-nearest neighbors family of learning algorithms. These predictors are literally constant over each

region containing all the points $\boldsymbol{x}$ that have the same set of k nearest neighbors in the training set. For $k = 1$, the number of distinguishable regions cannot be more than the number of training examples.

While the k-nearest neighbors algorithm copies the output from nearby training examples, most kernel machines interpolate between training set outputs associated with nearby training examples. An important class of kernels is the family of **local kernels** where $k(\boldsymbol{u}, \boldsymbol{v})$ is large when $\boldsymbol{u} = \boldsymbol{v}$ and decreases as $\boldsymbol{u}$ and $\boldsymbol{v}$ grow farther apart from each other. A local kernel can be thought of as a similarity function that performs template matching, by measuring how closely a test example $\boldsymbol{x}$ resembles each training example $\boldsymbol{x}^{(i)}$. Much of the modern motivation for deep learning is derived from studying the limitations of local template matching and how deep models are able to succeed in cases where local template matching fails (Bengio *et al.*, 2006b).

Decision trees also suffer from the limitations of exclusively smoothness-based learning because they break the input space into as many regions as there are leaves and use a separate parameter (or sometimes many parameters for extensions of decision trees) in each region. If the target function requires a tree with at least n leaves to be represented accurately, then at least n training examples are required to fit the tree. A multiple of n is needed to achieve some level of statistical confidence in the predicted output.

In general, to distinguish $O(k)$ regions in input space, all of these methods require $O(k)$ examples. Typically there are $O(k)$ parameters, with $O(1)$ parameters associated with each of the $O(k)$ regions. The case of a nearest neighbor scenario, where each training example can be used to define at most one region, is illustrated in figure 5.10.

Is there a way to represent a complex function that has many more regions to be distinguished than the number of training examples? Clearly, assuming only smoothness of the underlying function will not allow a learner to do that. For example, imagine that the target function is a kind of checkerboard. A checkerboard contains many variations but there is a simple structure to them. Imagine what happens when the number of training examples is substantially smaller than the number of black and white squares on the checkerboard. Based on only local generalization and the smoothness or local constancy prior, we would be guaranteed to correctly guess the color of a new point if it lies within the same checkerboard square as a training example. There is no guarantee that the learner could correctly extend the checkerboard pattern to points lying in squares that do not contain training examples. With this prior alone, the only information that an example tells us is the color of its square, and the only way to get the colors of the

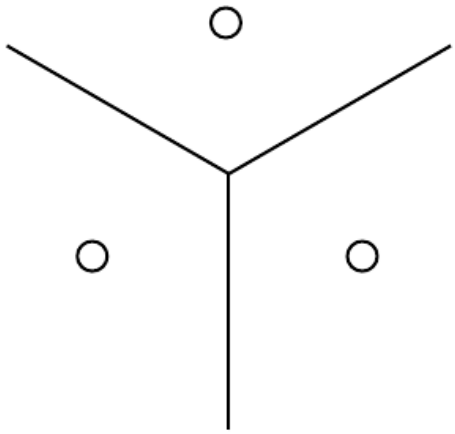

Figure 5.10: Illustration of how the nearest neighbor algorithm breaks up the input space into regions. An example (represented here by a circle) within each region defines the region boundary (represented here by the lines). The y value associated with each example defines what the output should be for all points within the corresponding region. The regions defined by nearest neighbor matching form a geometric pattern called a Voronoi diagram. The number of these contiguous regions cannot grow faster than the number of training examples. While this figure illustrates the behavior of the nearest neighbor algorithm specifically, other machine learning algorithms that rely exclusively on the local smoothness prior for generalization exhibit similar behaviors: each training example only informs the learner about how to generalize in some neighborhood immediately surrounding that example.

entire checkerboard right is to cover each of its cells with at least one example.

The smoothness assumption and the associated non-parametric learning algorithms work extremely well so long as there are enough examples for the learning algorithm to observe high points on most peaks and low points on most valleys of the true underlying function to be learned. This is generally true when the function to be learned is smooth enough and varies in few enough dimensions. In high dimensions, even a very smooth function can change smoothly but in a different way along each dimension. If the function additionally behaves differently in different regions, it can become extremely complicated to describe with a set of training examples. If the function is complicated (we want to distinguish a huge number of regions compared to the number of examples), is there any hope to generalize well?

The answer to both of these questions—whether it is possible to represent a complicated function efficiently, and whether it is possible for the estimated function to generalize well to new inputs—is yes. The key insight is that a very large number of regions, e.g., $O(2^k)$, can be defined with $O(k)$ examples, so long as we introduce some dependencies between the regions via additional assumptions about the underlying data generating distribution. In this way, we can actually generalize non-locally (Bengio and Monperrus, 2005; Bengio *et al.*, 2006c). Many different deep learning algorithms provide implicit or explicit assumptions that are reasonable for a broad range of AI tasks in order to capture these advantages.

Other approaches to machine learning often make stronger, task-specific assumptions. For example, we could easily solve the checkerboard task by providing the assumption that the target function is periodic. Usually we do not include such strong, task-specific assumptions into neural networks so that they can generalize to a much wider variety of structures. AI tasks have structure that is much too complex to be limited to simple, manually specified properties such as periodicity, so we want learning algorithms that embody more general-purpose assumptions. The core idea in deep learning is that we assume that the data was generated by the *composition of factors* or features, potentially at multiple levels in a hierarchy. Many other similarly generic assumptions can further improve deep learning algorithms. These apparently mild assumptions allow an exponential gain in the relationship between the number of examples and the number of regions that can be distinguished. These exponential gains are described more precisely in sections 6.4.1, 15.4 and 15.5. The exponential advantages conferred by the use of deep, distributed representations counter the exponential challenges posed by the curse of dimensionality.

5.11.3 Manifold Learning

An important concept underlying many ideas in machine learning is that of a manifold.

A **manifold** is a connected region. Mathematically, it is a set of points, associated with a neighborhood around each point. From any given point, the manifold locally appears to be a Euclidean space. In everyday life, we experience the surface of the world as a 2-D plane, but it is in fact a spherical manifold in 3-D space.

The definition of a neighborhood surrounding each point implies the existence of transformations that can be applied to move on the manifold from one position to a neighboring one. In the example of the world's surface as a manifold, one can walk north, south, east, or west.

Although there is a formal mathematical meaning to the term "manifold," in machine learning it tends to be used more loosely to designate a connected set of points that can be approximated well by considering only a small number of degrees of freedom, or dimensions, embedded in a higher-dimensional space. Each dimension corresponds to a local direction of variation. See figure 5.11 for an example of training data lying near a one-dimensional manifold embedded in two-dimensional space. In the context of machine learning, we allow the dimensionality of the manifold to vary from one point to another. This often happens when a manifold intersects itself. For example, a figure eight is a manifold that has a single dimension in most places but two dimensions at the intersection at the center.

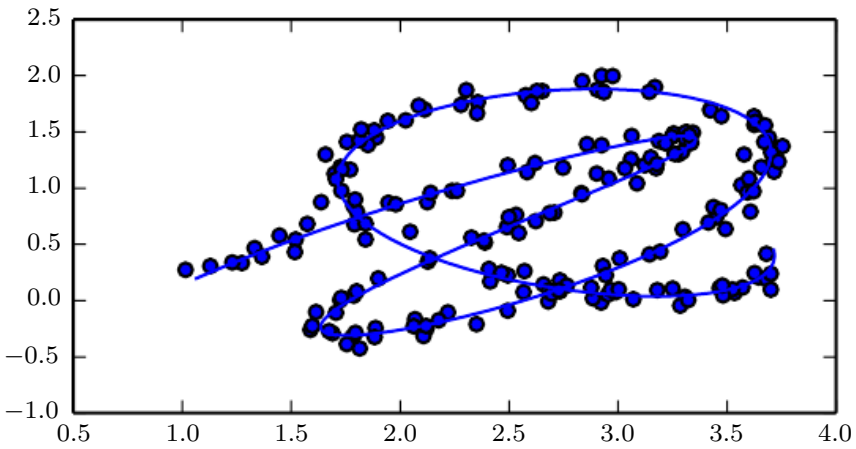

Figure 5.11: Data sampled from a distribution in a two-dimensional space that is actually concentrated near a one-dimensional manifold, like a twisted string. The solid line indicates the underlying manifold that the learner should infer.

Many machine learning problems seem hopeless if we expect the machine learning algorithm to learn functions with interesting variations across all of $\mathbb{R}^n$. **Manifold learning** algorithms surmount this obstacle by assuming that most of $\mathbb{R}^n$ consists of invalid inputs, and that interesting inputs occur only along a collection of manifolds containing a small subset of points, with interesting variations in the output of the learned function occurring only along directions that lie on the manifold, or with interesting variations happening only when we move from one manifold to another. Manifold learning was introduced in the case of continuous-valued data and the unsupervised learning setting, although this probability concentration idea can be generalized to both discrete data and the supervised learning setting: the key assumption remains that probability mass is highly concentrated.

The assumption that the data lies along a low-dimensional manifold may not always be correct or useful. We argue that in the context of AI tasks, such as those that involve processing images, sounds, or text, the manifold assumption is at least approximately correct. The evidence in favor of this assumption consists of two categories of observations.

The first observation in favor of the **manifold hypothesis** is that the probability distribution over images, text strings, and sounds that occur in real life is highly concentrated. Uniform noise essentially never resembles structured inputs from these domains. Figure 5.12 shows how, instead, uniformly sampled points look like the patterns of static that appear on analog television sets when no signal is available. Similarly, if you generate a document by picking letters uniformly at random, what is the probability that you will get a meaningful English-language text? Almost zero, again, because most of the long sequences of letters do not correspond to a natural language sequence: the distribution of natural language sequences occupies a very small volume in the total space of sequences of letters.

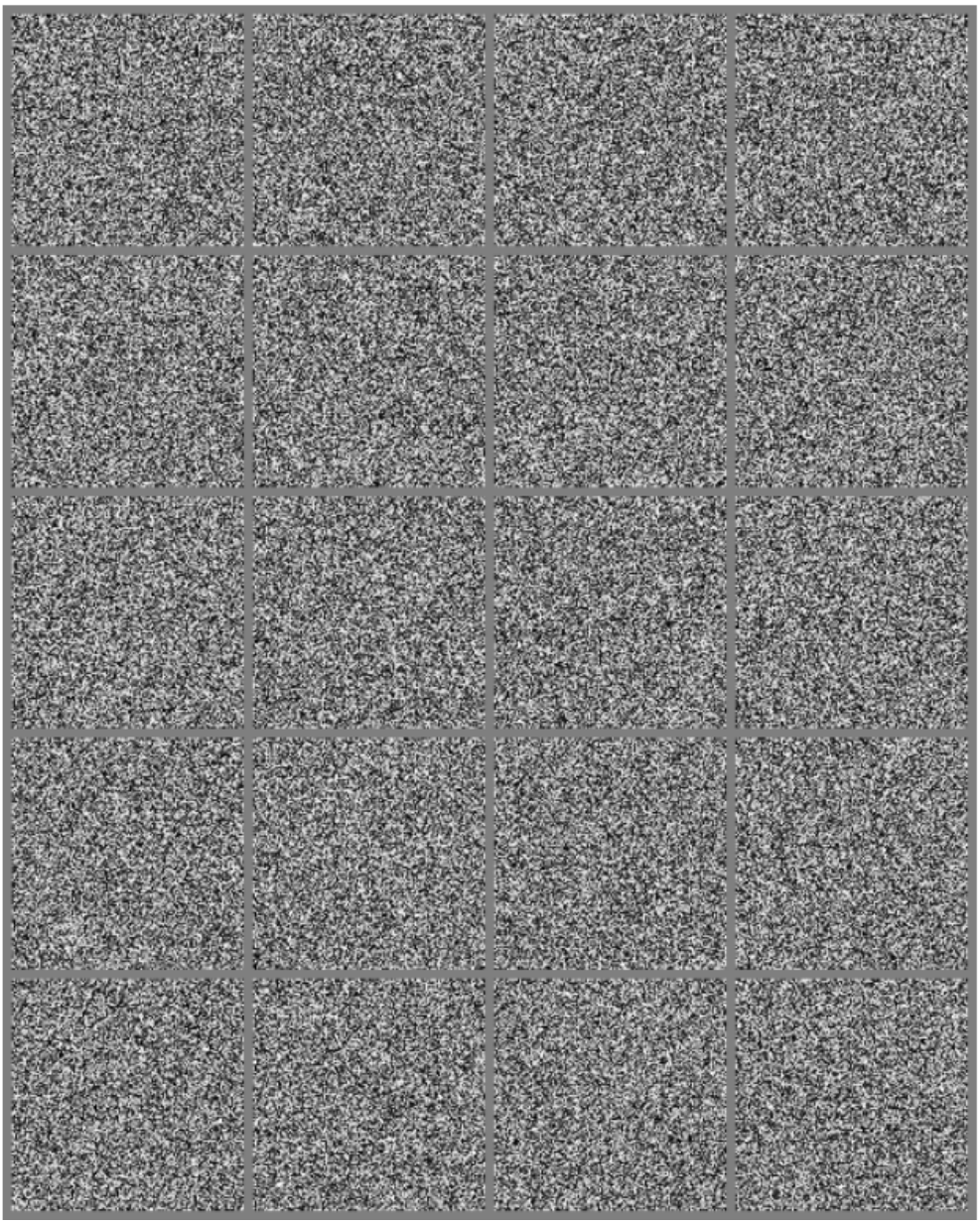

Figure 5.12: Sampling images uniformly at random (by randomly picking each pixel according to a uniform distribution) gives rise to noisy images. Although there is a non-zero probability to generate an image of a face or any other object frequently encountered in AI applications, we never actually observe this happening in practice. This suggests that the images encountered in AI applications occupy a negligible proportion of the volume of image space.

Of course, concentrated probability distributions are not sufficient to show that the data lies on a reasonably small number of manifolds. We must also establish that the examples we encounter are connected to each other by other

examples, with each example surrounded by other highly similar examples that may be reached by applying transformations to traverse the manifold. The second argument in favor of the manifold hypothesis is that we can also imagine such neighborhoods and transformations, at least informally. In the case of images, we can certainly think of many possible transformations that allow us to trace out a manifold in image space: we can gradually dim or brighten the lights, gradually move or rotate objects in the image, gradually alter the colors on the surfaces of objects, etc. It remains likely that there are multiple manifolds involved in most applications. For example, the manifold of images of human faces may not be connected to the manifold of images of cat faces.

These thought experiments supporting the manifold hypotheses convey some intuitive reasons supporting it. More rigorous experiments (Cayton, 2005; Narayanan and Mitter, 2010; Schölkopf *et al.*, 1998; Roweis and Saul, 2000; Tenenbaum *et al.*, 2000; Brand, 2003; Belkin and Niyogi, 2003; Donoho and Grimes, 2003; Weinberger and Saul, 2004) clearly support the hypothesis for a large class of datasets of interest in AI.

When the data lies on a low-dimensional manifold, it can be most natural for machine learning algorithms to represent the data in terms of coordinates on the manifold, rather than in terms of coordinates in $\mathbb{R}^n$. In everyday life, we can think of roads as 1-D manifolds embedded in 3-D space. We give directions to specific addresses in terms of address numbers along these 1-D roads, not in terms of coordinates in 3-D space. Extracting these manifold coordinates is challenging, but holds the promise to improve many machine learning algorithms. This general principle is applied in many contexts. Figure 5.13 shows the manifold structure of a dataset consisting of faces. By the end of this book, we will have developed the methods necessary to learn such a manifold structure. In figure 20.6, we will see how a machine learning algorithm can successfully accomplish this goal.

This concludes part I, which has provided the basic concepts in mathematics and machine learning which are employed throughout the remaining parts of the book. You are now prepared to embark upon your study of deep learning.

Figure 5.13: Training examples from the QMUL Multiview Face Dataset (Gong *et al.*, 2000) for which the subjects were asked to move in such a way as to cover the two-dimensional manifold corresponding to two angles of rotation. We would like learning algorithms to be able to discover and disentangle such manifold coordinates. Figure 20.6 illustrates such a feat.

Part II

Deep Networks: Modern Practices

This part of the book summarizes the state of modern deep learning as it is used to solve practical applications.

Deep learning has a long history and many aspirations. Several approaches have been proposed that have yet to entirely bear fruit. Several ambitious goals have yet to be realized. These less-developed branches of deep learning appear in the final part of the book.

This part focuses only on those approaches that are essentially working technologies that are already used heavily in industry.

Modern deep learning provides a very powerful framework for supervised learning. By adding more layers and more units within a layer, a deep network can represent functions of increasing complexity. Most tasks that consist of mapping an input vector to an output vector, and that are easy for a person to do rapidly, can be accomplished via deep learning, given sufficiently large models and sufficiently large datasets of labeled training examples. Other tasks, that can not be described as associating one vector to another, or that are difficult enough that a person would require time to think and reflect in order to accomplish the task, remain beyond the scope of deep learning for now.

This part of the book describes the core parametric function approximation technology that is behind nearly all modern practical applications of deep learning. We begin by describing the feedforward deep network model that is used to represent these functions. Next, we present advanced techniques for regularization and optimization of such models. Scaling these models to large inputs such as high resolution images or long temporal sequences requires specialization. We introduce the convolutional network for scaling to large images and the recurrent neural network for processing temporal sequences. Finally, we present general guidelines for the practical methodology involved in designing, building, and configuring an application involving deep learning, and review some of the applications of deep learning.

These chapters are the most important for a practitioner—someone who wants to begin implementing and using deep learning algorithms to solve real-world problems today.

Chapter 6

Deep Feedforward Networks

Deep feedforward networks, also often called **feedforward neural networks**, or **multilayer perceptrons** (MLPs), are the quintessential deep learning models. The goal of a feedforward network is to approximate some function f^*. For example, for a classifier, $y = f^*(\boldsymbol{x})$ maps an input $\boldsymbol{x}$ to a category y. A feedforward network defines a mapping $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ and learns the value of the parameters $\boldsymbol{\theta}$ that result in the best function approximation.

These models are called **feedforward** because information flows through the function being evaluated from $\boldsymbol{x}$, through the intermediate computations used to define f, and finally to the output $\boldsymbol{y}$. There are no **feedback** connections in which outputs of the model are fed back into itself. When feedforward neural networks are extended to include feedback connections, they are called **recurrent neural networks**, presented in chapter 10.

Feedforward networks are of extreme importance to machine learning practitioners. They form the basis of many important commercial applications. For example, the convolutional networks used for object recognition from photos are a specialized kind of feedforward network. Feedforward networks are a conceptual stepping stone on the path to recurrent networks, which power many natural language applications.

Feedforward neural networks are called **networks** because they are typically represented by composing together many different functions. The model is associated with a directed acyclic graph describing how the functions are composed together. For example, we might have three functions $f^{(1)}$, $f^{(2)}$, and $f^{(3)}$ connected in a chain, to form $f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$. These chain structures are the most commonly used structures of neural networks. In this case, $f^{(1)}$ is called the **first layer** of the network, $f^{(2)}$ is called the **second layer**, and so on. The overall

length of the chain gives the **depth** of the model. It is from this terminology that the name "deep learning" arises. The final layer of a feedforward network is called the **output layer**. During neural network training, we drive $f(\boldsymbol{x})$ to match $f^*(\boldsymbol{x})$. The training data provides us with noisy, approximate examples of $f^*(\boldsymbol{x})$ evaluated at different training points. Each example $\boldsymbol{x}$ is accompanied by a label $y \approx f^*(\boldsymbol{x})$. The training examples specify directly what the output layer must do at each point $\boldsymbol{x}$; it must produce a value that is close to y. The behavior of the other layers is not directly specified by the training data. The learning algorithm must decide how to use those layers to produce the desired output, but the training data does not say what each individual layer should do. Instead, the learning algorithm must decide how to use these layers to best implement an approximation of f^*. Because the training data does not show the desired output for each of these layers, these layers are called **hidden layers**.

Finally, these networks are called *neural* because they are loosely inspired by neuroscience. Each hidden layer of the network is typically vector-valued. The dimensionality of these hidden layers determines the **width** of the model. Each element of the vector may be interpreted as playing a role analogous to a neuron. Rather than thinking of the layer as representing a single vector-to-vector function, we can also think of the layer as consisting of many **units** that act in parallel, each representing a vector-to-scalar function. Each unit resembles a neuron in the sense that it receives input from many other units and computes its own activation value. The idea of using many layers of vector-valued representation is drawn from neuroscience. The choice of the functions $f^{(i)}(\boldsymbol{x})$ used to compute these representations is also loosely guided by neuroscientific observations about the functions that biological neurons compute. However, modern neural network research is guided by many mathematical and engineering disciplines, and the goal of neural networks is not to perfectly model the brain. It is best to think of feedforward networks as function approximation machines that are designed to achieve statistical generalization, occasionally drawing some insights from what we know about the brain, rather than as models of brain function.

One way to understand feedforward networks is to begin with linear models and consider how to overcome their limitations. Linear models, such as logistic regression and linear regression, are appealing because they may be fit efficiently and reliably, either in closed form or with convex optimization. Linear models also have the obvious defect that the model capacity is limited to linear functions, so the model cannot understand the interaction between any two input variables.

To extend linear models to represent nonlinear functions of $\boldsymbol{x}$, we can apply the linear model not to $\boldsymbol{x}$ itself but to a transformed input $\phi(\boldsymbol{x})$, where ϕ is a

nonlinear transformation. Equivalently, we can apply the kernel trick described in section 5.7.2, to obtain a nonlinear learning algorithm based on implicitly applying the ϕ mapping. We can think of ϕ as providing a set of features describing $\boldsymbol{x}$, or as providing a new representation for $\boldsymbol{x}$.

The question is then how to choose the mapping ϕ.

1. One option is to use a very generic ϕ, such as the infinite-dimensional ϕ that is implicitly used by kernel machines based on the RBF kernel. If $\phi(\boldsymbol{x})$ is of high enough dimension, we can always have enough capacity to fit the training set, but generalization to the test set often remains poor. Very generic feature mappings are usually based only on the principle of local smoothness and do not encode enough prior information to solve advanced problems.

2. Another option is to manually engineer ϕ. Until the advent of deep learning, this was the dominant approach. This approach requires decades of human effort for each separate task, with practitioners specializing in different domains such as speech recognition or computer vision, and with little transfer between domains.

3. The strategy of deep learning is to learn ϕ. In this approach, we have a model $y = f(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{w}) = \phi(\boldsymbol{x}; \boldsymbol{\theta})^\top \boldsymbol{w}$. We now have parameters $\boldsymbol{\theta}$ that we use to learn ϕ from a broad class of functions, and parameters $\boldsymbol{w}$ that map from $\phi(\boldsymbol{x})$ to the desired output. This is an example of a deep feedforward network, with ϕ defining a hidden layer. This approach is the only one of the three that gives up on the convexity of the training problem, but the benefits outweigh the harms. In this approach, we parametrize the representation as $\phi(\boldsymbol{x}; \boldsymbol{\theta})$ and use the optimization algorithm to find the $\boldsymbol{\theta}$ that corresponds to a good representation. If we wish, this approach can capture the benefit of the first approach by being highly generic—we do so by using a very broad family $\phi(\boldsymbol{x}; \boldsymbol{\theta})$. This approach can also capture the benefit of the second approach. Human practitioners can encode their knowledge to help generalization by designing families $\phi(\boldsymbol{x}; \boldsymbol{\theta})$ that they expect will perform well. The advantage is that the human designer only needs to find the right general function family rather than finding precisely the right function.

This general principle of improving models by learning features extends beyond the feedforward networks described in this chapter. It is a recurring theme of deep learning that applies to all of the kinds of models described throughout this book. Feedforward networks are the application of this principle to learning deterministic

mappings from $\boldsymbol{x}$ to $\boldsymbol{y}$ that lack feedback connections. Other models presented later will apply these principles to learning stochastic mappings, learning functions with feedback, and learning probability distributions over a single vector.

We begin this chapter with a simple example of a feedforward network. Next, we address each of the design decisions needed to deploy a feedforward network. First, training a feedforward network requires making many of the same design decisions as are necessary for a linear model: choosing the optimizer, the cost function, and the form of the output units. We review these basics of gradient-based learning, then proceed to confront some of the design decisions that are unique to feedforward networks. Feedforward networks have introduced the concept of a hidden layer, and this requires us to choose the **activation functions** that will be used to compute the hidden layer values. We must also design the architecture of the network, including how many layers the network should contain, how these layers should be connected to each other, and how many units should be in each layer. Learning in deep neural networks requires computing the gradients of complicated functions. We present the **back-propagation** algorithm and its modern generalizations, which can be used to efficiently compute these gradients. Finally, we close with some historical perspective.

6.1 Example: Learning XOR

To make the idea of a feedforward network more concrete, we begin with an example of a fully functioning feedforward network on a very simple task: learning the XOR function.

The XOR function ("exclusive or") is an operation on two binary values, x_1 and x_2. When exactly one of these binary values is equal to 1, the XOR function returns 1. Otherwise, it returns 0. The XOR function provides the target function $y = f^*(\boldsymbol{x})$ that we want to learn. Our model provides a function $y = f(\boldsymbol{x};\boldsymbol{\theta})$ and our learning algorithm will adapt the parameters $\boldsymbol{\theta}$ to make f as similar as possible to f^*.

In this simple example, we will not be concerned with statistical generalization. We want our network to perform correctly on the four points $\mathbb{X} = \{[0,0]^\top, [0,1]^\top, [1,0]^\top, \text{and } [1,1]^\top\}$. We will train the network on all four of these points. The only challenge is to fit the training set.

We can treat this problem as a regression problem and use a mean squared error loss function. We choose this loss function to simplify the math for this example as much as possible. In practical applications, MSE is usually not an

appropriate cost function for modeling binary data. More appropriate approaches are described in section 6.2.2.2.

Evaluated on our whole training set, the MSE loss function is

$$J(\boldsymbol{\theta}) = \frac{1}{4} \sum_{\boldsymbol{x} \in \mathbb{X}} (f^*(\boldsymbol{x}) - f(\boldsymbol{x}; \boldsymbol{\theta}))^2 . \tag{6.1}$$

Now we must choose the form of our model, $f(\boldsymbol{x}; \boldsymbol{\theta})$. Suppose that we choose a linear model, with $\boldsymbol{\theta}$ consisting of $\boldsymbol{w}$ and b. Our model is defined to be

$$f(\boldsymbol{x}; \boldsymbol{w}, b) = \boldsymbol{x}^\top \boldsymbol{w} + b. \tag{6.2}$$

We can minimize $J(\boldsymbol{\theta})$ in closed form with respect to $\boldsymbol{w}$ and b using the normal equations.

After solving the normal equations, we obtain $\boldsymbol{w} = \boldsymbol{0}$ and $b = \frac{1}{2}$. The linear model simply outputs 0.5 everywhere. Why does this happen? Figure 6.1 shows how a linear model is not able to represent the XOR function. One way to solve this problem is to use a model that learns a different feature space in which a linear model is able to represent the solution.

Specifically, we will introduce a very simple feedforward network with one hidden layer containing two hidden units. See figure 6.2 for an illustration of this model. This feedforward network has a vector of hidden units $\boldsymbol{h}$ that are computed by a function $f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$. The values of these hidden units are then used as the input for a second layer. The second layer is the output layer of the network. The output layer is still just a linear regression model, but now it is applied to $\boldsymbol{h}$ rather than to $\boldsymbol{x}$. The network now contains two functions chained together: $\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$ and $y = f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b)$, with the complete model being $f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = f^{(2)}(f^{(1)}(\boldsymbol{x}))$.

What function should $f^{(1)}$ compute? Linear models have served us well so far, and it may be tempting to make $f^{(1)}$ be linear as well. Unfortunately, if $f^{(1)}$ were linear, then the feedforward network as a whole would remain a linear function of its input. Ignoring the intercept terms for the moment, suppose $f^{(1)}(\boldsymbol{x}) = \boldsymbol{W}^\top \boldsymbol{x}$ and $f^{(2)}(\boldsymbol{h}) = \boldsymbol{h}^\top \boldsymbol{w}$. Then $f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{W}^\top \boldsymbol{x}$. We could represent this function as $f(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{w}'$ where $\boldsymbol{w}' = \boldsymbol{W}\boldsymbol{w}$.

Clearly, we must use a nonlinear function to describe the features. Most neural networks do so using an affine transformation controlled by learned parameters, followed by a fixed, nonlinear function called an activation function. We use that strategy here, by defining $\boldsymbol{h} = g(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c})$, where $\boldsymbol{W}$ provides the weights of a linear transformation and $\boldsymbol{c}$ the biases. Previously, to describe a linear regression

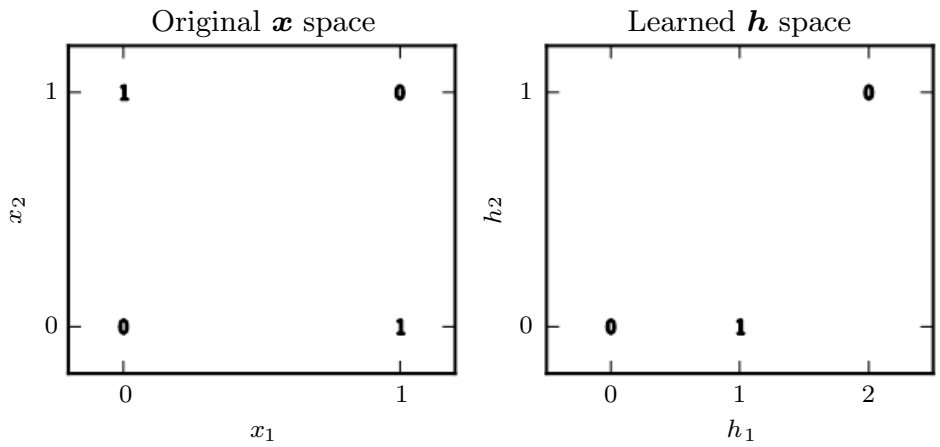

Figure 6.1: Solving the XOR problem by learning a representation. The bold numbers printed on the plot indicate the value that the learned function must output at each point. *(Left)*A linear model applied directly to the original input cannot implement the XOR function. When $x_1 = 0$, the model's output must increase as x_2 increases. When $x_1 = 1$, the model's output must decrease as x_2 increases. A linear model must apply a fixed coefficient w_2 to x_2. The linear model therefore cannot use the value of x_1 to change the coefficient on x_2 and cannot solve this problem. *(Right)*In the transformed space represented by the features extracted by a neural network, a linear model can now solve the problem. In our example solution, the two points that must have output 1 have been collapsed into a single point in feature space. In other words, the nonlinear features have mapped both $\boldsymbol{x} = [1, 0]^\top$ and $\boldsymbol{x} = [0, 1]^\top$ to a single point in feature space, $\boldsymbol{h} = [1\ ,0]^\top$. The linear model can now describe the function as increasing in h_1 and decreasing in h_2. In this example, the motivation for learning the feature space is only to make the model capacity greater so that it can fit the training set. In more realistic applications, learned representations can also help the model to generalize.

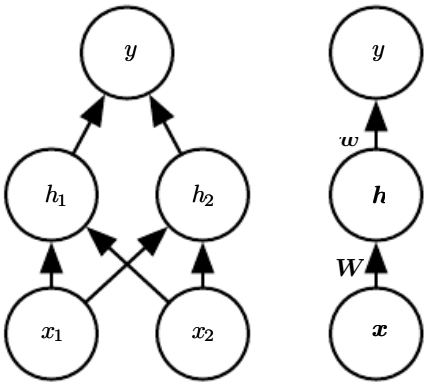

Figure 6.2: An example of a feedforward network, drawn in two different styles. Specifically, this is the feedforward network we use to solve the XOR example. It has a single hidden layer containing two units. *(Left)*In this style, we draw every unit as a node in the graph. This style is very explicit and unambiguous but for networks larger than this example it can consume too much space. *(Right)*In this style, we draw a node in the graph for each entire vector representing a layer's activations. This style is much more compact. Sometimes we annotate the edges in this graph with the name of the parameters that describe the relationship between two layers. Here, we indicate that a matrix $\boldsymbol{W}$ describes the mapping from $\boldsymbol{x}$ to $\boldsymbol{h}$, and a vector $\boldsymbol{w}$ describes the mapping from $\boldsymbol{h}$ to y. We typically omit the intercept parameters associated with each layer when labeling this kind of drawing.

model, we used a vector of weights and a scalar bias parameter to describe an affine transformation from an input vector to an output scalar. Now, we describe an affine transformation from a vector $\boldsymbol{x}$ to a vector $\boldsymbol{h}$, so an entire vector of bias parameters is needed. The activation function g is typically chosen to be a function that is applied element-wise, with $h_i = g(\boldsymbol{x}^\top \boldsymbol{W}_{:,i} + c_i)$. In modern neural networks, the default recommendation is to use the **rectified linear unit** or ReLU (Jarrett *et al.*, 2009; Nair and Hinton, 2010; Glorot *et al.*, 2011a) defined by the activation function $g(z) = \max\{0, z\}$ depicted in figure 6.3.

We can now specify our complete network as

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^\top \max\{0, \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c}\} + b. \tag{6.3}$$

We can now specify a solution to the XOR problem. Let

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \tag{6.4}$$

$$\boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \tag{6.5}$$

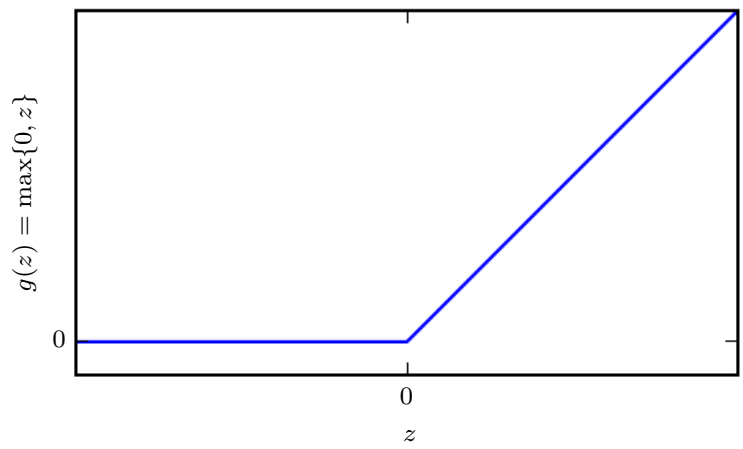

Figure 6.3: The rectified linear activation function. This activation function is the default activation function recommended for use with most feedforward neural networks. Applying this function to the output of a linear transformation yields a nonlinear transformation. However, the function remains very close to linear, in the sense that is a piecewise linear function with two linear pieces. Because rectified linear units are nearly linear, they preserve many of the properties that make linear models easy to optimize with gradient-based methods. They also preserve many of the properties that make linear models generalize well. A common principle throughout computer science is that we can build complicated systems from minimal components. Much as a Turing machine's memory needs only to be able to store 0 or 1 states, we can build a universal function approximator from rectified linear functions.

$$\boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \tag{6.6}$$

and $b = 0$.

We can now walk through the way that the model processes a batch of inputs. Let $\boldsymbol{X}$ be the design matrix containing all four points in the binary input space, with one example per row:

$$\boldsymbol{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}. \tag{6.7}$$

The first step in the neural network is to multiply the input matrix by the first layer's weight matrix:

$$\boldsymbol{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}. \tag{6.8}$$

Next, we add the bias vector $\boldsymbol{c}$, to obtain

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}. \tag{6.9}$$

In this space, all of the examples lie along a line with slope 1. As we move along this line, the output needs to begin at 0, then rise to 1, then drop back down to 0. A linear model cannot implement such a function. To finish computing the value of $\boldsymbol{h}$ for each example, we apply the rectified linear transformation:

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}. \tag{6.10}$$

This transformation has changed the relationship between the examples. They no longer lie on a single line. As shown in figure 6.1, they now lie in a space where a linear model can solve the problem.

We finish by multiplying by the weight vector $\boldsymbol{w}$:

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \tag{6.11}$$

The neural network has obtained the correct answer for every example in the batch.

In this example, we simply specified the solution, then showed that it obtained zero error. In a real situation, there might be billions of model parameters and billions of training examples, so one cannot simply guess the solution as we did here. Instead, a gradient-based optimization algorithm can find parameters that produce very little error. The solution we described to the XOR problem is at a global minimum of the loss function, so gradient descent could converge to this point. There are other equivalent solutions to the XOR problem that gradient descent could also find. The convergence point of gradient descent depends on the initial values of the parameters. In practice, gradient descent would usually not find clean, easily understood, integer-valued solutions like the one we presented here.

6.2 Gradient-Based Learning

Designing and training a neural network is not much different from training any other machine learning model with gradient descent. In section 5.10, we described how to build a machine learning algorithm by specifying an optimization procedure, a cost function, and a model family.

The largest difference between the linear models we have seen so far and neural networks is that the nonlinearity of a neural network causes most interesting loss functions to become non-convex. This means that neural networks are usually trained by using iterative, gradient-based optimizers that merely drive the cost function to a very low value, rather than the linear equation solvers used to train linear regression models or the convex optimization algorithms with global convergence guarantees used to train logistic regression or SVMs. Convex optimization converges starting from any initial parameters (in theory—in practice it is very robust but can encounter numerical problems). Stochastic gradient descent applied to non-convex loss functions has no such convergence guarantee, and is sensitive to the values of the initial parameters. For feedforward neural networks, it is important to initialize all weights to small random values. The biases may be initialized to zero or to small positive values. The iterative gradient-based optimization algorithms used to train feedforward networks and almost all other deep models will be described in detail in chapter 8, with parameter initialization in particular discussed in section 8.4. For the moment, it suffices to understand that the training algorithm is almost always based on using the gradient to descend the cost function in one way or another. The specific algorithms are improvements and refinements on the ideas of gradient descent, introduced in section 4.3, and,

more specifically, are most often improvements of the stochastic gradient descent algorithm, introduced in section 5.9.

We can of course, train models such as linear regression and support vector machines with gradient descent too, and in fact this is common when the training set is extremely large. From this point of view, training a neural network is not much different from training any other model. Computing the gradient is slightly more complicated for a neural network, but can still be done efficiently and exactly. Section 6.5 will describe how to obtain the gradient using the back-propagation algorithm and modern generalizations of the back-propagation algorithm.

As with other machine learning models, to apply gradient-based learning we must choose a cost function, and we must choose how to represent the output of the model. We now revisit these design considerations with special emphasis on the neural networks scenario.

6.2.1 Cost Functions

An important aspect of the design of a deep neural network is the choice of the cost function. Fortunately, the cost functions for neural networks are more or less the same as those for other parametric models, such as linear models.

In most cases, our parametric model defines a distribution $p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$ and we simply use the principle of maximum likelihood. This means we use the cross-entropy between the training data and the model's predictions as the cost function.

Sometimes, we take a simpler approach, where rather than predicting a complete probability distribution over $\boldsymbol{y}$, we merely predict some statistic of $\boldsymbol{y}$ conditioned on $\boldsymbol{x}$. Specialized loss functions allow us to train a predictor of these estimates.

The total cost function used to train a neural network will often combine one of the primary cost functions described here with a regularization term. We have already seen some simple examples of regularization applied to linear models in section 5.2.2. The weight decay approach used for linear models is also directly applicable to deep neural networks and is among the most popular regularization strategies. More advanced regularization strategies for neural networks will be described in chapter 7.

6.2.1.1 Learning Conditional Distributions with Maximum Likelihood

Most modern neural networks are trained using maximum likelihood. This means that the cost function is simply the negative log-likelihood, equivalently described

as the cross-entropy between the training data and the model distribution. This cost function is given by

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x}). \tag{6.12}$$

The specific form of the cost function changes from model to model, depending on the specific form of $\log p_{\text{model}}$. The expansion of the above equation typically yields some terms that do not depend on the model parameters and may be discarded. For example, as we saw in section 5.5.1, if $p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; f(\boldsymbol{x}; \boldsymbol{\theta}), \boldsymbol{I})$, then we recover the mean squared error cost,

$$J(\theta) = \frac{1}{2}\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} ||\boldsymbol{y} - f(\boldsymbol{x}; \boldsymbol{\theta})||^2 + \text{const}, \tag{6.13}$$

up to a scaling factor of $\frac{1}{2}$ and a term that does not depend on $\boldsymbol{\theta}$. The discarded constant is based on the variance of the Gaussian distribution, which in this case we chose not to parametrize. Previously, we saw that the equivalence between maximum likelihood estimation with an output distribution and minimization of mean squared error holds for a linear model, but in fact, the equivalence holds regardless of the $f(\boldsymbol{x}; \boldsymbol{\theta})$ used to predict the mean of the Gaussian.

An advantage of this approach of deriving the cost function from maximum likelihood is that it removes the burden of designing cost functions for each model. Specifying a model $p(\boldsymbol{y} \mid \boldsymbol{x})$ automatically determines a cost function $\log p(\boldsymbol{y} \mid \boldsymbol{x})$.

One recurring theme throughout neural network design is that the gradient of the cost function must be large and predictable enough to serve as a good guide for the learning algorithm. Functions that saturate (become very flat) undermine this objective because they make the gradient become very small. In many cases this happens because the activation functions used to produce the output of the hidden units or the output units saturate. The negative log-likelihood helps to avoid this problem for many models. Many output units involve an exp function that can saturate when its argument is very negative. The log function in the negative log-likelihood cost function undoes the exp of some output units. We will discuss the interaction between the cost function and the choice of output unit in section 6.2.2.

One unusual property of the cross-entropy cost used to perform maximum likelihood estimation is that it usually does not have a minimum value when applied to the models commonly used in practice. For discrete output variables, most models are parametrized in such a way that they cannot represent a probability of zero or one, but can come arbitrarily close to doing so. Logistic regression is an example of such a model. For real-valued output variables, if the model

can control the density of the output distribution (for example, by learning the variance parameter of a Gaussian output distribution) then it becomes possible to assign extremely high density to the correct training set outputs, resulting in cross-entropy approaching negative infinity. Regularization techniques described in chapter 7 provide several different ways of modifying the learning problem so that the model cannot reap unlimited reward in this way.

6.2.1.2 Learning Conditional Statistics

Instead of learning a full probability distribution $p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$ we often want to learn just one conditional statistic of $\boldsymbol{y}$ given $\boldsymbol{x}$.

For example, we may have a predictor $f(\boldsymbol{x}; \boldsymbol{\theta})$ that we wish to predict the mean of $\boldsymbol{y}$.

If we use a sufficiently powerful neural network, we can think of the neural network as being able to represent any function f from a wide class of functions, with this class being limited only by features such as continuity and boundedness rather than by having a specific parametric form. From this point of view, we can view the cost function as being a **functional** rather than just a function. A functional is a mapping from functions to real numbers. We can thus think of learning as choosing a function rather than merely choosing a set of parameters. We can design our cost functional to have its minimum occur at some specific function we desire. For example, we can design the cost functional to have its minimum lie on the function that maps $\boldsymbol{x}$ to the expected value of $\boldsymbol{y}$ given $\boldsymbol{x}$. Solving an optimization problem with respect to a function requires a mathematical tool called **calculus of variations**, described in section 19.4.2. It is not necessary to understand calculus of variations to understand the content of this chapter. At the moment, it is only necessary to understand that calculus of variations may be used to derive the following two results.

Our first result derived using calculus of variations is that solving the optimization problem

$$f^* = \arg\min_f \mathbb{E}_{\mathbf{x},\mathbf{y} \sim p_{\text{data}}} ||\boldsymbol{y} - f(\boldsymbol{x})||^2 \tag{6.14}$$

yields

$$f^*(\boldsymbol{x}) = \mathbb{E}_{\mathbf{y} \sim p_{\text{data}}(\boldsymbol{y}|\boldsymbol{x})}[\boldsymbol{y}], \tag{6.15}$$

so long as this function lies within the class we optimize over. In other words, if we could train on infinitely many samples from the true data generating distribution, minimizing the mean squared error cost function gives a function that predicts the mean of $\boldsymbol{y}$ for each value of $\boldsymbol{x}$.

Different cost functions give different statistics. A second result derived using calculus of variations is that

$$f^* = \arg\min_f \mathbb{E}_{\mathbf{x},\mathbf{y}\sim p_{\text{data}}} ||\boldsymbol{y} - f(\boldsymbol{x})||_1 \tag{6.16}$$

yields a function that predicts the *median* value of $\boldsymbol{y}$ for each $\boldsymbol{x}$, so long as such a function may be described by the family of functions we optimize over. This cost function is commonly called **mean absolute error**.

Unfortunately, mean squared error and mean absolute error often lead to poor results when used with gradient-based optimization. Some output units that saturate produce very small gradients when combined with these cost functions. This is one reason that the cross-entropy cost function is more popular than mean squared error or mean absolute error, even when it is not necessary to estimate an entire distribution $p(\boldsymbol{y} \mid \boldsymbol{x})$.

6.2.2 Output Units

The choice of cost function is tightly coupled with the choice of output unit. Most of the time, we simply use the cross-entropy between the data distribution and the model distribution. The choice of how to represent the output then determines the form of the cross-entropy function.

Any kind of neural network unit that may be used as an output can also be used as a hidden unit. Here, we focus on the use of these units as outputs of the model, but in principle they can be used internally as well. We revisit these units with additional detail about their use as hidden units in section 6.3.

Throughout this section, we suppose that the feedforward network provides a set of hidden features defined by $\boldsymbol{h} = f(\boldsymbol{x}; \boldsymbol{\theta})$. The role of the output layer is then to provide some additional transformation from the features to complete the task that the network must perform.

6.2.2.1 Linear Units for Gaussian Output Distributions

One simple kind of output unit is an output unit based on an affine transformation with no nonlinearity. These are often just called linear units.

Given features $\boldsymbol{h}$, a layer of linear output units produces a vector $\hat{\boldsymbol{y}} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$.

Linear output layers are often used to produce the mean of a conditional Gaussian distribution:

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; \hat{\boldsymbol{y}}, \boldsymbol{I}). \tag{6.17}$$

Maximizing the log-likelihood is then equivalent to minimizing the mean squared error.

The maximum likelihood framework makes it straightforward to learn the covariance of the Gaussian too, or to make the covariance of the Gaussian be a function of the input. However, the covariance must be constrained to be a positive definite matrix for all inputs. It is difficult to satisfy such constraints with a linear output layer, so typically other output units are used to parametrize the covariance. Approaches to modeling the covariance are described shortly, in section 6.2.2.4.

Because linear units do not saturate, they pose little difficulty for gradient-based optimization algorithms and may be used with a wide variety of optimization algorithms.

6.2.2.2 Sigmoid Units for Bernoulli Output Distributions

Many tasks require predicting the value of a binary variable y. Classification problems with two classes can be cast in this form.

The maximum-likelihood approach is to define a Bernoulli distribution over y conditioned on $\boldsymbol{x}$.

A Bernoulli distribution is defined by just a single number. The neural net needs to predict only $P(y = 1 \mid \boldsymbol{x})$. For this number to be a valid probability, it must lie in the interval $[0, 1]$.

Satisfying this constraint requires some careful design effort. Suppose we were to use a linear unit, and threshold its value to obtain a valid probability:

$$P(y = 1 \mid \boldsymbol{x}) = \max \left\{ 0, \min \left\{ 1, \boldsymbol{w}^\top \boldsymbol{h} + b \right\} \right\} . \tag{6.18}$$

This would indeed define a valid conditional distribution, but we would not be able to train it very effectively with gradient descent. Any time that $\boldsymbol{w}^\top \boldsymbol{h} + b$ strayed outside the unit interval, the gradient of the output of the model with respect to its parameters would be $\mathbf{0}$. A gradient of $\mathbf{0}$ is typically problematic because the learning algorithm no longer has a guide for how to improve the corresponding parameters.

Instead, it is better to use a different approach that ensures there is always a strong gradient whenever the model has the wrong answer. This approach is based on using sigmoid output units combined with maximum likelihood.

A sigmoid output unit is defined by

$$\hat{y} = \sigma \left(\boldsymbol{w}^\top \boldsymbol{h} + b \right) \tag{6.19}$$

where σ is the logistic sigmoid function described in section 3.10.

We can think of the sigmoid output unit as having two components. First, it uses a linear layer to compute $z = \boldsymbol{w}^\top \boldsymbol{h} + b$. Next, it uses the sigmoid activation function to convert z into a probability.

We omit the dependence on $\boldsymbol{x}$ for the moment to discuss how to define a probability distribution over y using the value z. The sigmoid can be motivated by constructing an unnormalized probability distribution $\tilde{P}(y)$, which does not sum to 1. We can then divide by an appropriate constant to obtain a valid probability distribution. If we begin with the assumption that the unnormalized log probabilities are linear in y and z, we can exponentiate to obtain the unnormalized probabilities. We then normalize to see that this yields a Bernoulli distribution controlled by a sigmoidal transformation of z:

$$\log \tilde{P}(y) = yz \tag{6.20}$$

$$\tilde{P}(y) = \exp(yz) \tag{6.21}$$

$$P(y) = \frac{\exp(yz)}{\sum_{y'=0}^{1} \exp(y' z)} \tag{6.22}$$

$$P(y) = \sigma\left((2y-1)z\right). \tag{6.23}$$

Probability distributions based on exponentiation and normalization are common throughout the statistical modeling literature. The z variable defining such a distribution over binary variables is called a **logit**.

This approach to predicting the probabilities in log-space is natural to use with maximum likelihood learning. Because the cost function used with maximum likelihood is $-\log P(y \mid \boldsymbol{x})$, the log in the cost function undoes the exp of the sigmoid. Without this effect, the saturation of the sigmoid could prevent gradient-based learning from making good progress. The loss function for maximum likelihood learning of a Bernoulli parametrized by a sigmoid is

$$J(\boldsymbol{\theta}) = -\log P(y \mid \boldsymbol{x}) \tag{6.24}$$

$$= -\log \sigma\left((2y-1)z\right) \tag{6.25}$$

$$= \zeta\left((1-2y)z\right). \tag{6.26}$$

This derivation makes use of some properties from section 3.10. By rewriting the loss in terms of the softplus function, we can see that it saturates only when $(1-2y)z$ is very negative. Saturation thus occurs only when the model already has the right answer—when $y = 1$ and z is very positive, or $y = 0$ and z is very negative. When z has the wrong sign, the argument to the softplus function,

$(1-2y)z$, may be simplified to $|z|$. As $|z|$ becomes large while z has the wrong sign, the softplus function asymptotes toward simply returning its argument $|z|$. The derivative with respect to z asymptotes to $\text{sign}(z)$, so, in the limit of extremely incorrect z, the softplus function does not shrink the gradient at all. This property is very useful because it means that gradient-based learning can act to quickly correct a mistaken z.

When we use other loss functions, such as mean squared error, the loss can saturate anytime $\sigma(z)$ saturates. The sigmoid activation function saturates to 0 when z becomes very negative and saturates to 1 when z becomes very positive. The gradient can shrink too small to be useful for learning whenever this happens, whether the model has the correct answer or the incorrect answer. For this reason, maximum likelihood is almost always the preferred approach to training sigmoid output units.

Analytically, the logarithm of the sigmoid is always defined and finite, because the sigmoid returns values restricted to the open interval $(0, 1)$, rather than using the entire closed interval of valid probabilities $[0, 1]$. In software implementations, to avoid numerical problems, it is best to write the negative log-likelihood as a function of z, rather than as a function of $\hat{y} = \sigma(z)$. If the sigmoid function underflows to zero, then taking the logarithm of $\hat{y}$ yields negative infinity.

6.2.2.3 Softmax Units for Multinoulli Output Distributions

Any time we wish to represent a probability distribution over a discrete variable with n possible values, we may use the softmax function. This can be seen as a generalization of the sigmoid function which was used to represent a probability distribution over a binary variable.

Softmax functions are most often used as the output of a classifier, to represent the probability distribution over n different classes. More rarely, softmax functions can be used inside the model itself, if we wish the model to choose between one of n different options for some internal variable.

In the case of binary variables, we wished to produce a single number

$$\hat{y} = P(y = 1 \mid \boldsymbol{x}). \tag{6.27}$$

Because this number needed to lie between 0 and 1, and because we wanted the logarithm of the number to be well-behaved for gradient-based optimization of the log-likelihood, we chose to instead predict a number $z = \log \tilde{P}(y = 1 \mid \boldsymbol{x})$. Exponentiating and normalizing gave us a Bernoulli distribution controlled by the sigmoid function.

To generalize to the case of a discrete variable with n values, we now need to produce a vector $\hat{\boldsymbol{y}}$, with $\hat{y}_i = P(y = i \mid \boldsymbol{x})$. We require not only that each element of $\hat{y}_i$ be between 0 and 1, but also that the entire vector sums to 1 so that it represents a valid probability distribution. The same approach that worked for the Bernoulli distribution generalizes to the multinoulli distribution. First, a linear layer predicts unnormalized log probabilities:

$$\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}, \tag{6.28}$$

where $z_i = \log \tilde{P}(y = i \mid \boldsymbol{x})$. The softmax function can then exponentiate and normalize $\boldsymbol{z}$ to obtain the desired $\hat{\boldsymbol{y}}$. Formally, the softmax function is given by

$$\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}. \tag{6.29}$$

As with the logistic sigmoid, the use of the exp function works very well when training the softmax to output a target value y using maximum log-likelihood. In this case, we wish to maximize $\log P(\mathrm{y} = i; \boldsymbol{z}) = \log \text{softmax}(\boldsymbol{z})_i$. Defining the softmax in terms of exp is natural because the log in the log-likelihood can undo the exp of the softmax:

$$\log \text{softmax}(\boldsymbol{z})_i = z_i - \log \sum_j \exp(z_j). \tag{6.30}$$

The first term of equation 6.30 shows that the input z_i always has a direct contribution to the cost function. Because this term cannot saturate, we know that learning can proceed, even if the contribution of z_i to the second term of equation 6.30 becomes very small. When maximizing the log-likelihood, the first term encourages z_i to be pushed up, while the second term encourages all of $\boldsymbol{z}$ to be pushed down. To gain some intuition for the second term, $\log \sum_j \exp(z_j)$, observe that this term can be roughly approximated by $\max_j z_j$. This approximation is based on the idea that $\exp(z_k)$ is insignificant for any z_k that is noticeably less than $\max_j z_j$. The intuition we can gain from this approximation is that the negative log-likelihood cost function always strongly penalizes the most active incorrect prediction. If the correct answer already has the largest input to the softmax, then the $-z_i$ term and the $\log \sum_j \exp(z_j) \approx \max_j z_j = z_i$ terms will roughly cancel. This example will then contribute little to the overall training cost, which will be dominated by other examples that are not yet correctly classified.

So far we have discussed only a single example. Overall, unregularized maximum likelihood will drive the model to learn parameters that drive the softmax to predict

the fraction of counts of each outcome observed in the training set:

$$\text{softmax}(\boldsymbol{z}(\boldsymbol{x};\boldsymbol{\theta}))_i \approx \frac{\sum_{j=1}^{m} \mathbf{1}_{y^{(j)}=i,\boldsymbol{x}^{(j)}=\boldsymbol{x}}}{\sum_{j=1}^{m} \mathbf{1}_{\boldsymbol{x}^{(j)}=\boldsymbol{x}}}. \tag{6.31}$$

Because maximum likelihood is a consistent estimator, this is guaranteed to happen so long as the model family is capable of representing the training distribution. In practice, limited model capacity and imperfect optimization will mean that the model is only able to approximate these fractions.

Many objective functions other than the log-likelihood do not work as well with the softmax function. Specifically, objective functions that do not use a log to undo the exp of the softmax fail to learn when the argument to the exp becomes very negative, causing the gradient to vanish. In particular, squared error is a poor loss function for softmax units, and can fail to train the model to change its output, even when the model makes highly confident incorrect predictions (Bridle, 1990). To understand why these other loss functions can fail, we need to examine the softmax function itself.

Like the sigmoid, the softmax activation can saturate. The sigmoid function has a single output that saturates when its input is extremely negative or extremely positive. In the case of the softmax, there are multiple output values. These output values can saturate when the differences between input values become extreme. When the softmax saturates, many cost functions based on the softmax also saturate, unless they are able to invert the saturating activating function.

To see that the softmax function responds to the difference between its inputs, observe that the softmax output is invariant to adding the same scalar to all of its inputs:

$$\text{softmax}(\boldsymbol{z}) = \text{softmax}(\boldsymbol{z} + c). \tag{6.32}$$

Using this property, we can derive a numerically stable variant of the softmax:

$$\text{softmax}(\boldsymbol{z}) = \text{softmax}(\boldsymbol{z} - \max_i z_i). \tag{6.33}$$

The reformulated version allows us to evaluate softmax with only small numerical errors even when z contains extremely large or extremely negative numbers. Examining the numerically stable variant, we see that the softmax function is driven by the amount that its arguments deviate from $\max_i z_i$.

An output $\text{softmax}(\boldsymbol{z})_i$ saturates to 1 when the corresponding input is maximal ($z_i = \max_i z_i$) and z_i is much greater than all of the other inputs. The output $\text{softmax}(\boldsymbol{z})_i$ can also saturate to 0 when z_i is not maximal and the maximum is much greater. This is a generalization of the way that sigmoid units saturate, and

can cause similar difficulties for learning if the loss function is not designed to compensate for it.

The argument $\boldsymbol{z}$ to the softmax function can be produced in two different ways. The most common is simply to have an earlier layer of the neural network output every element of $\boldsymbol{z}$, as described above using the linear layer $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$. While straightforward, this approach actually overparametrizes the distribution. The constraint that the n outputs must sum to 1 means that only $n-1$ parameters are necessary; the probability of the n-th value may be obtained by subtracting the first $n-1$ probabilities from 1. We can thus impose a requirement that one element of $\boldsymbol{z}$ be fixed. For example, we can require that $z_n = 0$. Indeed, this is exactly what the sigmoid unit does. Defining $P(y = 1 \mid \boldsymbol{x}) = \sigma(z)$ is equivalent to defining $P(y = 1 \mid \boldsymbol{x}) = \text{softmax}(\boldsymbol{z})_1$ with a two-dimensional $\boldsymbol{z}$ and $z_1 = 0$. Both the $n-1$ argument and the n argument approaches to the softmax can describe the same set of probability distributions, but have different learning dynamics. In practice, there is rarely much difference between using the overparametrized version or the restricted version, and it is simpler to implement the overparametrized version.

From a neuroscientific point of view, it is interesting to think of the softmax as a way to create a form of competition between the units that participate in it: the softmax outputs always sum to 1 so an increase in the value of one unit necessarily corresponds to a decrease in the value of others. This is analogous to the lateral inhibition that is believed to exist between nearby neurons in the cortex. At the extreme (when the difference between the maximal a_i and the others is large in magnitude) it becomes a form of **winner-take-all** (one of the outputs is nearly 1 and the others are nearly 0).

The name "softmax" can be somewhat confusing. The function is more closely related to the arg max function than the max function. The term "soft" derives from the fact that the softmax function is continuous and differentiable. The arg max function, with its result represented as a one-hot vector, is not continuous or differentiable. The softmax function thus provides a "softened" version of the arg max. The corresponding soft version of the maximum function is $\text{softmax}(\boldsymbol{z})^\top \boldsymbol{z}$. It would perhaps be better to call the softmax function "softargmax," but the current name is an entrenched convention.

6.2.2.4 Other Output Types

The linear, sigmoid, and softmax output units described above are the most common. Neural networks can generalize to almost any kind of output layer that we wish. The principle of maximum likelihood provides a guide for how to design

a good cost function for nearly any kind of output layer.

In general, if we define a conditional distribution $p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$, the principle of maximum likelihood suggests we use $-\log p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$ as our cost function.

In general, we can think of the neural network as representing a function $f(\boldsymbol{x}; \boldsymbol{\theta})$. The outputs of this function are not direct predictions of the value $\boldsymbol{y}$. Instead, $f(\boldsymbol{x}; \boldsymbol{\theta}) = \boldsymbol{\omega}$ provides the parameters for a distribution over y. Our loss function can then be interpreted as $-\log p(\mathbf{y}; \boldsymbol{\omega}(\boldsymbol{x}))$.

For example, we may wish to learn the variance of a conditional Gaussian for $\mathbf{y}$, given $\mathbf{x}$. In the simple case, where the variance σ^2 is a constant, there is a closed form expression because the maximum likelihood estimator of variance is simply the empirical mean of the squared difference between observations $\mathbf{y}$ and their expected value. A computationally more expensive approach that does not require writing special-case code is to simply include the variance as one of the properties of the distribution $p(\mathbf{y} \mid \boldsymbol{x})$ that is controlled by $\boldsymbol{\omega} = f(\boldsymbol{x}; \boldsymbol{\theta})$. The negative log-likelihood $-\log p(\boldsymbol{y}; \boldsymbol{\omega}(\boldsymbol{x}))$ will then provide a cost function with the appropriate terms necessary to make our optimization procedure incrementally learn the variance. In the simple case where the standard deviation does not depend on the input, we can make a new parameter in the network that is copied directly into $\boldsymbol{\omega}$. This new parameter might be σ itself or could be a parameter v representing σ^2 or it could be a parameter β representing $\frac{1}{\sigma^2}$, depending on how we choose to parametrize the distribution. We may wish our model to predict a different amount of variance in $\mathbf{y}$ for different values of $\mathbf{x}$. This is called a **heteroscedastic** model. In the heteroscedastic case, we simply make the specification of the variance be one of the values output by $f(\mathbf{x}; \boldsymbol{\theta})$. A typical way to do this is to formulate the Gaussian distribution using precision, rather than variance, as described in equation 3.22. In the multivariate case it is most common to use a diagonal precision matrix

$$\text{diag}(\boldsymbol{\beta}). \tag{6.34}$$

This formulation works well with gradient descent because the formula for the log-likelihood of the Gaussian distribution parametrized by $\boldsymbol{\beta}$ involves only multiplication by β_i and addition of $\log \boldsymbol{\beta}_i$. The gradient of multiplication, addition, and logarithm operations is well-behaved. By comparison, if we parametrized the output in terms of variance, we would need to use division. The division function becomes arbitrarily steep near zero. While large gradients can help learning, arbitrarily large gradients usually result in instability. If we parametrized the output in terms of standard deviation, the log-likelihood would still involve division, and would also involve squaring. The gradient through the squaring operation can vanish near zero, making it difficult to learn parameters that are squared.

Regardless of whether we use standard deviation, variance, or precision, we must ensure that the covariance matrix of the Gaussian is positive definite. Because the eigenvalues of the precision matrix are the reciprocals of the eigenvalues of the covariance matrix, this is equivalent to ensuring that the precision matrix is positive definite. If we use a diagonal matrix, or a scalar times the diagonal matrix, then the only condition we need to enforce on the output of the model is positivity. If we suppose that $\boldsymbol{a}$ is the raw activation of the model used to determine the diagonal precision, we can use the softplus function to obtain a positive precision vector: $\boldsymbol{\beta} = \zeta(\boldsymbol{a})$. This same strategy applies equally if using variance or standard deviation rather than precision or if using a scalar times identity rather than diagonal matrix.

It is rare to learn a covariance or precision matrix with richer structure than diagonal. If the covariance is full and conditional, then a parametrization must be chosen that guarantees positive-definiteness of the predicted covariance matrix. This can be achieved by writing $\boldsymbol{\Sigma}(\boldsymbol{x}) = \boldsymbol{B}(\boldsymbol{x})\boldsymbol{B}^\top(\boldsymbol{x})$, where $\boldsymbol{B}$ is an unconstrained square matrix. One practical issue if the matrix is full rank is that computing the likelihood is expensive, with a $d \times d$ matrix requiring $O(d^3)$ computation for the determinant and inverse of $\boldsymbol{\Sigma}(\boldsymbol{x})$ (or equivalently, and more commonly done, its eigendecomposition or that of $\boldsymbol{B}(\boldsymbol{x})$).

We often want to perform multimodal regression, that is, to predict real values that come from a conditional distribution $p(\boldsymbol{y} \mid \boldsymbol{x})$ that can have several different peaks in $\boldsymbol{y}$ space for the same value of $\boldsymbol{x}$. In this case, a Gaussian mixture is a natural representation for the output (Jacobs *et al.*, 1991; Bishop, 1994). Neural networks with Gaussian mixtures as their output are often called **mixture density networks**. A Gaussian mixture output with n components is defined by the conditional probability distribution

$$p(\boldsymbol{y} \mid \boldsymbol{x}) = \sum_{i=1}^{n} p(\mathrm{c} = i \mid \boldsymbol{x}) \mathcal{N}(\boldsymbol{y}; \boldsymbol{\mu}^{(i)}(\boldsymbol{x}), \boldsymbol{\Sigma}^{(i)}(\boldsymbol{x})). \tag{6.35}$$

The neural network must have three outputs: a vector defining $p(\mathrm{c} = i \mid \boldsymbol{x})$, a matrix providing $\boldsymbol{\mu}^{(i)}(\boldsymbol{x})$ for all i, and a tensor providing $\boldsymbol{\Sigma}^{(i)}(\boldsymbol{x})$ for all i. These outputs must satisfy different constraints:

1. Mixture components $p(\mathrm{c} = i \mid \boldsymbol{x})$: these form a multinoulli distribution over the n different components associated with latent variable[1] c, and can

[1] We consider c to be latent because we do not observe it in the data: given input $\mathbf{x}$ and target $\mathbf{y}$, it is not possible to know with certainty which Gaussian component was responsible for $\mathbf{y}$, but we can imagine that $\mathbf{y}$ was generated by picking one of them, and make that unobserved choice a random variable.

typically be obtained by a softmax over an n-dimensional vector, to guarantee that these outputs are positive and sum to 1.

2. Means $\boldsymbol{\mu}^{(i)}(\boldsymbol{x})$: these indicate the center or mean associated with the i-th Gaussian component, and are unconstrained (typically with no nonlinearity at all for these output units). If $\mathbf{y}$ is a d-vector, then the network must output an $n \times d$ matrix containing all n of these d-dimensional vectors. Learning these means with maximum likelihood is slightly more complicated than learning the means of a distribution with only one output mode. We only want to update the mean for the component that actually produced the observation. In practice, we do not know which component produced each observation. The expression for the negative log-likelihood naturally weights each example's contribution to the loss for each component by the probability that the component produced the example.

3. Covariances $\boldsymbol{\Sigma}^{(i)}(\boldsymbol{x})$: these specify the covariance matrix for each component i. As when learning a single Gaussian component, we typically use a diagonal matrix to avoid needing to compute determinants. As with learning the means of the mixture, maximum likelihood is complicated by needing to assign partial responsibility for each point to each mixture component. Gradient descent will automatically follow the correct process if given the correct specification of the negative log-likelihood under the mixture model.

It has been reported that gradient-based optimization of conditional Gaussian mixtures (on the output of neural networks) can be unreliable, in part because one gets divisions (by the variance) which can be numerically unstable (when some variance gets to be small for a particular example, yielding very large gradients). One solution is to **clip gradients** (see section 10.11.1) while another is to scale the gradients heuristically (Murray and Larochelle, 2014).

Gaussian mixture outputs are particularly effective in generative models of speech (Schuster, 1999) or movements of physical objects (Graves, 2013). The mixture density strategy gives a way for the network to represent multiple output modes and to control the variance of its output, which is crucial for obtaining a high degree of quality in these real-valued domains. An example of a mixture density network is shown in figure 6.4.

In general, we may wish to continue to model larger vectors $\boldsymbol{y}$ containing more variables, and to impose richer and richer structures on these output variables. For example, we may wish for our neural network to output a sequence of characters that forms a sentence. In these cases, we may continue to use the principle of maximum likelihood applied to our model $p(\boldsymbol{y}; \boldsymbol{\omega}(\boldsymbol{x}))$, but the model we use

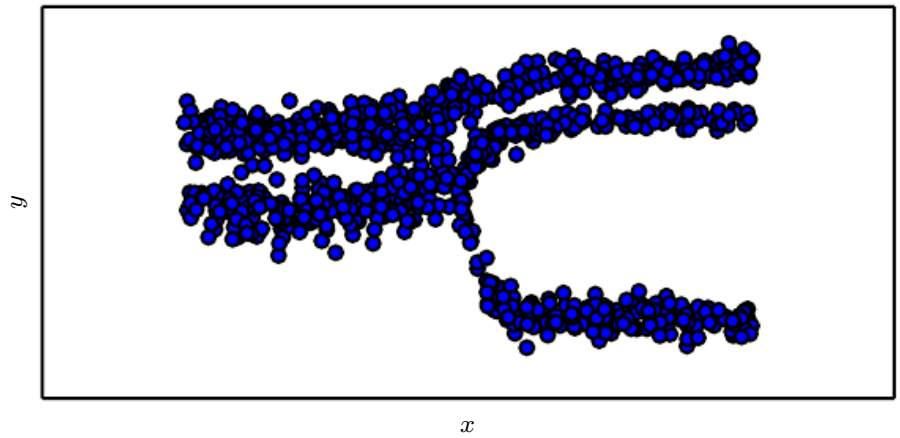

Figure 6.4: Samples drawn from a neural network with a mixture density output layer. The input x is sampled from a uniform distribution and the output y is sampled from $p_{\text{model}}(y \mid x)$. The neural network is able to learn nonlinear mappings from the input to the parameters of the output distribution. These parameters include the probabilities governing which of three mixture components will generate the output as well as the parameters for each mixture component. Each mixture component is Gaussian with predicted mean and variance. All of these aspects of the output distribution are able to vary with respect to the input x, and to do so in nonlinear ways.

to describe $\boldsymbol{y}$ becomes complex enough to be beyond the scope of this chapter. Chapter 10 describes how to use recurrent neural networks to define such models over sequences, and part III describes advanced techniques for modeling arbitrary probability distributions.

6.3 Hidden Units

So far we have focused our discussion on design choices for neural networks that are common to most parametric machine learning models trained with gradient-based optimization. Now we turn to an issue that is unique to feedforward neural networks: how to choose the type of hidden unit to use in the hidden layers of the model.

The design of hidden units is an extremely active area of research and does not yet have many definitive guiding theoretical principles.

Rectified linear units are an excellent default choice of hidden unit. Many other types of hidden units are available. It can be difficult to determine when to use which kind (though rectified linear units are usually an acceptable choice). We

describe here some of the basic intuitions motivating each type of hidden units. These intuitions can help decide when to try out each of these units. It is usually impossible to predict in advance which will work best. The design process consists of trial and error, intuiting that a kind of hidden unit may work well, and then training a network with that kind of hidden unit and evaluating its performance on a validation set.

Some of the hidden units included in this list are not actually differentiable at all input points. For example, the rectified linear function $g(z) = \max\{0, z\}$ is not differentiable at $z = 0$. This may seem like it invalidates g for use with a gradient-based learning algorithm. In practice, gradient descent still performs well enough for these models to be used for machine learning tasks. This is in part because neural network training algorithms do not usually arrive at a local minimum of the cost function, but instead merely reduce its value significantly, as shown in figure 4.3. These ideas will be described further in chapter 8. Because we do not expect training to actually reach a point where the gradient is $\mathbf{0}$, it is acceptable for the minima of the cost function to correspond to points with undefined gradient. Hidden units that are not differentiable are usually non-differentiable at only a small number of points. In general, a function $g(z)$ has a left derivative defined by the slope of the function immediately to the left of z and a right derivative defined by the slope of the function immediately to the right of z. A function is differentiable at z only if both the left derivative and the right derivative are defined and equal to each other. The functions used in the context of neural networks usually have defined left derivatives and defined right derivatives. In the case of $g(z) = \max\{0, z\}$, the left derivative at $z = 0$ is 0 and the right derivative is 1. Software implementations of neural network training usually return one of the one-sided derivatives rather than reporting that the derivative is undefined or raising an error. This may be heuristically justified by observing that gradient-based optimization on a digital computer is subject to numerical error anyway. When a function is asked to evaluate $g(0)$, it is very unlikely that the underlying value truly was 0. Instead, it was likely to be some small value ϵ that was rounded to 0. In some contexts, more theoretically pleasing justifications are available, but these usually do not apply to neural network training. The important point is that in practice one can safely disregard the non-differentiability of the hidden unit activation functions described below.

Unless indicated otherwise, most hidden units can be described as accepting a vector of inputs $\boldsymbol{x}$, computing an affine transformation $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}$, and then applying an element-wise nonlinear function $g(\boldsymbol{z})$. Most hidden units are distinguished from each other only by the choice of the form of the activation function $g(\boldsymbol{z})$.

6.3.1 Rectified Linear Units and Their Generalizations

Rectified linear units use the activation function $g(z) = \max\{0, z\}$.

Rectified linear units are easy to optimize because they are so similar to linear units. The only difference between a linear unit and a rectified linear unit is that a rectified linear unit outputs zero across half its domain. This makes the derivatives through a rectified linear unit remain large whenever the unit is active. The gradients are not only large but also consistent. The second derivative of the rectifying operation is 0 almost everywhere, and the derivative of the rectifying operation is 1 everywhere that the unit is active. This means that the gradient direction is far more useful for learning than it would be with activation functions that introduce second-order effects.

Rectified linear units are typically used on top of an affine transformation:

$$\boldsymbol{h} = g(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}). \tag{6.36}$$

When initializing the parameters of the affine transformation, it can be a good practice to set all elements of $\boldsymbol{b}$ to a small, positive value, such as 0.1. This makes it very likely that the rectified linear units will be initially active for most inputs in the training set and allow the derivatives to pass through.

Several generalizations of rectified linear units exist. Most of these generalizations perform comparably to rectified linear units and occasionally perform better.

One drawback to rectified linear units is that they cannot learn via gradient-based methods on examples for which their activation is zero. A variety of generalizations of rectified linear units guarantee that they receive gradient everywhere.

Three generalizations of rectified linear units are based on using a non-zero slope α_i when $z_i < 0$: $h_i = g(\boldsymbol{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$. **Absolute value rectification** fixes $\alpha_i = -1$ to obtain $g(z) = |z|$. It is used for object recognition from images (Jarrett *et al.*, 2009), where it makes sense to seek features that are invariant under a polarity reversal of the input illumination. Other generalizations of rectified linear units are more broadly applicable. A **leaky ReLU** (Maas *et al.*, 2013) fixes α_i to a small value like 0.01 while a **parametric ReLU** or **PReLU** treats α_i as a learnable parameter (He *et al.*, 2015).

Maxout units (Goodfellow *et al.*, 2013a) generalize rectified linear units further. Instead of applying an element-wise function $g(z)$, maxout units divide $\boldsymbol{z}$ into groups of k values. Each maxout unit then outputs the maximum element of

one of these groups:

$$g(\boldsymbol{z})_i = \max_{j \in \mathbb{G}^{(i)}} z_j \tag{6.37}$$

where $\mathbb{G}^{(i)}$ is the set of indices into the inputs for group i, $\{(i-1)k+1, \ldots, ik\}$. This provides a way of learning a piecewise linear function that responds to multiple directions in the input $\boldsymbol{x}$ space.

A maxout unit can learn a piecewise linear, convex function with up to k pieces. Maxout units can thus be seen as *learning the activation function* itself rather than just the relationship between units. With large enough k, a maxout unit can learn to approximate any convex function with arbitrary fidelity. In particular, a maxout layer with two pieces can learn to implement the same function of the input $\boldsymbol{x}$ as a traditional layer using the rectified linear activation function, absolute value rectification function, or the leaky or parametric ReLU, or can learn to implement a totally different function altogether. The maxout layer will of course be parametrized differently from any of these other layer types, so the learning dynamics will be different even in the cases where maxout learns to implement the same function of $\boldsymbol{x}$ as one of the other layer types.

Each maxout unit is now parametrized by k weight vectors instead of just one, so maxout units typically need more regularization than rectified linear units. They can work well without regularization if the training set is large and the number of pieces per unit is kept low (Cai *et al.*, 2013).

Maxout units have a few other benefits. In some cases, one can gain some statistical and computational advantages by requiring fewer parameters. Specifically, if the features captured by n different linear filters can be summarized without losing information by taking the max over each group of k features, then the next layer can get by with k times fewer weights.

Because each unit is driven by multiple filters, maxout units have some redundancy that helps them to resist a phenomenon called **catastrophic forgetting** in which neural networks forget how to perform tasks that they were trained on in the past (Goodfellow *et al.*, 2014a).

Rectified linear units and all of these generalizations of them are based on the principle that models are easier to optimize if their behavior is closer to linear. This same general principle of using linear behavior to obtain easier optimization also applies in other contexts besides deep linear networks. Recurrent networks can learn from sequences and produce a sequence of states and outputs. When training them, one needs to propagate information through several time steps, which is much easier when some linear computations (with some directional derivatives being of magnitude near 1) are involved. One of the best-performing recurrent network

architectures, the LSTM, propagates information through time via summation—a particular straightforward kind of such linear activation. This is discussed further in section 10.10.

6.3.2 Logistic Sigmoid and Hyperbolic Tangent

Prior to the introduction of rectified linear units, most neural networks used the logistic sigmoid activation function

$$g(z) = \sigma(z) \tag{6.38}$$

or the hyperbolic tangent activation function

$$g(z) = \tanh(z). \tag{6.39}$$

These activation functions are closely related because $\tanh(z) = 2\sigma(2z) - 1$.

We have already seen sigmoid units as output units, used to predict the probability that a binary variable is 1. Unlike piecewise linear units, sigmoidal units saturate across most of their domain—they saturate to a high value when z is very positive, saturate to a low value when z is very negative, and are only strongly sensitive to their input when z is near 0. The widespread saturation of sigmoidal units can make gradient-based learning very difficult. For this reason, their use as hidden units in feedforward networks is now discouraged. Their use as output units is compatible with the use of gradient-based learning when an appropriate cost function can undo the saturation of the sigmoid in the output layer.

When a sigmoidal activation function must be used, the hyperbolic tangent activation function typically performs better than the logistic sigmoid. It resembles the identity function more closely, in the sense that $\tanh(0) = 0$ while $\sigma(0) = \frac{1}{2}$. Because tanh is similar to the identity function near 0, training a deep neural network $\hat{y} = \boldsymbol{w}^\top \tanh(\boldsymbol{U}^\top \tanh(\boldsymbol{V}^\top \boldsymbol{x}))$ resembles training a linear model $\hat{y} = \boldsymbol{w}^\top \boldsymbol{U}^\top \boldsymbol{V}^\top \boldsymbol{x}$ so long as the activations of the network can be kept small. This makes training the tanh network easier.

Sigmoidal activation functions are more common in settings other than feed-forward networks. Recurrent networks, many probabilistic models, and some autoencoders have additional requirements that rule out the use of piecewise linear activation functions and make sigmoidal units more appealing despite the drawbacks of saturation.

6.3.3 Other Hidden Units

Many other types of hidden units are possible, but are used less frequently.

In general, a wide variety of differentiable functions perform perfectly well. Many unpublished activation functions perform just as well as the popular ones. To provide a concrete example, the authors tested a feedforward network using $\boldsymbol{h} = \cos(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$ on the MNIST dataset and obtained an error rate of less than 1%, which is competitive with results obtained using more conventional activation functions. During research and development of new techniques, it is common to test many different activation functions and find that several variations on standard practice perform comparably. This means that usually new hidden unit types are published only if they are clearly demonstrated to provide a significant improvement. New hidden unit types that perform roughly comparably to known types are so common as to be uninteresting.

It would be impractical to list all of the hidden unit types that have appeared in the literature. We highlight a few especially useful and distinctive ones.

One possibility is to not have an activation $g(z)$ at all. One can also think of this as using the identity function as the activation function. We have already seen that a linear unit can be useful as the output of a neural network. It may also be used as a hidden unit. If every layer of the neural network consists of only linear transformations, then the network as a whole will be linear. However, it is acceptable for some layers of the neural network to be purely linear. Consider a neural network layer with n inputs and p outputs, $\boldsymbol{h} = g(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b})$. We may replace this with two layers, with one layer using weight matrix $\boldsymbol{U}$ and the other using weight matrix $\boldsymbol{V}$. If the first layer has no activation function, then we have essentially factored the weight matrix of the original layer based on $\boldsymbol{W}$. The factored approach is to compute $\boldsymbol{h} = g(\boldsymbol{V}^\top \boldsymbol{U}^\top \boldsymbol{x} + \boldsymbol{b})$. If $\boldsymbol{U}$ produces q outputs, then $\boldsymbol{U}$ and $\boldsymbol{V}$ together contain only $(n + p)q$ parameters, while $\boldsymbol{W}$ contains np parameters. For small q, this can be a considerable saving in parameters. It comes at the cost of constraining the linear transformation to be low-rank, but these low-rank relationships are often sufficient. Linear hidden units thus offer an effective way of reducing the number of parameters in a network.

Softmax units are another kind of unit that is usually used as an output (as described in section 6.2.2.3) but may sometimes be used as a hidden unit. Softmax units naturally represent a probability distribution over a discrete variable with k possible values, so they may be used as a kind of switch. These kinds of hidden units are usually only used in more advanced architectures that explicitly learn to manipulate memory, described in section 10.12.

A few other reasonably common hidden unit types include:

- **Radial basis function** or RBF unit: $h_i = \exp\left(-\frac{1}{\sigma_i^2}||\boldsymbol{W}_{:,i} - \boldsymbol{x}||^2\right)$. This function becomes more active as $\boldsymbol{x}$ approaches a template $\boldsymbol{W}_{:,i}$. Because it saturates to 0 for most $\boldsymbol{x}$, it can be difficult to optimize.
- **Softplus**: $g(a) = \zeta(a) = \log(1+e^a)$. This is a smooth version of the rectifier, introduced by Dugas *et al.* (2001) for function approximation and by Nair and Hinton (2010) for the conditional distributions of undirected probabilistic models. Glorot *et al.* (2011a) compared the softplus and rectifier and found better results with the latter. The use of the softplus is generally discouraged. The softplus demonstrates that the performance of hidden unit types can be very counterintuitive—one might expect it to have an advantage over the rectifier due to being differentiable everywhere or due to saturating less completely, but empirically it does not.
- **Hard** tanh: this is shaped similarly to the tanh and the rectifier but unlike the latter, it is bounded, $g(a) = \max(-1, \min(1\,, a))$. It was introduced by Collobert (2004).

Hidden unit design remains an active area of research and many useful hidden unit types remain to be discovered.

6.4 Architecture Design

Another key design consideration for neural networks is determining the architecture. The word **architecture** refers to the overall structure of the network: how many units it should have and how these units should be connected to each other.

Most neural networks are organized into groups of units called layers. Most neural network architectures arrange these layers in a chain structure, with each layer being a function of the layer that preceded it. In this structure, the first layer is given by

$$\boldsymbol{h}^{(1)} = g^{(1)}\left(\boldsymbol{W}^{(1)\top}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right), \tag{6.40}$$

the second layer is given by

$$\boldsymbol{h}^{(2)} = g^{(2)}\left(\boldsymbol{W}^{(2)\top}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}\right), \tag{6.41}$$

and so on.

In these chain-based architectures, the main architectural considerations are to choose the depth of the network and the width of each layer. As we will see, a network with even one hidden layer is sufficient to fit the training set. Deeper networks often are able to use far fewer units per layer and far fewer parameters and often generalize to the test set, but are also often harder to optimize. The ideal network architecture for a task must be found via experimentation guided by monitoring the validation set error.

6.4.1 Universal Approximation Properties and Depth

A linear model, mapping from features to outputs via matrix multiplication, can by definition represent only linear functions. It has the advantage of being easy to train because many loss functions result in convex optimization problems when applied to linear models. Unfortunately, we often want to learn nonlinear functions.

At first glance, we might presume that learning a nonlinear function requires designing a specialized model family for the kind of nonlinearity we want to learn. Fortunately, feedforward networks with hidden layers provide a universal approximation framework. Specifically, the **universal approximation theorem** (Hornik *et al.*, 1989; Cybenko, 1989) states that a feedforward network with a linear output layer and at least one hidden layer with any "squashing" activation function (such as the logistic sigmoid activation function) can approximate any Borel measurable function from one finite-dimensional space to another with any desired non-zero amount of error, provided that the network is given enough hidden units. The derivatives of the feedforward network can also approximate the derivatives of the function arbitrarily well (Hornik *et al.*, 1990). The concept of Borel measurability is beyond the scope of this book; for our purposes it suffices to say that any continuous function on a closed and bounded subset of $\mathbb{R}^n$ is Borel measurable and therefore may be approximated by a neural network. A neural network may also approximate any function mapping from any finite dimensional discrete space to another. While the original theorems were first stated in terms of units with activation functions that saturate both for very negative and for very positive arguments, universal approximation theorems have also been proved for a wider class of activation functions, which includes the now commonly used rectified linear unit (Leshno *et al.*, 1993).

The universal approximation theorem means that regardless of what function we are trying to learn, we know that a large MLP will be able to *represent* this function. However, we are not guaranteed that the training algorithm will be able to *learn* that function. Even if the MLP is able to represent the function, learning can fail for two different reasons. First, the optimization algorithm used for training

may not be able to find the value of the parameters that corresponds to the desired function. Second, the training algorithm might choose the wrong function due to overfitting. Recall from section 5.2.1 that the "no free lunch" theorem shows that there is no universally superior machine learning algorithm. Feedforward networks provide a universal system for representing functions, in the sense that, given a function, there exists a feedforward network that approximates the function. There is no universal procedure for examining a training set of specific examples and choosing a function that will generalize to points not in the training set.

The universal approximation theorem says that there exists a network large enough to achieve any degree of accuracy we desire, but the theorem does not say how large this network will be. Barron (1993) provides some bounds on the size of a single-layer network needed to approximate a broad class of functions. Unfortunately, in the worse case, an exponential number of hidden units (possibly with one hidden unit corresponding to each input configuration that needs to be distinguished) may be required. This is easiest to see in the binary case: the number of possible binary functions on vectors $\boldsymbol{v} \in \{0, 1\}^n$ is 2^{2^n} and selecting one such function requires 2^n bits, which will in general require $O(2^n)$ degrees of freedom.

In summary, a feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasibly large and may fail to learn and generalize correctly. In many circumstances, using deeper models can reduce the number of units required to represent the desired function and can reduce the amount of generalization error.

There exist families of functions which can be approximated efficiently by an architecture with depth greater than some value d, but which require a much larger model if depth is restricted to be less than or equal to d. In many cases, the number of hidden units required by the shallow model is exponential in n. Such results were first proved for models that do not resemble the continuous, differentiable neural networks used for machine learning, but have since been extended to these models. The first results were for circuits of logic gates (Håstad, 1986). Later work extended these results to linear threshold units with non-negative weights (Håstad and Goldmann, 1991; Hajnal *et al.*, 1993), and then to networks with continuous-valued activations (Maass, 1992; Maass *et al.*, 1994). Many modern neural networks use rectified linear units. Leshno *et al.* (1993) demonstrated that shallow networks with a broad family of non-polynomial activation functions, including rectified linear units, have universal approximation properties, but these results do not address the questions of depth or efficiency—they specify only that a sufficiently wide rectifier network could represent any function. Montufar *et al.*

(2014) showed that functions representable with a deep rectifier net can require an exponential number of hidden units with a shallow (one hidden layer) network. More precisely, they showed that piecewise linear networks (which can be obtained from rectifier nonlinearities or maxout units) can represent functions with a number of regions that is exponential in the depth of the network. Figure 6.5 illustrates how a network with absolute value rectification creates mirror images of the function computed on top of some hidden unit, with respect to the input of that hidden unit. Each hidden unit specifies where to fold the input space in order to create mirror responses (on both sides of the absolute value nonlinearity). By composing these folding operations, we obtain an exponentially large number of piecewise linear regions which can capture all kinds of regular (e.g., repeating) patterns.

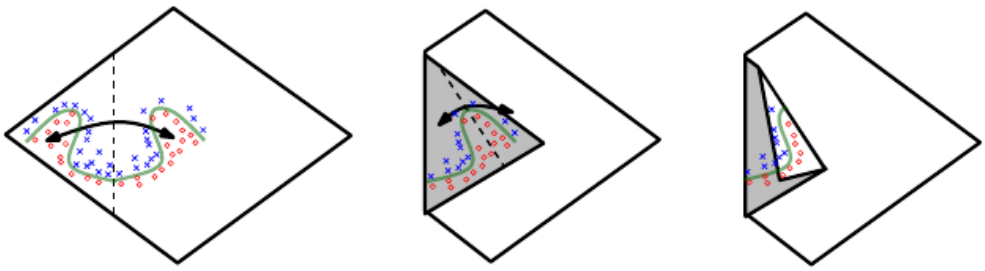

Figure 6.5: An intuitive, geometric explanation of the exponential advantage of deeper rectifier networks formally by Montufar *et al.* (2014). *(Left)*An absolute value rectification unit has the same output for every pair of mirror points in its input. The mirror axis of symmetry is given by the hyperplane defined by the weights and bias of the unit. A function computed on top of that unit (the green decision surface) will be a mirror image of a simpler pattern across that axis of symmetry. *(Center)*The function can be obtained by folding the space around the axis of symmetry. *(Right)*Another repeating pattern can be folded on top of the first (by another downstream unit) to obtain another symmetry (which is now repeated four times, with two hidden layers). Figure reproduced with permission from Montufar *et al.* (2014).

More precisely, the main theorem in Montufar *et al.* (2014) states that the number of linear regions carved out by a deep rectifier network with d inputs, depth l, and n units per hidden layer, is

$$O\left(\binom{n}{d}^{d(l-1)} n^d\right), \tag{6.42}$$

i.e., exponential in the depth l. In the case of maxout networks with k filters per unit, the number of linear regions is

$$O\left(k^{(l-1)+d}\right). \tag{6.43}$$

Of course, there is no guarantee that the kinds of functions we want to learn in applications of machine learning (and in particular for AI) share such a property.

We may also want to choose a deep model for statistical reasons. Any time we choose a specific machine learning algorithm, we are implicitly stating some set of prior beliefs we have about what kind of function the algorithm should learn. Choosing a deep model encodes a very general belief that the function we want to learn should involve composition of several simpler functions. This can be interpreted from a representation learning point of view as saying that we believe the learning problem consists of discovering a set of underlying factors of variation that can in turn be described in terms of other, simpler underlying factors of variation. Alternately, we can interpret the use of a deep architecture as expressing a belief that the function we want to learn is a computer program consisting of multiple steps, where each step makes use of the previous step's output. These intermediate outputs are not necessarily factors of variation, but can instead be analogous to counters or pointers that the network uses to organize its internal processing. Empirically, greater depth does seem to result in better generalization for a wide variety of tasks (Bengio *et al.*, 2007; Erhan *et al.*, 2009; Bengio, 2009; Mesnil *et al.*, 2011; Ciresan *et al.*, 2012; Krizhevsky *et al.*, 2012; Sermanet *et al.*, 2013; Farabet *et al.*, 2013; Couprie *et al.*, 2013; Kahou *et al.*, 2013; Goodfellow *et al.*, 2014d; Szegedy *et al.*, 2014a). See figure 6.6 and figure 6.7 for examples of some of these empirical results. This suggests that using deep architectures does indeed express a useful prior over the space of functions the model learns.

6.4.2 Other Architectural Considerations

So far we have described neural networks as being simple chains of layers, with the main considerations being the depth of the network and the width of each layer. In practice, neural networks show considerably more diversity.

Many neural network architectures have been developed for specific tasks. Specialized architectures for computer vision called convolutional networks are described in chapter 9. Feedforward networks may also be generalized to the recurrent neural networks for sequence processing, described in chapter 10, which have their own architectural considerations.

In general, the layers need not be connected in a chain, even though this is the most common practice. Many architectures build a main chain but then add extra architectural features to it, such as skip connections going from layer i to layer $i+2$ or higher. These skip connections make it easier for the gradient to flow from output layers to layers nearer the input.

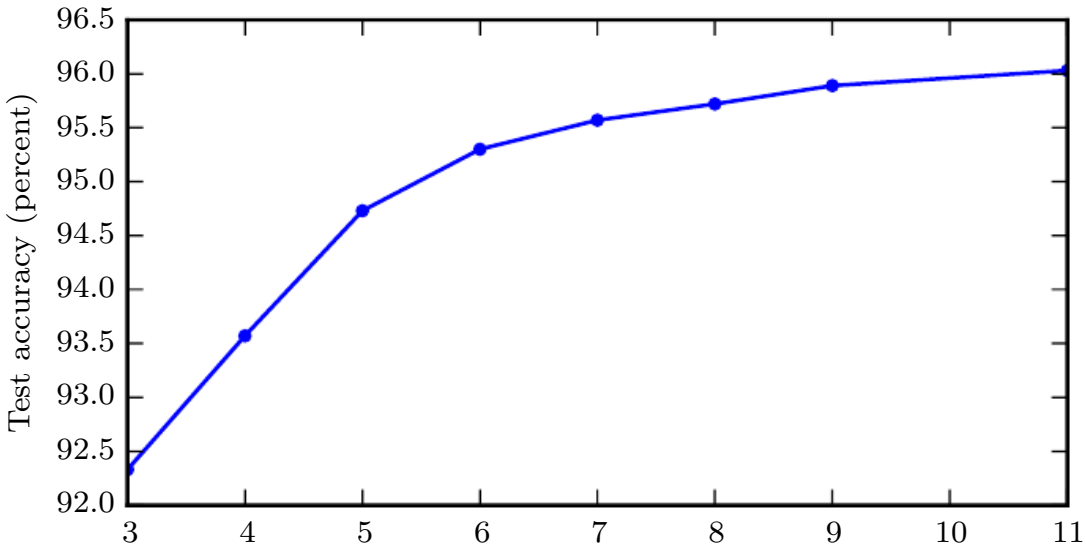

Figure 6.6: Empirical results showing that deeper networks generalize better when used to transcribe multi-digit numbers from photographs of addresses. Data from Goodfellow *et al.* (2014d). The test set accuracy consistently increases with increasing depth. See figure 6.7 for a control experiment demonstrating that other increases to the model size do not yield the same effect.

Another key consideration of architecture design is exactly how to connect a pair of layers to each other. In the default neural network layer described by a linear transformation via a matrix $\boldsymbol{W}$, every input unit is connected to every output unit. Many specialized networks in the chapters ahead have fewer connections, so that each unit in the input layer is connected to only a small subset of units in the output layer. These strategies for reducing the number of connections reduce the number of parameters and the amount of computation required to evaluate the network, but are often highly problem-dependent. For example, convolutional networks, described in chapter 9, use specialized patterns of sparse connections that are very effective for computer vision problems. In this chapter, it is difficult to give much more specific advice concerning the architecture of a generic neural network. Subsequent chapters develop the particular architectural strategies that have been found to work well for different application domains.

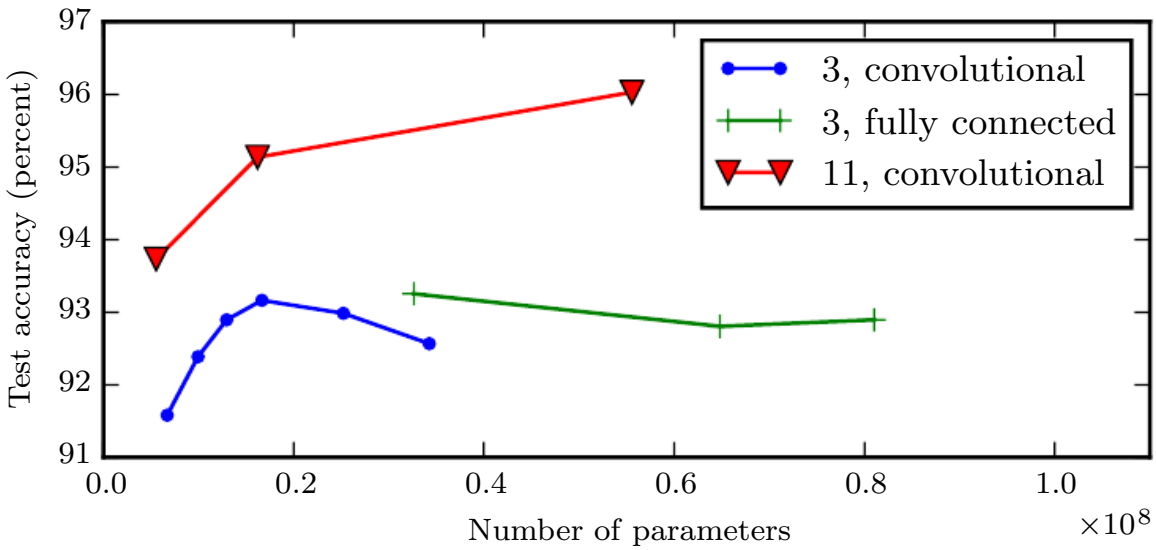

Figure 6.7: Deeper models tend to perform better. This is not merely because the model is larger. This experiment from Goodfellow *et al.* (2014d) shows that increasing the number of parameters in layers of convolutional networks without increasing their depth is not nearly as effective at increasing test set performance. The legend indicates the depth of network used to make each curve and whether the curve represents variation in the size of the convolutional or the fully connected layers. We observe that shallow models in this context overfit at around 20 million parameters while deep ones can benefit from having over 60 million. This suggests that using a deep model expresses a useful preference over the space of functions the model can learn. Specifically, it expresses a belief that the function should consist of many simpler functions composed together. This could result either in learning a representation that is composed in turn of simpler representations (e.g., corners defined in terms of edges) or in learning a program with sequentially dependent steps (e.g., first locate a set of objects, then segment them from each other, then recognize them).

6.5 Back-Propagation and Other Differentiation Algorithms

When we use a feedforward neural network to accept an input $\boldsymbol{x}$ and produce an output $\hat{\boldsymbol{y}}$, information flows forward through the network. The inputs $\boldsymbol{x}$ provide the initial information that then propagates up to the hidden units at each layer and finally produces $\hat{\boldsymbol{y}}$. This is called **forward propagation**. During training, forward propagation can continue onward until it produces a scalar cost $J(\boldsymbol{\theta})$. The **back-propagation** algorithm (Rumelhart *et al.*, 1986a), often simply called **backprop**, allows the information from the cost to then flow backwards through the network, in order to compute the gradient.

Computing an analytical expression for the gradient is straightforward, but numerically evaluating such an expression can be computationally expensive. The back-propagation algorithm does so using a simple and inexpensive procedure.

The term back-propagation is often misunderstood as meaning the whole learning algorithm for multi-layer neural networks. Actually, back-propagation refers only to the method for computing the gradient, while another algorithm, such as stochastic gradient descent, is used to perform learning using this gradient. Furthermore, back-propagation is often misunderstood as being specific to multi-layer neural networks, but in principle it can compute derivatives of any function (for some functions, the correct response is to report that the derivative of the function is undefined). Specifically, we will describe how to compute the gradient $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}, \boldsymbol{y})$ for an arbitrary function f, where $\boldsymbol{x}$ is a set of variables whose derivatives are desired, and $\boldsymbol{y}$ is an additional set of variables that are inputs to the function but whose derivatives are not required. In learning algorithms, the gradient we most often require is the gradient of the cost function with respect to the parameters, $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. Many machine learning tasks involve computing other derivatives, either as part of the learning process, or to analyze the learned model. The back-propagation algorithm can be applied to these tasks as well, and is not restricted to computing the gradient of the cost function with respect to the parameters. The idea of computing derivatives by propagating information through a network is very general, and can be used to compute values such as the Jacobian of a function f with multiple outputs. We restrict our description here to the most commonly used case where f has a single output.

6.5.1 Computational Graphs

So far we have discussed neural networks with a relatively informal graph language. To describe the back-propagation algorithm more precisely, it is helpful to have a more precise **computational graph** language.

Many ways of formalizing computation as graphs are possible.

Here, we use each node in the graph to indicate a variable. The variable may be a scalar, vector, matrix, tensor, or even a variable of another type.

To formalize our graphs, we also need to introduce the idea of an **operation**. An operation is a simple function of one or more variables. Our graph language is accompanied by a set of allowable operations. Functions more complicated than the operations in this set may be described by composing many operations together.

Without loss of generality, we define an operation to return only a single output variable. This does not lose generality because the output variable can have multiple entries, such as a vector. Software implementations of back-propagation usually support operations with multiple outputs, but we avoid this case in our description because it introduces many extra details that are not important to conceptual understanding.

If a variable y is computed by applying an operation to a variable x, then we draw a directed edge from x to y. We sometimes annotate the output node with the name of the operation applied, and other times omit this label when the operation is clear from context.

Examples of computational graphs are shown in figure 6.8.

6.5.2 Chain Rule of Calculus

The chain rule of calculus (not to be confused with the chain rule of probability) is used to compute the derivatives of functions formed by composing other functions whose derivatives are known. Back-propagation is an algorithm that computes the chain rule, with a specific order of operations that is highly efficient.

Let x be a real number, and let f and g both be functions mapping from a real number to a real number. Suppose that $y = g(x)$ and $z = f(g(x)) = f(y)$. Then the chain rule states that

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}. \tag{6.44}$$

We can generalize this beyond the scalar case. Suppose that $\boldsymbol{x} \in \mathbb{R}^m$, $\boldsymbol{y} \in \mathbb{R}^n$,

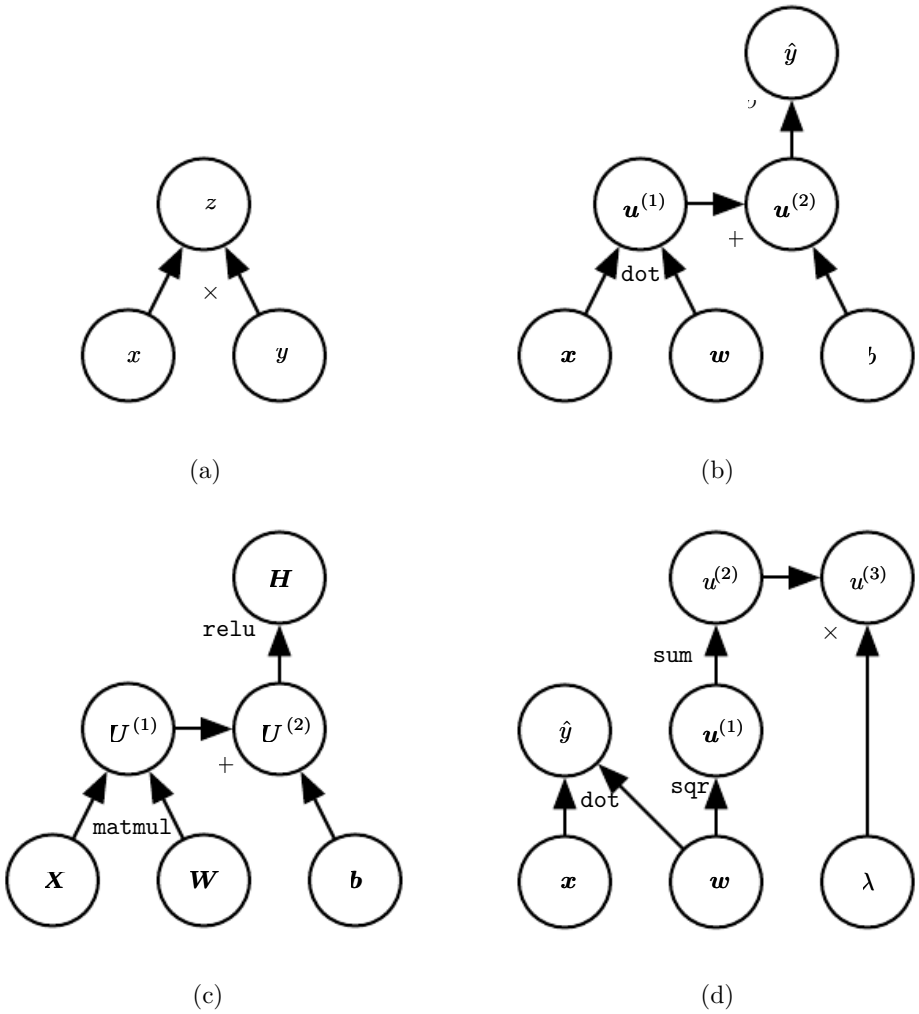

Figure 6.8: Examples of computational graphs. *(a)*The graph using the $\times$ operation to compute $z = xy$. *(b)*The graph for the logistic regression prediction $\hat{y} = \sigma\left(\boldsymbol{x}^\top \boldsymbol{w} + b\right)$. Some of the intermediate expressions do not have names in the algebraic expression but need names in the graph. We simply name the i-th such variable $\boldsymbol{u}^{(i)}$. *(c)*The computational graph for the expression $\boldsymbol{H} = \max\{0, \boldsymbol{X}\boldsymbol{W} + \boldsymbol{b}\}$, which computes a design matrix of rectified linear unit activations $\boldsymbol{H}$ given a design matrix containing a minibatch of inputs $\boldsymbol{X}$. *(d)*Examples a–c applied at most one operation to each variable, but it is possible to apply more than one operation. Here we show a computation graph that applies more than one operation to the weights $\boldsymbol{w}$ of a linear regression model. The weights are used to make both the prediction $\hat{y}$ and the weight decay penalty $\lambda \sum_i w_i^2$.

g maps from $\mathbb{R}^m$ to $\mathbb{R}^n$, and f maps from $\mathbb{R}^n$ to $\mathbb{R}$. If $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$, then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}. \tag{6.45}$$

In vector notation, this may be equivalently written as

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^\top \nabla_{\boldsymbol{y}} z, \tag{6.46}$$

where $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ is the $n \times m$ Jacobian matrix of g.

From this we see that the gradient of a variable $\boldsymbol{x}$ can be obtained by multiplying a Jacobian matrix $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ by a gradient $\nabla_{\boldsymbol{y}} z$. The back-propagation algorithm consists of performing such a Jacobian-gradient product for each operation in the graph.

Usually we do not apply the back-propagation algorithm merely to vectors, but rather to tensors of arbitrary dimensionality. Conceptually, this is exactly the same as back-propagation with vectors. The only difference is how the numbers are arranged in a grid to form a tensor. We could imagine flattening each tensor into a vector before we run back-propagation, computing a vector-valued gradient, and then reshaping the gradient back into a tensor. In this rearranged view, back-propagation is still just multiplying Jacobians by gradients.

To denote the gradient of a value z with respect to a tensor $\mathbf{X}$, we write $\nabla_{\mathbf{X}} z$, just as if $\mathbf{X}$ were a vector. The indices into $\mathbf{X}$ now have multiple coordinates—for example, a 3-D tensor is indexed by three coordinates. We can abstract this away by using a single variable i to represent the complete tuple of indices. For all possible index tuples i, $(\nabla_{\mathbf{X}} z)_i$ gives $\frac{\partial z}{\partial X_i}$. This is exactly the same as how for all possible integer indices i into a vector, $(\nabla_{\boldsymbol{x}} z)_i$ gives $\frac{\partial z}{\partial x_i}$. Using this notation, we can write the chain rule as it applies to tensors. If $\mathbf{Y} = g(\mathbf{X})$ and $z = f(\mathbf{Y})$, then

$$\nabla_{\mathbf{X}} z = \sum_j (\nabla_{\mathbf{X}} Y_j) \frac{\partial z}{\partial Y_j}. \tag{6.47}$$

6.5.3 Recursively Applying the Chain Rule to Obtain Backprop

Using the chain rule, it is straightforward to write down an algebraic expression for the gradient of a scalar with respect to any node in the computational graph that produced that scalar. However, actually evaluating that expression in a computer introduces some extra considerations.

Specifically, many subexpressions may be repeated several times within the overall expression for the gradient. Any procedure that computes the gradient

will need to choose whether to store these subexpressions or to recompute them several times. An example of how these repeated subexpressions arise is given in figure 6.9. In some cases, computing the same subexpression twice would simply be wasteful. For complicated graphs, there can be exponentially many of these wasted computations, making a naive implementation of the chain rule infeasible. In other cases, computing the same subexpression twice could be a valid way to reduce memory consumption at the cost of higher runtime.

We first begin by a version of the back-propagation algorithm that specifies the actual gradient computation directly (algorithm 6.2 along with algorithm 6.1 for the associated forward computation), in the order it will actually be done and according to the recursive application of chain rule. One could either directly perform these computations or view the description of the algorithm as a symbolic specification of the computational graph for computing the back-propagation. However, this formulation does not make explicit the manipulation and the construction of the symbolic graph that performs the gradient computation. Such a formulation is presented below in section 6.5.6, with algorithm 6.5, where we also generalize to nodes that contain arbitrary tensors.

First consider a computational graph describing how to compute a single scalar $u^{(n)}$ (say the loss on a training example). This scalar is the quantity whose gradient we want to obtain, with respect to the n_i input nodes $u^{(1)}$ to $u^{(n_i)}$. In other words we wish to compute $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ for all $i \in \{1, 2, \ldots, n_i\}$. In the application of back-propagation to computing gradients for gradient descent over parameters, $u^{(n)}$ will be the cost associated with an example or a minibatch, while $u^{(1)}$ to $u^{(n_i)}$ correspond to the parameters of the model.

We will assume that the nodes of the graph have been ordered in such a way that we can compute their output one after the other, starting at $u^{(n_i+1)}$ and going up to $u^{(n)}$. As defined in algorithm 6.1, each node $u^{(i)}$ is associated with an operation $f^{(i)}$ and is computed by evaluating the function

$$u^{(i)} = f(\mathbb{A}^{(i)}) \tag{6.48}$$

where $\mathbb{A}^{(i)}$ is the set of all nodes that are parents of $u^{(i)}$.

That algorithm specifies the forward propagation computation, which we could put in a graph $\mathcal{G}$. In order to perform back-propagation, we can construct a computational graph that depends on $\mathcal{G}$ and adds to it an extra set of nodes. These form a subgraph $\mathcal{B}$ with one node per node of $\mathcal{G}$. Computation in $\mathcal{B}$ proceeds in exactly the reverse of the order of computation in $\mathcal{G}$, and each node of $\mathcal{B}$ computes the derivative $\frac{\partial u^{(n)}}{\partial u^{(i)}}$ associated with the forward graph node $u^{(i)}$. This is done

Algorithm 6.1 A procedure that performs the computations mapping n_i inputs $u^{(1)}$ to $u^{(n_i)}$ to an output $u^{(n)}$. This defines a computational graph where each node computes numerical value $u^{(i)}$ by applying a function $f^{(i)}$ to the set of arguments $\mathbb{A}^{(i)}$ that comprises the values of previous nodes $u^{(j)}$, $j < i$, with $j \in Pa(u^{(i)})$. The input to the computational graph is the vector $\boldsymbol{x}$, and is set into the first n_i nodes $u^{(1)}$ to $u^{(n_i)}$. The output of the computational graph is read off the last (output) node $u^{(n)}$.

for $i = 1, \ldots, n_i$ **do**
  $u^{(i)} \leftarrow x_i$
end for
for $i = n_i + 1, \ldots, n$ **do**
  $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$
  $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$
end for
return $u^{(n)}$

using the chain rule with respect to scalar output $u^{(n)}$:

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}} \tag{6.49}$$

as specified by algorithm 6.2. The subgraph $\mathcal{B}$ contains exactly one edge for each edge from node $u^{(j)}$ to node $u^{(i)}$ of $\mathcal{G}$. The edge from $u^{(j)}$ to $u^{(i)}$ is associated with the computation of $\frac{\partial u^{(i)}}{\partial u^{(j)}}$. In addition, a dot product is performed for each node, between the gradient already computed with respect to nodes $u^{(i)}$ that are children of $u^{(j)}$ and the vector containing the partial derivatives $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ for the same children nodes $u^{(i)}$. To summarize, the amount of computation required for performing the back-propagation scales linearly with the number of edges in $\mathcal{G}$, where the computation for each edge corresponds to computing a partial derivative (of one node with respect to one of its parents) as well as performing one multiplication and one addition. Below, we generalize this analysis to tensor-valued nodes, which is just a way to group multiple scalar values in the same node and enable more efficient implementations.

The back-propagation algorithm is designed to reduce the number of common subexpressions without regard to memory. Specifically, it performs on the order of one Jacobian product per node in the graph. This can be seen from the fact that backprop (algorithm 6.2) visits each edge from node $u^{(j)}$ to node $u^{(i)}$ of the graph exactly once in order to obtain the associated partial derivative $\frac{\partial u^{(i)}}{\partial u^{(j)}}$.

Algorithm 6.2 Simplified version of the back-propagation algorithm for computing the derivatives of $u^{(n)}$ with respect to the variables in the graph. This example is intended to further understanding by showing a simplified case where all variables are scalars, and we wish to compute the derivatives with respect to $u^{(1)}, \ldots, u^{(n_i)}$. This simplified version computes the derivatives of all nodes in the graph. The computational cost of this algorithm is proportional to the number of edges in the graph, assuming that the partial derivative associated with each edge requires a constant time. This is of the same order as the number of computations for the forward propagation. Each $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ is a function of the parents $u^{(j)}$ of $u^{(i)}$, thus linking the nodes of the forward graph to those added for the back-propagation graph.

Run forward propagation (algorithm 6.1 for this example) to obtain the activations of the network

Initialize `grad_table`, a data structure that will store the derivatives that have been computed. The entry `grad_table`$[u^{(i)}]$ will store the computed value of $\frac{\partial u^{(n)}}{\partial u^{(i)}}$.

`grad_table`$[u^{(n)}] \leftarrow 1$

for $j = n - 1$ down to 1 **do**

  The next line computes $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$ using stored values:

  `grad_table`$[u^{(j)}] \leftarrow \sum_{i:j \in Pa(u^{(i)})}$ `grad_table`$[u^{(i)}] \frac{\partial u^{(i)}}{\partial u^{(j)}}$

end for

return $\{$`grad_table`$[u^{(i)}] \mid i = 1, \ldots, n_i\}$

Back-propagation thus avoids the exponential explosion in repeated subexpressions. However, other algorithms may be able to avoid more subexpressions by performing simplifications on the computational graph, or may be able to conserve memory by recomputing rather than storing some subexpressions. We will revisit these ideas after describing the back-propagation algorithm itself.

6.5.4 Back-Propagation Computation in Fully-Connected MLP

To clarify the above definition of the back-propagation computation, let us consider the specific graph associated with a fully-connected multi-layer MLP.

Algorithm 6.3 first shows the forward propagation, which maps parameters to the supervised loss $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ associated with a single (input,target) training example $(\boldsymbol{x}, \boldsymbol{y})$, with $\hat{\boldsymbol{y}}$ the output of the neural network when $\boldsymbol{x}$ is provided in input.

Algorithm 6.4 then shows the corresponding computation to be done for

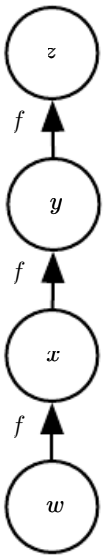

Figure 6.9: A computational graph that results in repeated subexpressions when computing the gradient. Let $w \in \mathbb{R}$ be the input to the graph. We use the same function $f : \mathbb{R} \to \mathbb{R}$ as the operation that we apply at every step of a chain: $x = f(w)$, $y = f(x)$, $z = f(y)$. To compute $\frac{\partial z}{\partial w}$, we apply equation 6.44 and obtain:

$$\frac{\partial z}{\partial w} \tag{6.50}$$

$$= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \tag{6.51}$$

$$= f'(y)f'(x)f'(w) \tag{6.52}$$

$$= f'(f(f(w)))f'(f(w))f'(w) \tag{6.53}$$

Equation 6.52 suggests an implementation in which we compute the value of $f(w)$ only once and store it in the variable x. This is the approach taken by the back-propagation algorithm. An alternative approach is suggested by equation 6.53, where the subexpression $f(w)$ appears more than once. In the alternative approach, $f(w)$ is recomputed each time it is needed. When the memory required to store the value of these expressions is low, the back-propagation approach of equation 6.52 is clearly preferable because of its reduced runtime. However, equation 6.53 is also a valid implementation of the chain rule, and is useful when memory is limited.

applying the back-propagation algorithm to this graph.

Algorithms 6.3 and 6.4 are demonstrations that are chosen to be simple and straightforward to understand. However, they are specialized to one specific problem.

Modern software implementations are based on the generalized form of back-propagation described in section 6.5.6 below, which can accommodate any computational graph by explicitly manipulating a data structure for representing symbolic computation.

Algorithm 6.3 Forward propagation through a typical deep neural network and the computation of the cost function. The loss $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ depends on the output $\hat{\boldsymbol{y}}$ and on the target $\boldsymbol{y}$ (see section 6.2.1.1 for examples of loss functions). To obtain the total cost J, the loss may be added to a regularizer $\Omega(\theta)$, where θ contains all the parameters (weights and biases). Algorithm 6.4 shows how to compute gradients of J with respect to parameters $\boldsymbol{W}$ and $\boldsymbol{b}$. For simplicity, this demonstration uses only a single input example $\boldsymbol{x}$. Practical applications should use a minibatch. See section 6.5.7 for a more realistic demonstration.

Require: Network depth, l
Require: $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
Require: $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: $\boldsymbol{x}$, the input to process
Require: $\boldsymbol{y}$, the target output
  $\boldsymbol{h}^{(0)} = \boldsymbol{x}$
  for $k = 1, \ldots, l$ **do**
    $\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$
    $\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
  end for
  $\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
  $J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$

6.5.5 Symbol-to-Symbol Derivatives

Algebraic expressions and computational graphs both operate on **symbols**, or variables that do not have specific values. These algebraic and graph-based representations are called **symbolic** representations. When we actually use or train a neural network, we must assign specific values to these symbols. We replace a symbolic input to the network $\boldsymbol{x}$ with a specific **numeric** value, such as $[1.2, 3.765, -1.8]^\top$.

Algorithm 6.4 Backward computation for the deep neural network of algorithm 6.3, which uses in addition to the input $\boldsymbol{x}$ a target $\boldsymbol{y}$. This computation yields the gradients on the activations $\boldsymbol{a}^{(k)}$ for each layer k, starting from the output layer and going backwards to the first hidden layer. From these gradients, which can be interpreted as an indication of how each layer's output should change to reduce error, one can obtain the gradient on the parameters of each layer. The gradients on weights and biases can be immediately used as part of a stochastic gradient update (performing the update right after the gradients have been computed) or used with other gradient-based optimization methods.

After the forward computation, compute the gradient on the output layer:

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$

for $k = l, l-1, \ldots, 1$ **do**

Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if f is element-wise):

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$

Compute gradients on weights and biases (including the regularization term, where needed):

$\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$

$\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$

Propagate the gradients w.r.t. the next lower-level hidden layer's activations:

$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top}\, \boldsymbol{g}$

end for

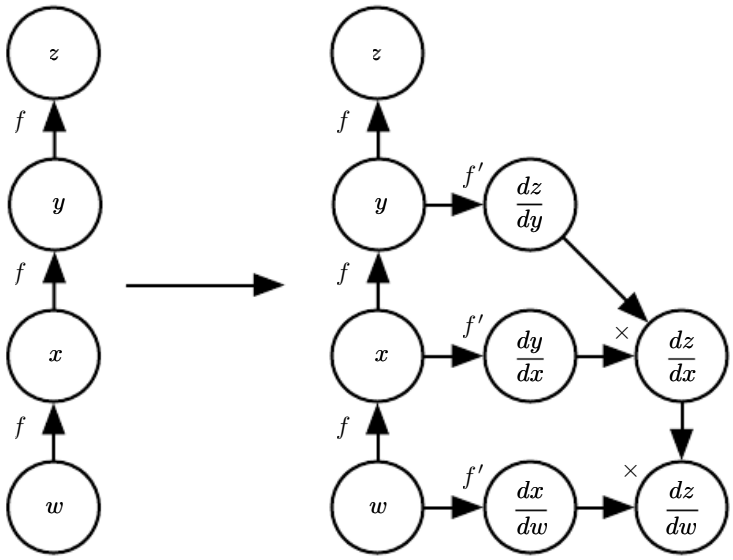

Figure 6.10: An example of the symbol-to-symbol approach to computing derivatives. In this approach, the back-propagation algorithm does not need to ever access any actual specific numeric values. Instead, it adds nodes to a computational graph describing how to compute these derivatives. A generic graph evaluation engine can later compute the derivatives for any specific numeric values. *(Left)*In this example, we begin with a graph representing $z = f(f(f(w)))$. *(Right)*We run the back-propagation algorithm, instructing it to construct the graph for the expression corresponding to $\frac{dz}{dw}$. In this example, we do not explain how the back-propagation algorithm works. The purpose is only to illustrate what the desired result is: a computational graph with a symbolic description of the derivative.

Some approaches to back-propagation take a computational graph and a set of numerical values for the inputs to the graph, then return a set of numerical values describing the gradient at those input values. We call this approach "symbol-to-number" differentiation. This is the approach used by libraries such as Torch (Collobert *et al.*, 2011b) and Caffe (Jia, 2013).

Another approach is to take a computational graph and add additional nodes to the graph that provide a symbolic description of the desired derivatives. This is the approach taken by Theano (Bergstra *et al.*, 2010; Bastien *et al.*, 2012) and TensorFlow (Abadi *et al.*, 2015). An example of how this approach works is illustrated in figure 6.10. The primary advantage of this approach is that the derivatives are described in the same language as the original expression. Because the derivatives are just another computational graph, it is possible to run back-propagation again, differentiating the derivatives in order to obtain higher derivatives. Computation of higher-order derivatives is described in section 6.5.10.

We will use the latter approach and describe the back-propagation algorithm in

terms of constructing a computational graph for the derivatives. Any subset of the graph may then be evaluated using specific numerical values at a later time. This allows us to avoid specifying exactly when each operation should be computed. Instead, a generic graph evaluation engine can evaluate every node as soon as its parents' values are available.

The description of the symbol-to-symbol based approach subsumes the symbol-to-number approach. The symbol-to-number approach can be understood as performing exactly the same computations as are done in the graph built by the symbol-to-symbol approach. The key difference is that the symbol-to-number approach does not expose the graph.

6.5.6 General Back-Propagation

The back-propagation algorithm is very simple. To compute the gradient of some scalar z with respect to one of its ancestors $\boldsymbol{x}$ in the graph, we begin by observing that the gradient with respect to z is given by $\frac{dz}{dz} = 1$. We can then compute the gradient with respect to each parent of z in the graph by multiplying the current gradient by the Jacobian of the operation that produced z. We continue multiplying by Jacobians traveling backwards through the graph in this way until we reach $\boldsymbol{x}$. For any node that may be reached by going backwards from z through two or more paths, we simply sum the gradients arriving from different paths at that node.

More formally, each node in the graph $\mathcal{G}$ corresponds to a variable. To achieve maximum generality, we describe this variable as being a tensor V. Tensor can in general have any number of dimensions. They subsume scalars, vectors, and matrices.

We assume that each variable V is associated with the following subroutines:

- `get_operation`(V): This returns the operation that computes V, represented by the edges coming into V in the computational graph. For example, there may be a Python or C++ class representing the matrix multiplication operation, and the `get_operation` function. Suppose we have a variable that is created by matrix multiplication, $\boldsymbol{C} = \boldsymbol{AB}$. Then `get_operation`($\mathsf{V}$) returns a pointer to an instance of the corresponding C++ class.

- `get_consumers`($\mathsf{V}, \mathcal{G}$): This returns the list of variables that are children of V in the computational graph $\mathcal{G}$.

- `get_inputs`($\mathsf{V}, \mathcal{G}$): This returns the list of variables that are parents of V in the computational graph $\mathcal{G}$.

Each operation `op` is also associated with a `bprop` operation. This `bprop` operation can compute a Jacobian-vector product as described by equation 6.47. This is how the back-propagation algorithm is able to achieve great generality. Each operation is responsible for knowing how to back-propagate through the edges in the graph that it participates in. For example, we might use a matrix multiplication operation to create a variable $\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B}$. Suppose that the gradient of a scalar z with respect to $\boldsymbol{C}$ is given by $\boldsymbol{G}$. The matrix multiplication operation is responsible for defining two back-propagation rules, one for each of its input arguments. If we call the `bprop` method to request the gradient with respect to $\boldsymbol{A}$ given that the gradient on the output is $\boldsymbol{G}$, then the `bprop` method of the matrix multiplication operation must state that the gradient with respect to $\boldsymbol{A}$ is given by $\boldsymbol{G}\boldsymbol{B}^\top$. Likewise, if we call the `bprop` method to request the gradient with respect to $\boldsymbol{B}$, then the matrix operation is responsible for implementing the `bprop` method and specifying that the desired gradient is given by $\boldsymbol{A}^\top \boldsymbol{G}$. The back-propagation algorithm itself does not need to know any differentiation rules. It only needs to call each operation's `bprop` rules with the right arguments. Formally, $\texttt{op.bprop}(\texttt{inputs}, \mathsf{X}, \mathsf{G})$ must return

$$\sum_i \left(\nabla_{\mathsf{X}} \texttt{op.f}(\texttt{inputs})_i\right) \mathsf{G}_i, \tag{6.54}$$

which is just an implementation of the chain rule as expressed in equation 6.47. Here, `inputs` is a list of inputs that are supplied to the operation, `op.f` is the mathematical function that the operation implements, X is the input whose gradient we wish to compute, and G is the gradient on the output of the operation.

The `op.bprop` method should always pretend that all of its inputs are distinct from each other, even if they are not. For example, if the `mul` operator is passed two copies of x to compute x^2, the `op.bprop` method should still return x as the derivative with respect to both inputs. The back-propagation algorithm will later add both of these arguments together to obtain $2x$, which is the correct total derivative on x.

Software implementations of back-propagation usually provide both the operations and their `bprop` methods, so that users of deep learning software libraries are able to back-propagate through graphs built using common operations like matrix multiplication, exponents, logarithms, and so on. Software engineers who build a new implementation of back-propagation or advanced users who need to add their own operation to an existing library must usually derive the `op.bprop` method for any new operations manually.

The back-propagation algorithm is formally described in algorithm 6.5.

Algorithm 6.5 The outermost skeleton of the back-propagation algorithm. This portion does simple setup and cleanup work. Most of the important work happens in the `build_grad` subroutine of algorithm 6.6

Require: $\mathbb{T}$, the target set of variables whose gradients must be computed.
Require: $\mathcal{G}$, the computational graph
Require: z, the variable to be differentiated
Let $\mathcal{G}'$ be $\mathcal{G}$ pruned to contain only nodes that are ancestors of z and descendents of nodes in $\mathbb{T}$.
Initialize `grad_table`, a data structure associating tensors to their gradients
`grad_table`$[z] \leftarrow 1$
for $\mathbf{V}$ in $\mathbb{T}$ **do**
  `build_grad`$(\mathbf{V}, \mathcal{G}, \mathcal{G}',$ `grad_table`$)$
end for
Return `grad_table` restricted to $\mathbb{T}$

In section 6.5.2, we explained that back-propagation was developed in order to avoid computing the same subexpression in the chain rule multiple times. The naive algorithm could have exponential runtime due to these repeated subexpressions. Now that we have specified the back-propagation algorithm, we can understand its computational cost. If we assume that each operation evaluation has roughly the same cost, then we may analyze the computational cost in terms of the number of operations executed. Keep in mind here that we refer to an operation as the fundamental unit of our computational graph, which might actually consist of very many arithmetic operations (for example, we might have a graph that treats matrix multiplication as a single operation). Computing a gradient in a graph with n nodes will never execute more than $O(n^2)$ operations or store the output of more than $O(n^2)$ operations. Here we are counting operations in the computational graph, not individual operations executed by the underlying hardware, so it is important to remember that the runtime of each operation may be highly variable. For example, multiplying two matrices that each contain millions of entries might correspond to a single operation in the graph. We can see that computing the gradient requires as most $O(n^2)$ operations because the forward propagation stage will at worst execute all n nodes in the original graph (depending on which values we want to compute, we may not need to execute the entire graph). The back-propagation algorithm adds one Jacobian-vector product, which should be expressed with $O(1)$ nodes, per edge in the original graph. Because the computational graph is a directed acyclic graph it has at most $O(n^2)$ edges. For the kinds of graphs that are commonly used in practice, the situation is even better. Most neural network cost functions are

Algorithm 6.6 The inner loop subroutine `build_grad`$(\mathbf{V}, \mathcal{G}, \mathcal{G}', \texttt{grad\_table})$ of the back-propagation algorithm, called by the back-propagation algorithm defined in algorithm 6.5.

Require: $\mathbf{V}$, the variable whose gradient should be added to $\mathcal{G}$ and `grad_table`.
Require: $\mathcal{G}$, the graph to modify.
Require: $\mathcal{G}'$, the restriction of $\mathcal{G}$ to nodes that participate in the gradient.
Require: `grad_table`, a data structure mapping nodes to their gradients
  if $\mathbf{V}$ is in `grad_table` **then**
    Return `grad_table`$[\mathbf{V}]$
  end if
  $i \leftarrow 1$
  for $\mathbf{C}$ in `get_consumers`$(\mathbf{V}, \mathcal{G}')$ **do**
    `op` $\leftarrow$ `get_operation`$(\mathbf{C})$
    $\mathbf{D} \leftarrow$ `build_grad`$(\mathbf{C}, \mathcal{G}, \mathcal{G}', \texttt{grad\_table})$
    $\mathbf{G}^{(i)} \leftarrow$ `op.bprop`(`get_inputs`$(\mathbf{C}, \mathcal{G}'), \mathbf{V}, \mathbf{D}$)
    $i \leftarrow i + 1$
  end for
  $\mathbf{G} \leftarrow \sum_i \mathbf{G}^{(i)}$
  `grad_table`$[\mathbf{V}] = \mathbf{G}$
  Insert $\mathbf{G}$ and the operations creating it into $\mathcal{G}$
  Return $\mathbf{G}$

roughly chain-structured, causing back-propagation to have $O(n)$ cost. This is far better than the naive approach, which might need to execute exponentially many nodes. This potentially exponential cost can be seen by expanding and rewriting the recursive chain rule (equation 6.49) non-recursively:

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{\substack{\text{path } (u^{(\pi_1)}, u^{(\pi_2)}, \ldots, u^{(\pi_t)}), \\ \text{from } \pi_1 = j \text{ to } \pi_t = n}} \prod_{k=2}^{t} \frac{\partial u^{(\pi_k)}}{\partial u^{(\pi_{k-1})}}. \tag{6.55}$$

Since the number of paths from node j to node n can grow exponentially in the length of these paths, the number of terms in the above sum, which is the number of such paths, can grow exponentially with the depth of the forward propagation graph. This large cost would be incurred because the same computation for $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ would be redone many times. To avoid such recomputation, we can think of back-propagation as a table-filling algorithm that takes advantage of storing intermediate results $\frac{\partial u^{(n)}}{\partial u^{(i)}}$. Each node in the graph has a corresponding slot in a table to store the gradient for that node. By filling in these table entries in order,

back-propagation avoids repeating many common subexpressions. This table-filling strategy is sometimes called **dynamic programming**.

6.5.7 Example: Back-Propagation for MLP Training

As an example, we walk through the back-propagation algorithm as it is used to train a multilayer perceptron.

Here we develop a very simple multilayer perception with a single hidden layer. To train this model, we will use minibatch stochastic gradient descent. The back-propagation algorithm is used to compute the gradient of the cost on a single minibatch. Specifically, we use a minibatch of examples from the training set formatted as a design matrix $\boldsymbol{X}$ and a vector of associated class labels $\boldsymbol{y}$. The network computes a layer of hidden features $\boldsymbol{H} = \max\{0, \boldsymbol{X}\boldsymbol{W}^{(1)}\}$. To simplify the presentation we do not use biases in this model. We assume that our graph language includes a `relu` operation that can compute $\max\{0, \boldsymbol{Z}\}$ element-wise. The predictions of the unnormalized log probabilities over classes are then given by $\boldsymbol{H}\boldsymbol{W}^{(2)}$. We assume that our graph language includes a `cross_entropy` operation that computes the cross-entropy between the targets $\boldsymbol{y}$ and the probability distribution defined by these unnormalized log probabilities. The resulting cross-entropy defines the cost J_{MLE}. Minimizing this cross-entropy performs maximum likelihood estimation of the classifier. However, to make this example more realistic, we also include a regularization term. The total cost

$$J = J_{\text{MLE}} + \lambda \left(\sum_{i,j} \left(W_{i,j}^{(1)} \right)^2 + \sum_{i,j} \left(W_{i,j}^{(2)} \right)^2 \right) \tag{6.56}$$

consists of the cross-entropy and a weight decay term with coefficient λ. The computational graph is illustrated in figure 6.11.

The computational graph for the gradient of this example is large enough that it would be tedious to draw or to read. This demonstrates one of the benefits of the back-propagation algorithm, which is that it can automatically generate gradients that would be straightforward but tedious for a software engineer to derive manually.

We can roughly trace out the behavior of the back-propagation algorithm by looking at the forward propagation graph in figure 6.11. To train, we wish to compute both $\nabla_{\boldsymbol{W}^{(1)}} J$ and $\nabla_{\boldsymbol{W}^{(2)}} J$. There are two different paths leading backward from J to the weights: one through the cross-entropy cost, and one through the weight decay cost. The weight decay cost is relatively simple; it will always contribute $2\lambda \boldsymbol{W}^{(i)}$ to the gradient on $\boldsymbol{W}^{(i)}$.

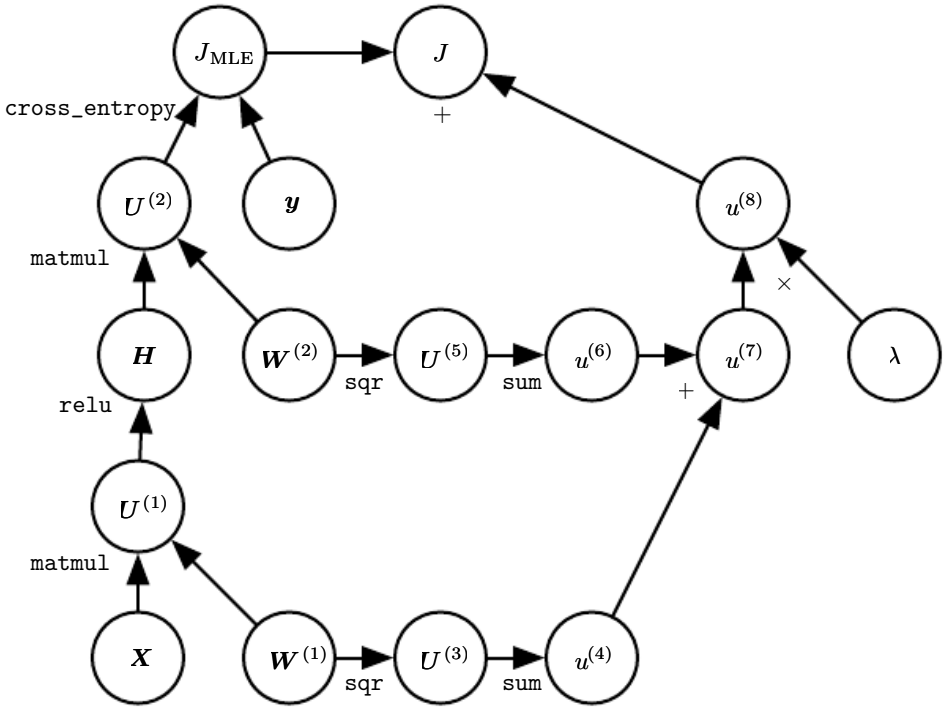

Figure 6.11: The computational graph used to compute the cost used to train our example of a single-layer MLP using the cross-entropy loss and weight decay.

The other path through the cross-entropy cost is slightly more complicated. Let $\boldsymbol{G}$ be the gradient on the unnormalized log probabilities $\boldsymbol{U}^{(2)}$ provided by the `cross_entropy` operation. The back-propagation algorithm now needs to explore two different branches. On the shorter branch, it adds $\boldsymbol{H}^\top \boldsymbol{G}$ to the gradient on $\boldsymbol{W}^{(2)}$, using the back-propagation rule for the second argument to the matrix multiplication operation. The other branch corresponds to the longer chain descending further along the network. First, the back-propagation algorithm computes $\nabla_{\boldsymbol{H}} J = \boldsymbol{G}\boldsymbol{W}^{(2)\top}$ using the back-propagation rule for the first argument to the matrix multiplication operation. Next, the `relu` operation uses its back-propagation rule to zero out components of the gradient corresponding to entries of $\boldsymbol{U}^{(1)}$ that were less than 0. Let the result be called $\boldsymbol{G}'$. The last step of the back-propagation algorithm is to use the back-propagation rule for the second argument of the `matmul` operation to add $\boldsymbol{X}^\top \boldsymbol{G}'$ to the gradient on $\boldsymbol{W}^{(1)}$.

After these gradients have been computed, it is the responsibility of the gradient descent algorithm, or another optimization algorithm, to use these gradients to update the parameters.

For the MLP, the computational cost is dominated by the cost of matrix multiplication. During the forward propagation stage, we multiply by each weight

matrix, resulting in $O(w)$ multiply-adds, where w is the number of weights. During the backward propagation stage, we multiply by the transpose of each weight matrix, which has the same computational cost. The main memory cost of the algorithm is that we need to store the input to the nonlinearity of the hidden layer. This value is stored from the time it is computed until the backward pass has returned to the same point. The memory cost is thus $O(mn_h)$, where m is the number of examples in the minibatch and n_h is the number of hidden units.

6.5.8 Complications

Our description of the back-propagation algorithm here is simpler than the implementations actually used in practice.

As noted above, we have restricted the definition of an operation to be a function that returns a single tensor. Most software implementations need to support operations that can return more than one tensor. For example, if we wish to compute both the maximum value in a tensor and the index of that value, it is best to compute both in a single pass through memory, so it is most efficient to implement this procedure as a single operation with two outputs.

We have not described how to control the memory consumption of back-propagation. Back-propagation often involves summation of many tensors together. In the naive approach, each of these tensors would be computed separately, then all of them would be added in a second step. The naive approach has an overly high memory bottleneck that can be avoided by maintaining a single buffer and adding each value to that buffer as it is computed.

Real-world implementations of back-propagation also need to handle various data types, such as 32-bit floating point, 64-bit floating point, and integer values. The policy for handling each of these types takes special care to design.

Some operations have undefined gradients, and it is important to track these cases and determine whether the gradient requested by the user is undefined.

Various other technicalities make real-world differentiation more complicated. These technicalities are not insurmountable, and this chapter has described the key intellectual tools needed to compute derivatives, but it is important to be aware that many more subtleties exist.

6.5.9 Differentiation outside the Deep Learning Community

The deep learning community has been somewhat isolated from the broader computer science community and has largely developed its own cultural attitudes

concerning how to perform differentiation. More generally, the field of **automatic differentiation** is concerned with how to compute derivatives algorithmically. The back-propagation algorithm described here is only one approach to automatic differentiation. It is a special case of a broader class of techniques called **reverse mode accumulation**. Other approaches evaluate the subexpressions of the chain rule in different orders. In general, determining the order of evaluation that results in the lowest computational cost is a difficult problem. Finding the optimal sequence of operations to compute the gradient is NP-complete (Naumann, 2008), in the sense that it may require simplifying algebraic expressions into their least expensive form.

For example, suppose we have variables $p_1, p_2, \ldots, p_n$ representing probabilities and variables $z_1, z_2, \ldots, z_n$ representing unnormalized log probabilities. Suppose we define

$$q_i = \frac{\exp(z_i)}{\sum_i \exp(z_i)}, \tag{6.57}$$

where we build the softmax function out of exponentiation, summation and division operations, and construct a cross-entropy loss $J = -\sum_i p_i \log q_i$. A human mathematician can observe that the derivative of J with respect to z_i takes a very simple form: $q_i - p_i$. The back-propagation algorithm is not capable of simplifying the gradient this way, and will instead explicitly propagate gradients through all of the logarithm and exponentiation operations in the original graph. Some software libraries such as Theano (Bergstra *et al.*, 2010; Bastien *et al.*, 2012) are able to perform some kinds of algebraic substitution to improve over the graph proposed by the pure back-propagation algorithm.

When the forward graph $\mathcal{G}$ has a single output node and each partial derivative $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ can be computed with a constant amount of computation, back-propagation guarantees that the number of computations for the gradient computation is of the same order as the number of computations for the forward computation: this can be seen in algorithm 6.2 because each local partial derivative $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ needs to be computed only once along with an associated multiplication and addition for the recursive chain-rule formulation (equation 6.49). The overall computation is therefore $O(\#\text{ edges})$. However, it can potentially be reduced by simplifying the computational graph constructed by back-propagation, and this is an NP-complete task. Implementations such as Theano and TensorFlow use heuristics based on matching known simplification patterns in order to iteratively attempt to simplify the graph. We defined back-propagation only for the computation of a gradient of a scalar output but back-propagation can be extended to compute a Jacobian (either of k different scalar nodes in the graph, or of a tensor-valued node containing k values). A naive implementation may then need k times more computation: for

each scalar internal node in the original forward graph, the naive implementation computes k gradients instead of a single gradient. When the number of outputs of the graph is larger than the number of inputs, it is sometimes preferable to use another form of automatic differentiation called **forward mode accumulation**. Forward mode computation has been proposed for obtaining real-time computation of gradients in recurrent networks, for example (Williams and Zipser, 1989). This also avoids the need to store the values and gradients for the whole graph, trading off computational efficiency for memory. The relationship between forward mode and backward mode is analogous to the relationship between left-multiplying versus right-multiplying a sequence of matrices, such as

$$\boldsymbol{ABCD}, \tag{6.58}$$

where the matrices can be thought of as Jacobian matrices. For example, if $\boldsymbol{D}$ is a column vector while $\boldsymbol{A}$ has many rows, this corresponds to a graph with a single output and many inputs, and starting the multiplications from the end and going backwards only requires matrix-vector products. This corresponds to the backward mode. Instead, starting to multiply from the left would involve a series of matrix-matrix products, which makes the whole computation much more expensive. However, if $\boldsymbol{A}$ has fewer rows than $\boldsymbol{D}$ has columns, it is cheaper to run the multiplications left-to-right, corresponding to the forward mode.

In many communities outside of machine learning, it is more common to implement differentiation software that acts directly on traditional programming language code, such as Python or C code, and automatically generates programs that differentiate functions written in these languages. In the deep learning community, computational graphs are usually represented by explicit data structures created by specialized libraries. The specialized approach has the drawback of requiring the library developer to define the `bprop` methods for every operation and limiting the user of the library to only those operations that have been defined. However, the specialized approach also has the benefit of allowing customized back-propagation rules to be developed for each operation, allowing the developer to improve speed or stability in non-obvious ways that an automatic procedure would presumably be unable to replicate.

Back-propagation is therefore not the only way or the optimal way of computing the gradient, but it is a very practical method that continues to serve the deep learning community very well. In the future, differentiation technology for deep networks may improve as deep learning practitioners become more aware of advances in the broader field of automatic differentiation.

6.5.10 Higher-Order Derivatives

Some software frameworks support the use of higher-order derivatives. Among the deep learning software frameworks, this includes at least Theano and TensorFlow. These libraries use the same kind of data structure to describe the expressions for derivatives as they use to describe the original function being differentiated. This means that the symbolic differentiation machinery can be applied to derivatives.

In the context of deep learning, it is rare to compute a single second derivative of a scalar function. Instead, we are usually interested in properties of the Hessian matrix. If we have a function $f : \mathbb{R}^n \to \mathbb{R}$, then the Hessian matrix is of size $n \times n$. In typical deep learning applications, n will be the number of parameters in the model, which could easily number in the billions. The entire Hessian matrix is thus infeasible to even represent.

Instead of explicitly computing the Hessian, the typical deep learning approach is to use **Krylov methods**. Krylov methods are a set of iterative techniques for performing various operations like approximately inverting a matrix or finding approximations to its eigenvectors or eigenvalues, without using any operation other than matrix-vector products.

In order to use Krylov methods on the Hessian, we only need to be able to compute the product between the Hessian matrix $\boldsymbol{H}$ and an arbitrary vector $\boldsymbol{v}$. A straightforward technique (Christianson, 1992) for doing so is to compute

$$\boldsymbol{H}\boldsymbol{v} = \nabla_{\boldsymbol{x}} \left[\left(\nabla_{\boldsymbol{x}} f(x) \right)^\top \boldsymbol{v} \right]. \tag{6.59}$$

Both of the gradient computations in this expression may be computed automatically by the appropriate software library. Note that the outer gradient expression takes the gradient of a function of the inner gradient expression.

If $\boldsymbol{v}$ is itself a vector produced by a computational graph, it is important to specify that the automatic differentiation software should not differentiate through the graph that produced $\boldsymbol{v}$.

While computing the Hessian is usually not advisable, it is possible to do with Hessian vector products. One simply computes $\boldsymbol{H}\boldsymbol{e}^{(i)}$ for all $i = 1, \ldots, n$, where $\boldsymbol{e}^{(i)}$ is the one-hot vector with $e_i^{(i)} = 1$ and all other entries equal to 0.

6.6 Historical Notes

Feedforward networks can be seen as efficient nonlinear function approximators based on using gradient descent to minimize the error in a function approximation.