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 that 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 5.12. That section covers some perspectives on traditional machine learning techniques that have strongly influenced the development of deep learning algorithms.

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? A popular definition of learning in the context of computer programs is "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" (Mitchell, 1997). 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 is mostly interesting because of the tasks we can accomplish with it. From an engineering point of view, 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 understanding it allows us to understand the principles that underlie intelligent behavior, and intelligent behavior is defined as being able to accomplish certain tasks.

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.

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 $f : \mathbb{R}^n \to \{1, \ldots, k\}$ which may then be applied to any input. Here the output of $f(\boldsymbol{x})$ can be interpreted as an estimate of the category that $\boldsymbol{x}$ belongs to. 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.*, 2012a; 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 premia), 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).

- *Translation*: In a 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 to translate from English to French. Deep

learning has recently begun to have an important impact on this kind of task (Sutskever *et al.*, 2014a; Bahdanau *et al.*, 2014).

- *Structured output* tasks involve any task where the output is a vector containing important relationships between the different elements. This is a broad category, and includes 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 the relative role of its constituents. 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 an 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.*, 2015; Donahue *et al.*, 2014; Karpathy and Li, 2015; Fang *et al.*, 2015; Xu *et al.*, 2015a). 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.

- *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. This 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 or probability function estimation*: In the density estimation problem, the machine learning algorithm is asked to learn a function $p_{\text{model}} : \mathbb{R}^n \to \mathbb{R}$, where $p_{\text{model}}(\boldsymbol{x})$ can be interpreted as a probability density function (if $\mathbf{x}$ is continuous) or a probability 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 that the learning algorithm has at least implicitly captured 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, then we know the distribution over it is given by $p(x_i \mid x_{-i})$. 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 listed here are only intended 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, we can measure the 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 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 objects called *examples*, with each example containing many *features* that have been objectively measured. 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 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 $\mathbf{x}$, e.g. 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 that 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

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

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

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

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.

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.

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. Each example is a collection of observations called *features* collected from a different time or place. If we wish to make a system for recognizing objects from photographs, we might use a machine learning algorithm where each example is a photograph, and the features within the example are the brightness values of each of the pixels within the photograph. If we wish to perform speech recognition, we might collect a dataset where each example is a recording of a person saying a word or sentence, and each of the features is the amplitude of the sound wave at a particular moment in time.

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. Different sections of this book describe how to handle different types of 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, e.g.

$\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(m)}\}$. 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.

5.2 Example: Linear Regression

In the previous section, we saw that a machine learning algorithm is an algorithm that is capable of improving a computer program's performance at some task via experience. Now it is time to define some specific machine learning algorithms.

Let's begin with an example of a simple machine learning algorithm: *linear regression*. In this section, we will only describe what the linear regression algorithm does. We wait until later sections of this chapter to justify the algorithm and show more formally that it actually works.

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

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

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.

Let's 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*. Let's 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 $\hat{\boldsymbol{y}}^{(\text{test})}$ is the predictions of the model on the test set, then the mean squared error is given by

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

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

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

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) 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 $\boldsymbol{0}$:

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

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

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

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

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

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

$$\Rightarrow 2\boldsymbol{X} \qquad\qquad \boldsymbol{w} - 2\boldsymbol{X}$$

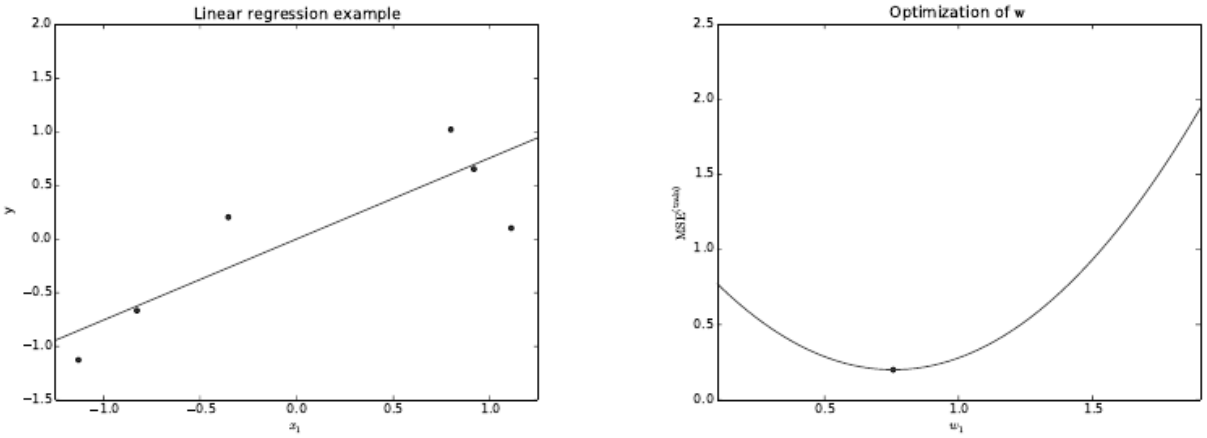

Figure 5.1: Consider this example linear regression problem, with a training set consisting of 5 data points, each containing one feature. This means that the weight vector $\boldsymbol{w}$ contains only a single parameter to learn, w_1. *(Left)* Observe that linear regression learns to set w_1 such 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.

$$\Rightarrow \boldsymbol{w} = (\boldsymbol{X}^{(\mathrm{train})\top} \boldsymbol{X}^{(\mathrm{train})})^{-1} \boldsymbol{X}^{(\mathrm{train})\top} \boldsymbol{y}^{(\mathrm{train})} \tag{5.1}$$

The system of equations defined by Eq. 5.1 is known as the *normal equations*. Solving these equations constitutes a simple learning algorithm. For an example of the linear regression learning algorithm in action, see Fig. 5.1.

It's 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

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

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. We will frequently use the term "linear" when referring to affine functions throughout this book.

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.

5.3 Generalization, 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* 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 separate from the training set.

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

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

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 only get to observe 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.

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. We call that shared underlying distribution the *data generating distribution*, or *data generating process* (which is particularly relevant if the examples are not independent). This probabilistic framework allows 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 for a randomly selected model, the two have the same expected value. 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 $\boldsymbol{w}$, then the expected training set error under this sampling process is exactly the same

as the expected test set error under this sampling process. The only difference between the two conditions is the name we assign to the dataset we sample. From this observation, we can see that it is natural for there to be some relationship between training and test error under these assumptions.

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, i.e., memorize 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 choose 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

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

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:

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

Note that this 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:

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

Machine learning algorithms will generally perform best when their capacity is appropriate in regard to 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. Model with high capacity can solve complex tasks, but when their capacity is higher than needed to solve the present task they may overfit.

Fig. 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 the true structure of the task so it generalizes well to new data.

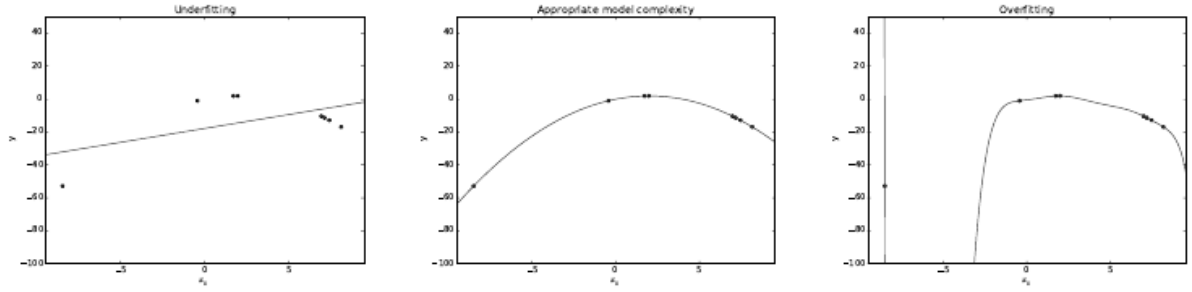

Figure 5.2: We fit three models to this example training set. The training data was generated synthetically, by randomly sampling x values and choosing y deterministically by evaluating a quadratic function. (*Left*) A linear function fit to the data suffers from underfitting—it cannot capture the curvature 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 pseudo-inverse 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.

Here we have only described 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 many other ways of controlling the capacity of a machine learning algorithm, which we will explore in the sections ahead.

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 and showing that the discrepancy between training error and generalization error is bounded by a quantity that grows with the ratio of capacity to number of training examples (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.

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 your 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 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 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*. Unlikely 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

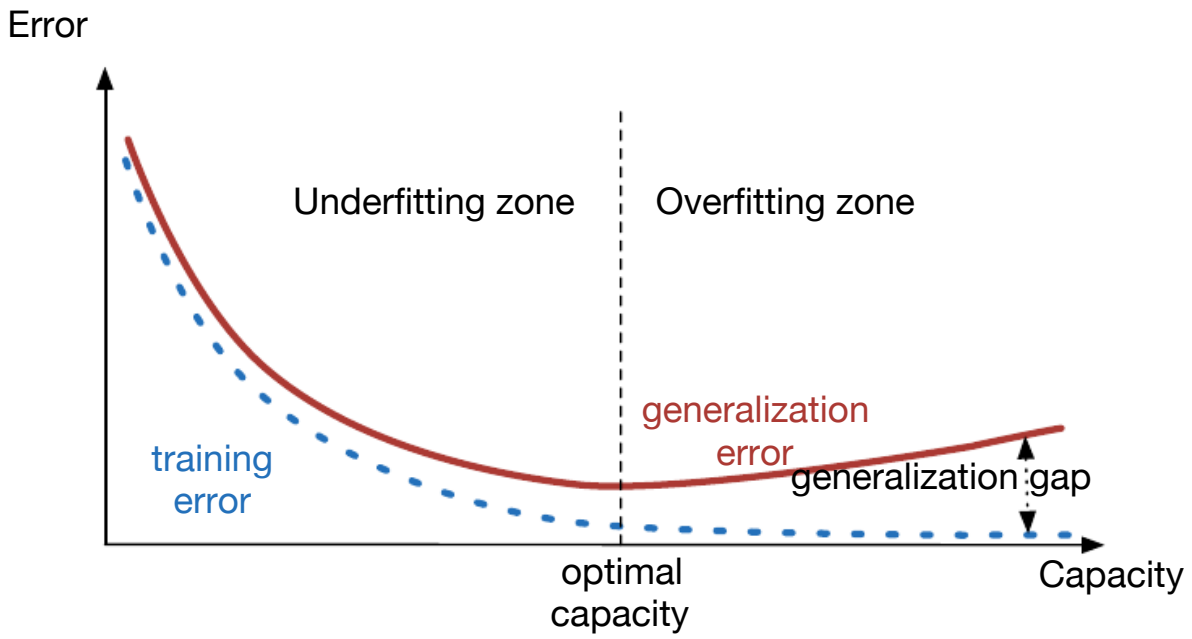

Figure 5.3: Typical relationship between capacity (horizontal axis) and both training (bottom curve, dotted) and generalization (or test) error (top curve, bold). At the left end of the graph, training error and generalization error are both high. This is the *underfitting regime*. As we increase capacity, 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*.

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 $i = \arg\min ||\boldsymbol{X}_{i,:} - \boldsymbol{x}||_2^2$. This learning 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 regression, the mapping from $\boldsymbol{x}$ to y may be inherently stochastic, or $\boldsymbol{x}$ may contain insufficient information to perfectly predict y, resulting in a probability distribution over possible values for y. The error incurred by an oracle making predictions from the true distribution $p(\boldsymbol{x}, y)$ is called the *Bayes error*.

Training and generalization vary as the size of the training set varies. Expected generalization error can never decrease as the number of training examples decreases. For non-parameteric 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 a higher error value. See Fig. 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.

It's worth mentioning that capacity is not just determined by which model we use. 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, just one that significantly reduces the training error. These additional restrictions mean that the model's *effective capacity* may be less than its *representational capacity.*

5.3.1 The No Free Lunch Theorem

Although learning theory, sketched above, suggests that it is possible to generalize, one should consider a serious caveat, discussed here. Generally speaking, 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. One may wonder then how the

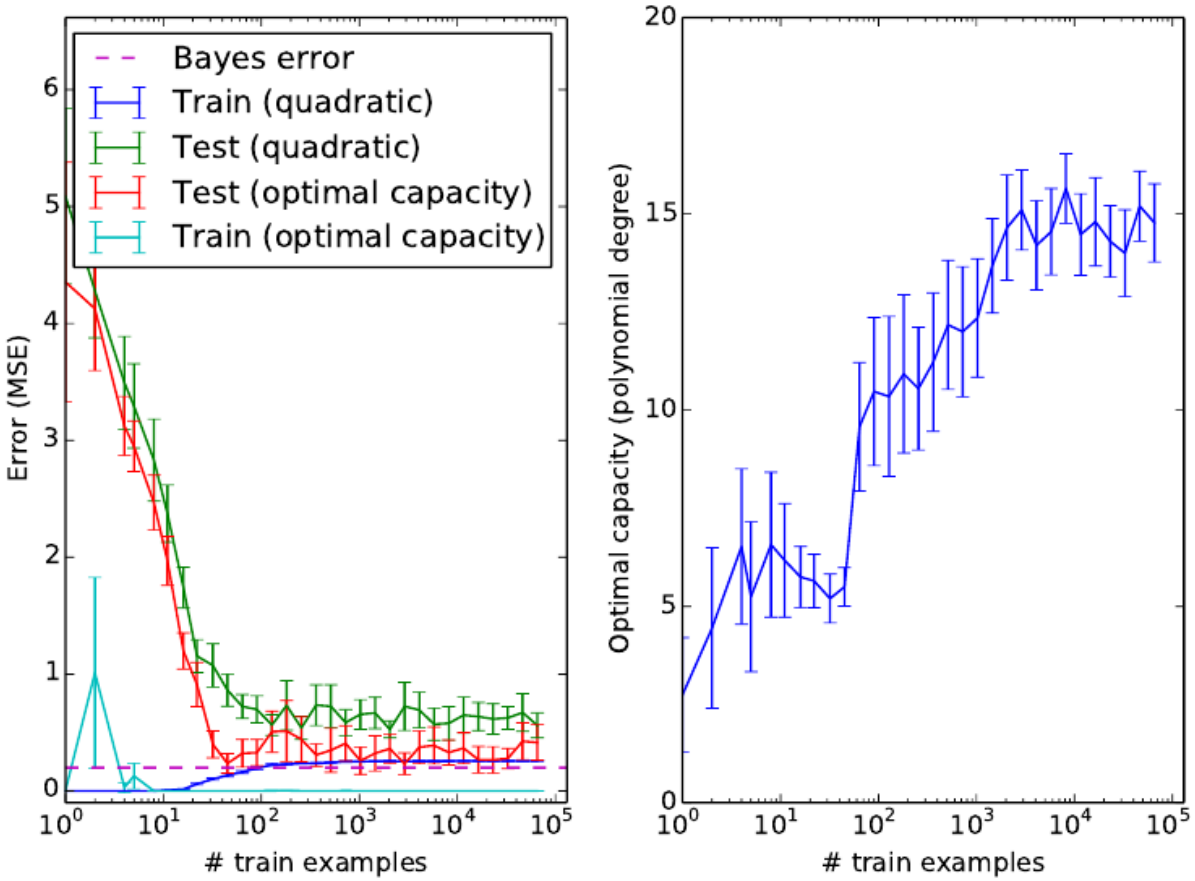

Figure 5.4: The effect of the training dataset size on the train and test error of the model, as well as on the optimal model capacity. We used a synthetic regression problem, generated a single test set, and then generated several different sizes of training set. For each size, we generated 40 different training sets in order to plot error bars showing 95% confidence intervals. Left) The mean squared error on the train and test set for two different models. One is a quadratic model, while the other has its degree chosen to minimize the test error. Both are fit in closed form, using the Moore-Penrose pseudoinverse to solve the normal equations. 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. However, the quadratic model does not have enough capacity to solve the task, so its test error asymptotes to a high value. Ultimately test error of the optimal capacity model asymptotes to the Bayes error. The training error of the optimal capacity model can dip below the Bayes error, due to the ability of the training algorithm to memorize specific instances of this noise, but as the training size increases to infinity the training error of any model must rise to at least the Bayes error. Right) As the training set size increases, the optimal capacity increases. We observe this effect here by plotting the degree of the optimal polynomial regressor. The optimal capacity plateaus after reaching a level of sufficient complexity to solve the task.

claims that machine learning can generalize well are logically valid.

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 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.

5.3.2 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 we have discussed is to increase or decrease the model's 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 non-linear 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 may only be chosen 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 not only the mean squared error on the training set, but instead a criterion $J(\boldsymbol{w})$ that expresses a preference for the weights to have smaller squared L^2 norm. Specifically,

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

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 Fig. 5.5 for the results.

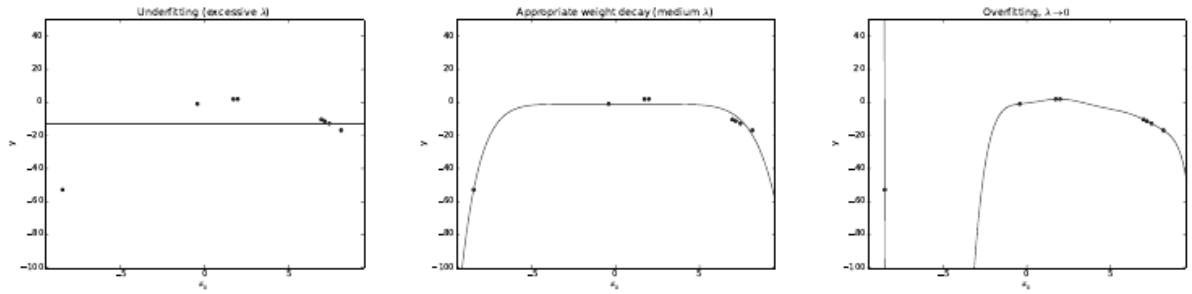

Figure 5.5: We fit a high-degree polynomial regression model to our example training set from Fig. 5.2. The true function is quadratic, but here we use only models with degree 9. We vary the amount of weight decay to prevent these high-degree models from overfitting. (*Left*) With very large λ, we can 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 pseudo-inverse to solve the underdetermined problem with minimal regularization), the degree-9 polynomial overfits significantly, as we saw in Fig. 5.2.

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, rivalled in its importance only by optimization. KEY IDEA

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.

5.4 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 Fig. 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, we do not learn the 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 $\lambda = 0$..

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.

For this reason, 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. Since the validation set is used to "train" the hyperparameters, the validation set error will underestimate the test set error, though typically by a smaller amount than the training error. Typically, one uses about 80% of the data for training and 20% for validation.

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.

5.4.1 Cross-Validation

One issue with the idea of splitting the dataset into train/test or train/validation/test subsets is that only a small fraction of examples are used to evaluate generalization. The consequence is that there is a lot of 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.

With large datasets with hundreds of thousands of examples or more, this is not a serious issue, but when the dataset is too small, there are alternative procedures, which allow 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 / 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, in which a partition of the dataset is formed by splitting it in k non-overlapping subsets. Then k train/test splits can be obtained by keeping each time the i-th subset as a test set and the rest as a training set. The average test error across all these k training/testing experiments can then be reported. One problem is that there exists no unbiased estimators of the variance of such average error estimators (Bengio and Grandvalet, 2004), but approximations are typically used.

If model selection or hyperparameter optimization is required, things get more computationally expensive: one can recurse the k-fold cross-validation idea, inside the training set. So we can have an outer loop that estimates test error and provides a "training set" for a hyperparameter-free learner, calling it k times to "train". That hyperparameter-free learner can then split its received training set by k-fold cross-validation into internal training/validation subsets (for example, splitting into $k-1$ subsets is convenient, to reuse the same test blocks as the outer loop), call a hyperparameter-specific learner for each choice of hyperparameter value on each of the training partition of this inner loop, and compute the validation error by averaging across the $k-1$ validation sets the errors made by the $k-1$ hyperparameter-specific learners trained on each of the internal training subsets.

5.5 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.

5.5.1 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 5.2, 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 θ by $\hat{\theta}$.

Let $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ be a set of m *independent and identically distributed* (i.i.d.) data points. A **point estimator** is any function of the data:

$$\hat{\theta}_m = g(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}). \tag{5.2}$$

In other words, any statistic[1] is a point estimate. Notice that no mention is made of any correspondence between the estimator and the parameter being estimated. There is also no constraint that the range of $g(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)})$ should correspond to that of the true parameter.

This definition of a point estimator is very general and allows the designer of an estimator great flexibility. What distinguishes "just any" function of the data from most of the estimators that are in common usage is their properties. 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 (or vector) $\boldsymbol{y}$ given an input vector $\boldsymbol{x}$ (also called the covariates). We consider that there is a function $f(\boldsymbol{x})$ that describes the relationship

[1] A statistic is a function of the data, typically of the whole training set, such as the mean.

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 $\hat{f}$. Note that we are really not adding anything new here to our notion of a point estimator, the function estimator $\hat{f}$ is simply a point estimator in function space.

The linear regression example we discussed above in Section. 5.2 and the polynomial regression example discussed in Section. 5.3 are both examples of function estimation where we estimate a model $\hat{f}$ of the relationship between an input $\boldsymbol{x}$ and target $\boldsymbol{y}$.

In the following we will review the most commonly studied properties of point estimators and discuss what they tell us about these estimators.

As $\hat{\boldsymbol{\theta}}$ and $\hat{f}$ are random variables (or vectors, or functions), they are distributed according to some probability distribution. We refer to this distribution as the *sampling distribution*. When we discuss properties of the estimator, we are really describing properties of the sampling distribution.

5.5.2 Bias

The bias of an estimator is defined as:

$$\text{bias}(\hat{\boldsymbol{\theta}}_m) = \mathbb{E}(\hat{\boldsymbol{\theta}}_m) - \boldsymbol{\theta} \tag{5.3}$$

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

Example: Bernoulli Distribution Consider a set of samples $\{x^{(1)}, \ldots, x^{(m)}\}$ that are independently and identically distributed according to a Bernoulli distribution, $x^{(i)} \in \{0, 1\}$, where $i \in [1, m]$. The Bernoulli *p.m.f.* (probability mass function, or probability function) is given by $P(x^{(i)}; \theta) = \theta^{x^{(i)}} (1 - \theta)^{(1-x^{(i)})}$.

We are interested in knowing if the estimator $\hat{\theta}_m = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$ is biased.

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

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 (Normal) distribution ($x^{(i)} \sim \text{Gaussian}(\mu, \sigma^2)$, where $i \in [1, m]$). The Gaussian *p.d.f.* (probability density function) is given by $p(x^{(i)}; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\frac{(x^{(i)}-\mu)^2}{\sigma^2}\right)$.

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

$$
\hat{\mu}_m = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \tag{5.4}
$$

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

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

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

Example: Gaussian Distribution Estimators of the Variance Sticking with the Gaussian family of distributions. We consider two different estimators of the variance parameter σ^2. We are interested in knowing if either estimator is biased.

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

$$\hat{\sigma}_m^2 = \frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)} - \hat{\mu}_m\right)^2, \tag{5.5}$$

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

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

We now simplify the term $\mathbb{E}[\hat{\sigma}_m^2]$

$$\begin{aligned}
\mathbb{E}[\hat{\sigma}_m^2] &= \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)} - \hat{\mu}_m\right)^2\right] \\
&= \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m}(x^{(i)})^2 - 2x^{(i)}\hat{\mu}_m + \hat{\mu}_m^2\right] \\
&= \frac{1}{m}\sum_{i=1}^{m}\mathbb{E}\left[(x^{(i)})^2\right] - 2\mathbb{E}\left[x^{(i)}\frac{1}{m}\sum_{j=1}^{m}x^{(j)}\right] + \mathbb{E}\left[\left(\frac{1}{m}\sum_{j=1}^{m}x^{(j)}\right)\left(\frac{1}{m}\sum_{k=1}^{m}x^{(k)}\right)\right] \\
&= \frac{1}{m}\sum_{i=1}^{m}\left(\left(1-\frac{2}{m}\right)\mathbb{E}\left[(x^{(i)})^2\right] - \frac{2}{m}\sum_{j\neq i}\mathbb{E}\left[x^{(i)}x^{(j)}\right] + \frac{1}{m^2}\sum_{j=1}^{m}\mathbb{E}\left[(x^{(j)})^2\right]\right. \\
&\quad \left. + \frac{1}{m^2}\sum_{j=1}^{m}\sum_{k\neq j}\mathbb{E}\left[x^{(j)}x^{(k)}\right]\right) \\
&= \frac{1}{m}\sum_{i=1}^{m}\left(\left(\frac{m-2}{m}\right)(\mu^2+\sigma^2) - \frac{2(m-1)}{m}(\mu^2) + \frac{1}{m}(\mu^2+\sigma^2) + \frac{(m-1)}{m}(\mu^2)\right) \\
&= \frac{m-1}{m}\sigma^2
\end{aligned}$$

So the bias of $\hat{\sigma}_m^2$ is $-\sigma^2/m$. Therefore, the sample variance is a biased estimator.

We now consider a modified estimator of the variance sometimes called the *unbiased sample variance*:

$$\tilde{\sigma}^2_m = \frac{1}{m-1}\sum_{i=1}^{m}\left(x^{(i)} - \hat{\mu}_m\right)^2 \tag{5.6}$$

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

$$\begin{aligned}
\mathbb{E}[\tilde{\sigma}^2_m] &= \mathbb{E}\left[\frac{1}{m-1}\sum_{i=1}^{m}\left(x^{(i)} - \hat{\mu}_m\right)^2\right] \\
&= \frac{m}{m-1}\mathbb{E}[\hat{\sigma}^2_m] \\
&= \frac{m}{m-1}\left(\frac{m-1}{m}\sigma^2\right) \\
&= \sigma^2.
\end{aligned}$$

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.

5.5.3 Variance

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*.

$$\mathrm{Var}(\hat{\theta}) = \mathbb{E}[\hat{\theta}^2] - \mathbb{E}[\hat{\theta}]^2 \tag{5.7}$$

The variance 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.

We can also define the standard error (se) of the estimator as

$$\mathrm{se}(\hat{\theta}) = \sqrt{\mathrm{Var}[\hat{\theta}]} \tag{5.8}$$

Example: Bernoulli Distribution Let's once again consider a set 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 $\hat{\theta}_m = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$.

$$\begin{aligned}
\text{Var}\left(\hat{\theta}_m\right) &= \text{Var}\left(\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right) \\
&= \frac{1}{m^2}\sum_{i=1}^{m} \text{Var}\left(x^{(i)}\right) \\
&= \frac{1}{m^2}\sum_{i=1}^{m} \theta(1-\theta) \\
&= \frac{1}{m^2} m\theta(1-\theta) \\
&= \frac{1}{m}\theta(1-\theta)
\end{aligned}$$

Note that 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 Sec. 5.5.5).

Example: Gaussian Distribution Estimators of the Variance We again consider a set of samples $\{x^{(1)}, \ldots, x^{(m)}\}$ independently and identically distributed according to a Gaussian distribution ($x^{(i)} \sim \text{Gaussian}(\mu, \sigma^2)$, where $i \in [1, m]$).

We now consider the variance of the two estimators of the variance: the sample variance,

$$\hat{\sigma}_m^2 = \frac{1}{m}\sum_{i=1}^{m}\left(x^{(1)} - \hat{\mu}_m\right)^2, \tag{5.9}$$

and the unbiased sample variance,

$$\tilde{\sigma}_m^2 = \frac{1}{m-1}\sum_{i=1}^{m}\left(x^{(1)} - \hat{\mu}_m\right)^2. \tag{5.10}$$

In order to determine the variance of these estimators we will take advantage of a known relationship between the sample variance and the Chi Squared distribution, specifically, that $\frac{m-1}{\sigma^2}\hat{\sigma}^2$ happens to be χ^2 distributed. We can then use this together with the fact that the variance of a χ^2 random variable with $m-1$ degrees of freedom is $2(m-1)$.

$$\begin{aligned}
\text{Var}\left(\frac{m-1}{\sigma^2}\tilde{\sigma}^2\right) &= 2(m-1) \\
\frac{(m-1)^2}{\sigma^4}\text{Var}\left(\tilde{\sigma}^2\right) &= 2(m-1) \\
\text{Var}\left(\tilde{\sigma}^2\right) &= \frac{2\sigma^4}{(m-1)}
\end{aligned}$$

By noticing that $\hat{\sigma}^2 = \frac{m-1}{m}\tilde{\sigma}^2$, and using $\tilde{\sigma}^2$'s relationship to the χ^2 distribution, it is straightforward to show that $\text{Var}\left(\hat{\sigma}^2\right) = \frac{2(m-1)\sigma^4}{m^2}$.

To derive this last relation, we used the fact that $\text{Var}\left(\tilde{\sigma}^2\right) = \left(\frac{m}{m-1}\right)^2 \text{Var}\left(\hat{\sigma}^2\right)$, that is $\text{Var}\left(\tilde{\sigma}^2\right) > \text{Var}(\hat{\sigma}^2)$. So while the bias of $\tilde{\sigma}^2$ is smaller than the bias of $\hat{\sigma}^2$, the variance of $\tilde{\sigma}^2$ is greater.

5.5.4 Trading off Bias and Variance and the 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 true 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, let's imagine that we are interested in approximating the function shown in Fig. 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?

In machine learning, perhaps the most common and empirically successful way to negotiate this kind of trade-off, in general is by cross-validation, discussed in Section 5.4.1. Alternatively, we can also compare the *mean squared error* (MSE) of the estimates:

$$\begin{aligned} \text{MSE} &= \mathbb{E}[(\hat{\theta_n} - \theta)^2] \\ &= \text{Bias}(\hat{\theta}_n)^2 + \text{Var}(\hat{\theta}_n) \end{aligned} \tag{5.11}$$

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 Eq. 5.11, 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.

The relationship between bias and variance is tightly linked to the machine learning concepts of capacity, underfitting and overfitting discussed in Section. 5.3. 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 of capacity, as in Section 5.3 and Figure 5.3.

Example: Gaussian Distribution Estimators of the Variance In the last section we saw that when we compared the sample variance, $\hat{\sigma}^2$, and the unbiased sample variance, $\tilde{\sigma}^2$, we see that while $\hat{\sigma}^2$ has higher bias, $\tilde{\sigma}^2$ has higher variance.

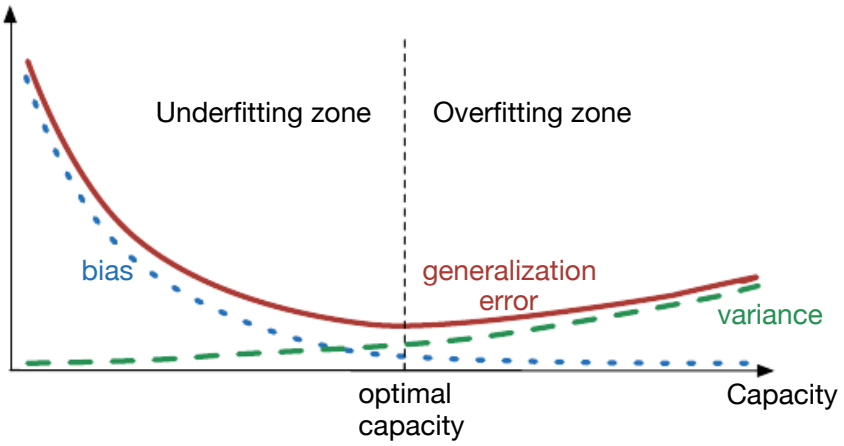

Figure 5.6: As capacity increases (x-axis), bias (dotted) decreases and variance (dashed) increases, yielding another U-shaped curve for generalization error (bold curve). If we vary capacity along one axis, there is an optimal capacity, with underfitting when the capacity is below this optimum and overfitting when it is above.

The mean squared error offers a way of balancing the tradeoff between bias and variance and suggest which estimator we might prefer. For $\hat{\sigma}^2$, the mean squared error is given by:

$$\text{MSE}(\hat{\sigma}_m^2) = \text{Bias}(\hat{\sigma}_m^2)^2 + \text{Var}(\hat{\sigma}_m^2) \tag{5.12}$$

$$= \left(\frac{-\sigma^2}{m}\right)^2 + \frac{2(m-1)\sigma^4}{m^2} \tag{5.13}$$

$$= \left(\frac{1+2(m-1)}{m^2}\right)\sigma^4 \tag{5.14}$$

$$= \left(\frac{2m-1}{m^2}\right)\sigma^4 \tag{5.15}$$

The mean squared error of the unbiased alternative is given by:

$$\text{MSE}(\tilde{\sigma}_m^2) = \text{Bias}(\tilde{\sigma}_m^2)^2 + \text{Var}(\tilde{\sigma}_m^2) \tag{5.16}$$

$$= 0 + \frac{2\sigma^4}{(m-1)} \tag{5.17}$$

$$= \frac{2}{(m-1)}\sigma^4. \tag{5.18}$$

Comparing the two, we see that the MSE of the unbiased sample variance, $\tilde{\sigma}_m^2$, is actually higher than the MSE of the (biased) sample variance, $\hat{\sigma}_m^2$. This implies that despite incurring bias in the estimator $\hat{\sigma}_m^2$, the resulting reduction in variance more than makes up for the difference, at least in a mean squared sense.

5.5.5 Consistency

As we have already discussed, sometimes we may wish to choose an estimator that is biased. For example, in order to minimize the variance of the estimator. However we might still wish that, as the number of data points in our dataset increases, our point estimates converge to the true value of the parameter. More formally, we would like that $\lim_{n\to\infty} \hat{\theta}_n \xrightarrow{p} \theta$?[2] This condition is known as *consistency*[3] and ensures that the bias induced by the estimator is assured to diminish as the number of data examples grows.

Asymptotic unbiasedness is not equivalent to consistency. For example, consider estimating the mean parameter μ of a normal distribution $\mathcal{N}(\mu, \sigma^2)$, with a dataset consisting of n samples: $\{x_1, \ldots, x_n\}$. We could use the first sample x_1 of the dataset as an *unbiased* estimator: $\hat{\theta} = x_1$, In that case, $\mathbb{E}(\hat{\theta}_n) = \theta$ 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 $\hat{\theta}_n \to \theta$ as $n \to \infty$.

5.6 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

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} p_{\text{model}}(\mathbb{X}; \boldsymbol{\theta}) \tag{5.19}$$

$$= \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{m} p_{\text{model}}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \tag{5.20}$$

[2] The symbol $\xrightarrow{p}$ means that the convergence is in probability, i.e. for any $\epsilon > 0$, $P(|\hat{\theta}_n - \theta| > \epsilon) \to 0$ as $n \to \infty$.

[3] This is sometime referred to as weak consistency, with strong consistency referring to the *almost sure* convergence of $\hat{\theta}$ to θ.

This product over many probabilities can be inconvenient for a variety of reasons. For example, it is prone to numerical underflow. To attain a more convenient but equivalent optimization problem, we observe that the logarithm of the arg max is the arg max of logarithm:

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\text{model}}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}). \tag{5.21}$$

Because the argmax 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:

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\boldsymbol{x} \sim \hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta}). \tag{5.22}$$

One way to interpret maximium likelihood estimation is to view at as minimizing the dissimilarity between the empirical distribution 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

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

Note that 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

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

which is of course the same as the maximization in Eq. 5.22. Note that this also corresponds exactly to minimizing the cross entropy between the distributions.

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. Maximimum 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.

5.6.1 Conditional Log-Likelihood and Mean Squared Error

The maximum likelihood estimator can readily be generalized to the case where our goal is not to estimate a probability function but rather a *conditional probability*, e.g., $P(\mathbf{y} \mid \mathbf{x}; \boldsymbol{\theta})$, to predict $\mathbf{y}$ given $\mathbf{x}$. This is actually the most common situation where we do supervised learning (Section 5.8), i.e., the examples are pairs $(\boldsymbol{x}, \boldsymbol{y})$. If $\boldsymbol{X}$ represents all our inputs and $\boldsymbol{Y}$ all our observed targets, then the conditional maximum likelihood estimator is

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} P(\boldsymbol{Y} \mid \boldsymbol{X}; \boldsymbol{\theta}). \tag{5.23}$$

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

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log P(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}). \tag{5.24}$$

Example: Linear Regression Let us consider as an example the special case of linear regression, introduced earlier in Section 5.2. In that case, the conditional density of $\mathbf{y}$, given $\mathbf{x} = \boldsymbol{x}$, is a Gaussian with mean $\mu(\boldsymbol{x})$ that is a learned function of $\boldsymbol{x}$, with unconditional variance σ^2. Since the examples are assumed to be i.i.d., the conditional log-likelihood (Eq. 5.23) becomes

$$\begin{aligned}
\log P(\boldsymbol{Y} \mid \boldsymbol{X}; \boldsymbol{\theta}) &= \sum_{i=1}^{m} \log P(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}) \\
&= \sum_{i=1}^{m} \frac{-1}{2\sigma^2} ||\hat{\boldsymbol{y}}^{(i)} - \boldsymbol{y}^{(i)}||^2 - m \log \sigma - \frac{m}{2} \log(2\pi)
\end{aligned}$$

where $\hat{\boldsymbol{y}}^{(i)} = \mu(\boldsymbol{x}^{(i)})$ is the output of the linear regression on the i-th input $\boldsymbol{x}^{(i)}$ and m is the dimension of the $\boldsymbol{y}$ vectors. Comparing the above with the mean squared error (Section 5.2) we immediately see that if σ is fixed, maximizing the above is equivalent (up to an additive and a multiplicative constant that do not change the value of the optimal parameter) to minimizing the training set mean squared error, i.e.,

$$MSE_{\text{train}} = \frac{1}{m} \sum_{i=1}^{m} ||\hat{\boldsymbol{y}}^{(i)} - \boldsymbol{y}^{(i)}||^2.$$

Note that the MSE is an average rather than a sum, which is more practical from a numerical point of view (so you can compare MSEs of sets of different sizes more easily). In practice, researchers reporting log-likelihoods and conditional log-likelihoods also tend to report the per-example average log-likelihood, for the

very same reason. The exponential of the average log-likelihood is also called the *perplexity* and is used in language modeling applications.

Whereas in the case of linear regression we have $\mu(\boldsymbol{x}) = \boldsymbol{w}\cdot\boldsymbol{x}$, the above equally applies to other forms of regression, e.g., with a neural network predicting with $\mu(\boldsymbol{x})$ the expected value of $\mathbf{y}$ given $\boldsymbol{x}$.

5.6.2 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.

The maximum likelihood estimator has the property of consistency (see Sec. 5.5.5 above), i.e., as more training are considered, the estimator converges to the best one in some sense. There are other inductive principles besides the maximum likelihood estimator, many of which share the property of being consistent estimators. However, there is the question of how many training examples one needs to achieve a particular generalization error, or equivalently what estimation error one gets for a given number of training examples, also called *efficiency*. This 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), the maximum likelihood induction principle is often considered the preferred one in machine learning, modulo slight adjustments such as described in the next Section, to better deal with the non-asymptotic case where the number of examples is small enough to yield overfitting behavior.

5.7 Bayesian Statistics

So far we have discussed 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. *Bayesian statistics* provides a natural and theoretically elegant way to carry out this approach.

Historically, statistics has become divided between two communities. One of these communities is known as *frequentist statistics* or *orthodox statistics*. The

other is known as *Bayesian statistics*. The difference is mainly one of world view but can have important practical implications.

As discussed in Sec. 5.5.1, 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 data (which are seen as random).

The Bayesian perspective on statistics is quite different and, in some sense, more intuitive. The Bayesian uses probability to reflect degrees of certainty of states of knowledge. The data 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 prior distribution 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, we 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:

$$p(\boldsymbol{\theta} \mid x^{(1)}, \ldots, x^{(m)}) = \frac{p(x^{(1)}, \ldots, x^{(m)} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta})}{p(x^{(1)}, \ldots, x^{(m)})} \tag{5.25}$$

If the data is at all informative about the value of $\boldsymbol{\theta}$, the *posterior distribution* $p(\boldsymbol{\theta} \mid x^{(1)}, \ldots, x^{(m)})$ will have less entropy (will be more 'peaky') than the prior $p(\boldsymbol{\theta})$.

Relative to maximum likelihood estimation, Bayesian estimation offers two important differences. First, unlike the maximum likelihood point estimate of $\boldsymbol{\theta}$, the Bayesian makes decision with respect to 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

$$p(x^{(m+1)} \mid x^{(1)}, \ldots, x^{(m)}) = \int p(x^{(m+1)} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta} \mid x^{(1)}, \ldots, x^{(m)}) \, d\boldsymbol{\theta} \tag{5.26}$$

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 Sec. 5.5, we discussed how the frequentist statistics addresses the uncertainty in a given point estimator of $\boldsymbol{\theta}$ by evaluating its variance. The variance of the estimator is an assessment of how the estimate might change will alternative samplings of the observed (or training) 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.

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. One important effect of the prior is to actually reduce the uncertainty (or entropy) in the posterior density over $\boldsymbol{\theta}$.

We have already noted that combining the prior, $p(\boldsymbol{\theta})$, with the data likelihood $p(x^{(1)}, \ldots, x^{(m)} \mid \boldsymbol{\theta})$ results in a distribution that is less entropic (more peaky) than the prior. This is just the result of a basic property of probability distributions: Entropy(product of two densities) $\leq$ Entropy(either density). This implies that the posterior density on $\boldsymbol{\theta}$ is also less entropic than the data likelihood alone (when viewed and normalized as a density over $\boldsymbol{\theta}$). The hypothesis space with the Bayesian approach is, to some extent, more constrained than that with an ML approach. Thus we expect a contribution of the prior to be a further reduction in overfitting as compared to ML estimation.

Example: 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 $\boldsymbol{w} \in \mathbb{R}^n$:

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

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:

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

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

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

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

where we will 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 $\boldsymbol{\theta}$ in our prior belief.

For real-valued parameters it is common to use a Gaussian as a prior distribution:

$$p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \boldsymbol{\mu}_0, \Lambda_0) \propto \exp\left(-\frac{1}{2}(\boldsymbol{w} - \boldsymbol{\mu}_0)^\top \Lambda_0^{-1}(\boldsymbol{w} - \boldsymbol{\mu}_0)\right)$$

where $\boldsymbol{\mu}_0$ and Λ_0 are the prior distribution mean vector and covariance matrix (inverse of covariance matrix) respectively.[4]

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

$$\begin{aligned} p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}) &\propto p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w}) p(\boldsymbol{w}) \\ &\propto \exp\left(-\frac{1}{2}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})^\top (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})\right) \exp\left(-\frac{1}{2}(\boldsymbol{w} - \boldsymbol{\mu}_0)^\top \Lambda_0^{-1}(\boldsymbol{w} - \boldsymbol{\mu}_0)\right) \\ &\propto \exp\left(-\frac{1}{2}\left(-2\boldsymbol{y}^\top \boldsymbol{X}\boldsymbol{w} + \boldsymbol{w}^\top \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{w} + \boldsymbol{w}^\top \Lambda_0^{-1}\boldsymbol{w} - 2\boldsymbol{\mu}_0^\top \Lambda_0^{-1}\boldsymbol{w}\right)\right) \end{aligned}$$

We now make the substitutions $\Lambda_m = \left(\boldsymbol{X}^\top \boldsymbol{X} + \Lambda_0^{-1}\right)^{-1}$ and $\boldsymbol{\mu}_m = \Lambda_m\left(\boldsymbol{X}^\top \boldsymbol{y} + \Lambda_0^{-1}\boldsymbol{\mu}_0\right)$ into the derivation of the posterior (and complete the square) to get:

$$p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}) \propto \exp\left(-\frac{1}{2}(\boldsymbol{w} - \boldsymbol{\mu}_m)^\top \Lambda_m^{-1}(\boldsymbol{w} - \boldsymbol{\mu}_m) + \frac{1}{2}\boldsymbol{\mu}_m^\top \Lambda_m^{-1}\boldsymbol{\mu}_m\right) \tag{5.27}$$

$$\propto \exp\left(-\frac{1}{2}(\boldsymbol{w} - \boldsymbol{\mu}_m)^\top \Lambda_m^{-1}(\boldsymbol{w} - \boldsymbol{\mu}_m)\right). \tag{5.28}$$

In the above, we have dropped all terms that do not include the parameter vector $\boldsymbol{w}$. In Eq. 5.28, we recognize that the posterior distribution has the form of a Gaussian distribution with mean vector $\boldsymbol{\mu}_m$ and covariance matrix Λ_m. It is interesting to note that this justifies our dropping all terms unrelated to $\boldsymbol{w}$, since we know that the posterior distribution must be normalized and, as a Gaussian,

[4] Unless there is a reason to assume a particular covariance structure, we typically assume a diagonal covariance matrix $\Lambda_0 = \text{diag}(\boldsymbol{\lambda}_0)$.

we know what that normalization constant must be (where n is the dimension of the input):

$$p(\boldsymbol{w} \mid \boldsymbol{X}^{(\text{train})}, \boldsymbol{y}^{(\text{train})}) = \frac{1}{\sqrt{(2\pi)^n \, |\Lambda_m|}} \exp\left(-\frac{1}{2}(\boldsymbol{w} - \boldsymbol{\mu}_m)^\top \Lambda_m^{-1}(\boldsymbol{w} - \boldsymbol{\mu}_m) \right). \tag{5.29}$$

5.7.1 Maximum *A Posteriori* (MAP) Estimation

While, in principle, we can use the full Bayesian posterior distribution over the parameter $\boldsymbol{\theta}$ as our estimate of this parameter, it is still often desirable to have a single point estimate (for example, most operations involving the Bayesian posterior for most interesting models are intractable and must be heavily approximated). 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 $\boldsymbol{\theta}$).

$$\boldsymbol{\theta}_{\text{MAP}} = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta} \mid \boldsymbol{x}) = \arg\max_{\boldsymbol{\theta}} \log p(\boldsymbol{x} \mid \boldsymbol{\theta}) + \log p(\boldsymbol{\theta}) \tag{5.30}$$

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 discussed above the advantage brought by introducing the influence of the prior on the MAP estimate is to leverage information other than that contained 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.

Example: Regularized Linear Regression We discussed above the Bayesian approach to linear regression. Given a set of m training samples of input output pairs: $(\boldsymbol{X}^{(\text{train})}, \boldsymbol{y}^{(\text{train})})$, we can express the prediction of y over the entire training set as:

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

where prediction is parametrized by the vector $\boldsymbol{w} \in \mathbb{R}^n$.

Recall from Sec. 5.6.1 that the maximum likelihood estimate for the model parameters is given by:

$$\hat{\boldsymbol{w}}_{\text{ML}} = (\boldsymbol{X}^{(\text{train})\top} \boldsymbol{X}^{(\text{train})})^{-1} \boldsymbol{X}^{(\text{train})\top} \boldsymbol{y}^{(\text{train})} \tag{5.31}$$

For the sake of comparison to the maximum likelihood solution, we will make the simplifying assumption that the prior covariance matrix is scalar: $\Lambda_0 = \lambda_0 \boldsymbol{I}$. As mentioned previously, in practice, this is a very common form of prior distribution. We will also assume that $\boldsymbol{\mu}_0 = 0$. This is also a very common assumption in practice and corresponds to acknowledging that *a priori*, we do not know if the features of $\boldsymbol{x}$ have a positive or negative correlation with y. Adding these assumptions, the MAP estimate of the model parameters (corresponding to the mean of the Gaussian posterior density, in Eq. 5.28) becomes:

$$\hat{\boldsymbol{w}}_{\text{MAP}} = \Lambda_m \boldsymbol{X}^{(\text{train})\top} \boldsymbol{y}^{(\text{train})} \tag{5.32}$$

where $\boldsymbol{\mu}_0$ and Λ_0 are the prior mean and covariance respectively and Λ_m is the posterior covariance and is given by:

$$\Lambda_m = \left(\boldsymbol{X}^{(\text{train})\top} \boldsymbol{X}^{(\text{train})} + \lambda_0^{-1} \boldsymbol{I} \right)^{-1} \tag{5.33}$$

Comparing Eqs. 5.31 and 5.32, we see that the MAP estimate amounts to a weighted combination of the prior maximum probability value, $\boldsymbol{\mu}_0$, and the ML estimate. As the variance of the prior distribution tends to infinity, the MAP estimate reduces to the ML estimate. As the variance of the prior tends to zero, the MAP estimate tends to zero (actually it tends to $\boldsymbol{\mu}_0$ which here is assumed to be zero).

We can make the model capacity tradeoff between the ML estimate and the MAP estimate more explicit by analyzing the bias and variance of these estimates.

It is relatively easy to show that the ML estimate is unbiased, i.e. that $\mathbb{E}[\hat{\boldsymbol{w}}_{\text{ML}}] = \boldsymbol{w}$ and that it has a variance given by:

$$\text{Var}(\boldsymbol{w}_{\text{ML}}) = (\boldsymbol{X}^{(\text{train})\top} \boldsymbol{X}^{(\text{train})})^{-1} \tag{5.34}$$

In order to derive the bias of the MAP estimate, we need to calculate the expectation:

$$\begin{aligned} E[\hat{\boldsymbol{w}}_{\text{MAP}}] &= \mathbb{E}[\Lambda_m \boldsymbol{X}^{(\text{train})\top} \boldsymbol{y}^{(\text{train})}] \\ &= E\left[\Lambda_m \boldsymbol{X}^{(\text{train})\top} \left(\boldsymbol{X}^{(\text{train})} \boldsymbol{w} + \epsilon \right) \right] \\ &= \Lambda_m \quad \boldsymbol{X}^{(\text{train})\top} \boldsymbol{X}^{(\text{train})} \boldsymbol{w} \quad + \Lambda_m \boldsymbol{X}^{(\text{train})\top} E[\epsilon] \\ &= \quad \left(\boldsymbol{X}^{(\text{train})\top} \boldsymbol{X}^{(\text{train})} + \lambda_0^{-1} \boldsymbol{I} \right)^{-1} \boldsymbol{X}^{(\text{train})\top} \boldsymbol{X}^{(\text{train})} \boldsymbol{w}, \end{aligned} \tag{5.35}$$

We see that while the expected value of the ML estimate is the true parameter value $\boldsymbol{w}$ (i.e. the parameters that we assume generated the data); the expected

value of the MAP estimate is a weighted average of $\boldsymbol{w}$ and the prior mean $\boldsymbol{\mu}$. We compute the bias as:

$$\begin{aligned}\text{Bias}(\hat{\boldsymbol{w}}_{\text{MAP}}) &= E[\hat{\boldsymbol{w}}_{\text{MAP}}] - \boldsymbol{w} \\ &= -\left(\lambda_0 \boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})} + \boldsymbol{I}\right)^{-1}\boldsymbol{w}.\end{aligned}$$

Since the bias is not zero, we can conclude that the MAP estimate is biased, and as expected we can see that as the variance of the prior $\lambda_0 \to \infty$, the bias tends to zero. As the variance of the prior $\lambda_0 \to \mathbf{0}$, the bias tends to $\boldsymbol{w}$.

In order to compute the variance, we use the identity $\text{Var}(\hat{\theta}) = \mathbb{E}\left[\hat{\theta}^2\right] - \mathbb{E}\left[\hat{\theta}\right]^2$. So before computing the variance we need to compute $\mathbb{E}\left[\hat{\boldsymbol{w}}_{\text{MAP}}\hat{\boldsymbol{w}}_{MAP}^{\top}\right]$:

$$\begin{aligned}\mathbb{E}\left[\hat{\boldsymbol{w}}_{\text{MAP}}\hat{\boldsymbol{w}}_{MAP}^{\top}\right] &= \mathbb{E}\Big[\Lambda_m \boldsymbol{X}^{(\text{train})\top}\hat{\boldsymbol{y}}^{(\text{train})}\boldsymbol{y}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\Lambda_m\Big] \\ &= \mathbb{E}\Big[\Lambda_m \boldsymbol{X}^{(\text{train})\top}\left(\boldsymbol{X}^{(\text{train})}\boldsymbol{w} + \boldsymbol{\epsilon}\right)\left(\boldsymbol{X}^{(\text{train})}\boldsymbol{w} + \boldsymbol{\epsilon}\right)^{\top}\boldsymbol{X}^{(\text{train})}\Lambda_m\Big] \\ &= \Lambda_m \boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\boldsymbol{w}\boldsymbol{w}^{\top}\boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\Lambda_m \\ &\quad + \Lambda_m \boldsymbol{X}^{(\text{train})\top}\mathbb{E}\left[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^{\top}\right]\boldsymbol{X}^{(\text{train})}\Lambda_m \\ &= \Lambda_m \boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\boldsymbol{w}\boldsymbol{w}^{\top}\boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\Lambda_m \\ &\quad + \Lambda_m \boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\Lambda_m \\ &= E[\hat{\boldsymbol{w}}_{\text{MAP}}]E[\hat{\boldsymbol{w}}_{\text{MAP}}]^{\top} + \boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\Lambda_m\end{aligned}$$

With $\mathbb{E}\left[\hat{\boldsymbol{w}}_{\text{MAP}}\hat{\boldsymbol{w}}_{MAP}^{\top}\right]$ thus computed, the variance of the MAP estimate of our linear regression model is given by:

$$\begin{aligned}\text{Var}(\hat{\boldsymbol{w}}_{\text{MAP}}) &= \mathbb{E}\left[\hat{\boldsymbol{w}}_{MAP}\hat{\boldsymbol{w}}_{MAP}^{\top}\right] - \mathbb{E}\left[\hat{\boldsymbol{w}}_{MAP}\right]\mathbb{E}\left[\hat{\boldsymbol{w}}_{MAP}^{\top}\right] \\ &= E[\hat{\boldsymbol{w}}_{\text{MAP}}]E[\hat{\boldsymbol{w}}_{\text{MAP}}]^{\top} + \Lambda_m \boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\Lambda_m - E[\hat{\boldsymbol{w}}_{\text{MAP}}]E[\hat{\boldsymbol{w}}_{\text{MAP}}]^{\top} \\ &= \Lambda_m \boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})}\Lambda_m \\ &= \left(\boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})} + \lambda_0^{-1}\boldsymbol{I}\right)^{-1}\boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})} \\ &\qquad \times \left(\boldsymbol{X}^{(\text{train})\top}\boldsymbol{X}^{(\text{train})} + \lambda_0^{-1}\boldsymbol{I}\right)^{-1} \qquad (5.36)\end{aligned}$$

It is perhaps difficult to compare Eqs. 5.34 and 5.36. But if we assume that $\boldsymbol{w}$ is one-dimensional (along with $\boldsymbol{x}$), it becomes a bit easier to see that, as long as λ_0 is bounded, then $\text{Var}(\hat{w}_{\text{ML}}) = \frac{1}{\sum_{i=1}^{m} x_i^2} > \text{Var}(\hat{w}_{\text{MAP}}) = \frac{\lambda_0 \sum_{i=1}^{m} x_i^2}{\left(1+\lambda_0 \sum_{i=1}^{m} x_i^2\right)^2}$.

From the above analysis we can see that the role of the prior in the MAP estimate is to trade increased bias for a reduction in variance. The goal, of

course, is to try to avoid overfitting. The incurred bias is a consequence of the reduction in model capacity caused by limiting the space of hypotheses to those with significant probability density under the prior.

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 such regularizer terms 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.

5.8 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.

5.8.1 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 conditional likelihood estimation (Sec. 5.6.1, also just called maximum likelihood for short) 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 $p(y \mid \boldsymbol{x}; \boldsymbol{\theta}) = \mathcal{N}(y \mid \boldsymbol{\theta}^\top \boldsymbol{x}, \boldsymbol{I})$. 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 parameterized 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:

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

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 probability of conditional distributions over the right kind of input and output variables.

5.8.2 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.

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

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

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 $k(\boldsymbol{x}, \boldsymbol{x}^{(i)}) = \phi(\boldsymbol{x})^\top \phi(\boldsymbol{x}^{(i)})$ called a *kernel*.

We can then make predictions using the function

$$f(x) = b + \sum_i \alpha_i k(\boldsymbol{x}, \boldsymbol{x}^{(i)}). \tag{5.37}$$

This function is linear in the space that ϕ maps to, but non-linear as a function of $\boldsymbol{x}$.

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

are guaranteed to converge efficiently. This is only possible because we consider ϕ fixed and only optimize $\boldsymbol{\alpha}$, i.e., the optimization algorithm can view the decision function as being linear in a different space. Second, the kernel function k need not be implemented in terms of explicitly applying the ϕ mapping and then applying the dot product. The dot product in ϕ space might be equivalent to a non-linear but computationally less expensive operation in $\boldsymbol{x}$ space. For example, we could design an *infinite-dimensional* feature mapping $\phi(\boldsymbol{x})$ over the non-negative integers. Suppose that this mapping returns a vector containing $\boldsymbol{x}$ ones followed by infinitely many zeros. Explicitly constructing this mapping, or taking the dot product between two such vectors, costs infinite time and memory. But we can write a kernel function $k(x, x^{(i)}) = \min(x, x^{(i)})$ that is exactly equivalent to to this infinite-dimensional dot product. The most commonly used kernel is the *Gaussian kernel*

$$k(\boldsymbol{u}, \boldsymbol{v}) = \mathcal{N}(\boldsymbol{u} - \boldsymbol{v}; 0, \sigma^2 I) \tag{5.38}$$

where $\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$ is the standard normal density. This kernel corresponds to the dot product $k(\boldsymbol{u}, \boldsymbol{v}) = \phi(\boldsymbol{x})^\top \phi(\boldsymbol{x})$ on an infinite-dimensional feature space ϕ and also has an interpretation as a similarity function, acting like a kind of template matching.

Support vector machines are not the only algorithm that can be enhanced using the kernel trick. Many linear models can be enhanced in this way. This category of algorithms is known as *kernel machines* or *kernel methods*.

A major drawback to kernel machines is that the cost of learning the $\boldsymbol{\alpha}$ coefficients is quadratic in the number of training examples. A related problem 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*. Another major drawback of common kernel machines (such as those using the Gaussian kernel) is more statistical and regards their difficulty in generalizing to complex variations far from the training examples, as discussed in Section 5.12.

The analysis of the statistical limitations of support vector machines with general purpose kernels like the Gaussian kernels actually motivated the rebirth of neural networks through deep learning. Support vector machines and other kernel machines have often been viewed as a competitor to deep learning (though some deep networks can in fact be interpreted as support vector machines with learned kernels). The current deep learning renaissance began when deep networks were shown to outperform support vector machines on the MNIST benchmark dataset (Hinton *et al.*, 2006). One of the main reasons for the current

popularity of deep learning relative to support vector machines is the fact that the cost of training kernel machines usually scales quadratically with the number of examples in the training set. For a deep network of fixed size, the memory cost of training is constant with respect to training set size (except for the memory needed to store the examples themselves) and the runtime of a single pass through the training set is linear in training set size. These asymptotic results meant that kernelized SVMs dominated while datasets were small, but deep models currently dominate now that datasets are large.

5.8.3 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, there are no parameters. 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 esentially 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 neighbors has unlimited capacity and will eventually converge to the Bayes error given a large enough training set if k is properly reduced as the number of examples is increased. However, it may perform very badly on small, finite training sets. 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 $\boldsymbol{x} \in \mathbb{R}^1 00$ 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 $x_1 00$, not by the lone feature x_1. Thus the output on small training sets will essentially be random.

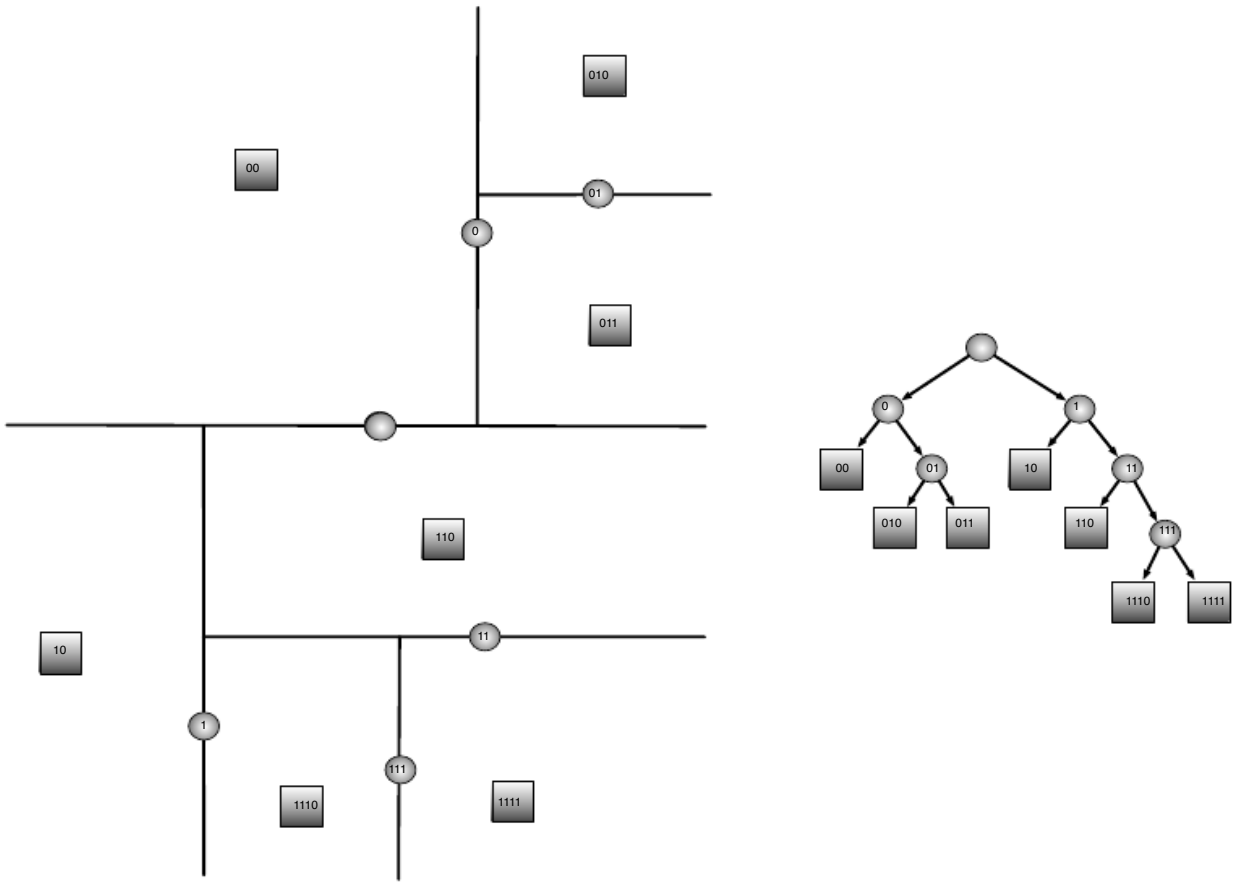

Figure 5.7: Decision tree (right) and how it cuts the input space into regions, with a constant output in each region (left). Each node of the tree (circle or square) is associated with a region (the entire space for the root node, with the empty string identifier). Internal nodes (circles) split their region in two, via an axis-aligned cut (occasionally, some decision trees use more complicated cuts than shown in this example). Leaf nodes (squares) are associated with a model output, typically set to the average target output for the training examples that fall in the corresponding region. 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). Note that the result is a piecewise-constant function, and note how the number of regions (pieces) cannot be greater than the number of examples, hence it is not possible 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 Fig. 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 usally 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. Note that 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.

5.9 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.

Learning a representation of data 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 x as possible while obeying some penalty or constraint aimed at keeping the representation *simpler* or more accessible than x itself.

There are multiple ways of defining a *simpler* representation, some 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 generally embed the dataset into a high-dimensional representation[5] where the number of non-zero entries is small. 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. Chapter 16 discusses some of the qualities we would like in our learned representations, along with specific representation learning algorithms more powerful than the simple one presented next, Principal Components Analysis.

5.9.1 Principal Components Analysis

In the remainder of this section we will consider one of the most widely used unsupervised learning methods: Principle Components Analysis (PCA). PCA is an orthogonal, linear transformation of the data that projects it into a representation where the elements are uncorrelated (shown in Figure 5.8).

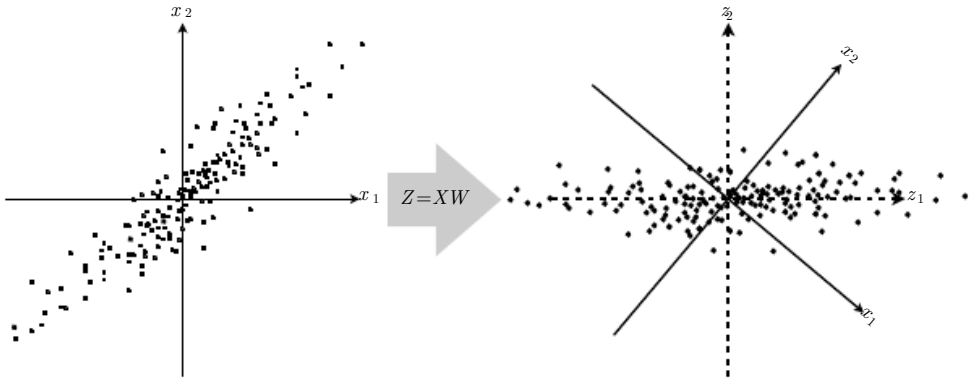

Figure 5.8: Illustration of the data representation learned via PCA.

In section 2.12, we saw that we could learn a one-dimensional representation

[5]sparse representations often use over-complete representations: the representation dimension is greater than the original dimensionality of the data.

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 take a look at other properties of the PCA representation. Specifically, we will study how the PCA representation can be said to decorrelate the original data representation $\boldsymbol{X}$.

Let us consider the $n \times m$-dimensional design matrix $\boldsymbol{X}$. We will assume that the data has a mean of zero, $\mathbb{E}[\boldsymbol{x}] = \mathbf{0}$. If this is not the case, the data can easily be centered (mean removed). The unbiased sample covariance matrix associated with $\boldsymbol{X}$ is given by:

$$\text{Var}[\boldsymbol{x}] = \frac{1}{n-1}\boldsymbol{X}^\top \boldsymbol{X} \tag{5.39}$$

One important aspect of PCA is that it finds a representation (through linear transformation) $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x}$ where $\text{Var}[\boldsymbol{z}]$ is diagonal. To do this, we will make use of the singular value decomposition (SVD) of $\boldsymbol{X}$: $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top$, where $\boldsymbol{\Sigma}$ is an $n \times m$-dimensional rectangular diagonal matrix with the singular values of $\boldsymbol{X}$ on the main diagonal, $\boldsymbol{U}$ is an $n \times n$ matrix whose columns are orthonormal (i.e. unit length and orthogonal) and $\boldsymbol{W}$ is an $m \times m$ matrix also composed of orthonormal column vectors.

Using the SVD of $\boldsymbol{X}$, we can re-express the variance of $\boldsymbol{X}$ as:

$$\begin{aligned}
\text{Var}[\boldsymbol{x}] &= \frac{1}{n-1}\boldsymbol{X}^\top \boldsymbol{X} && (5.40)\\
&= \frac{1}{n-1}(\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top)^\top \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top && (5.41)\\
&= \frac{1}{n-1}\boldsymbol{W}\boldsymbol{\Sigma}^\top \boldsymbol{U}^\top \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top && (5.42)\\
&= \frac{1}{n-1}\boldsymbol{W}\boldsymbol{\Sigma}^2\boldsymbol{W}^\top, && (5.43)
\end{aligned}$$

where we use the orthonormality of $\boldsymbol{U}$ ($\boldsymbol{U}^\top\boldsymbol{U} = \boldsymbol{I}$) and define $\boldsymbol{\Sigma}^2$ as an $m \times m$-dimensional diagonal matrix with the squares of the singular values of $\boldsymbol{X}$ on the diagonal, i.e. the ith diagonal elements is given by $\Sigma^2_{i,i}$. This shows that if we

take $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x}$, we can ensure that the covariance of $\boldsymbol{z}$ is diagonal as required.

$$\text{Var}[\boldsymbol{z}] = \frac{1}{n-1}\boldsymbol{Z}^\top \boldsymbol{Z} \tag{5.44}$$

$$= \frac{1}{n-1}\boldsymbol{W}^\top \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{W} \tag{5.45}$$

$$= \frac{1}{n-1}\boldsymbol{W}\boldsymbol{W}^\top \boldsymbol{\Sigma}^2 \boldsymbol{W}\boldsymbol{W}^\top \tag{5.46}$$

$$= \frac{1}{n-1}\boldsymbol{\Sigma}^2 \tag{5.47}$$

Similar to our analysis of the variance of $\boldsymbol{X}$ above, we exploit the orthonormality of $\boldsymbol{W}$ (i.e., $\boldsymbol{W}^\top \boldsymbol{W} = \boldsymbol{I}$). Our use of SVD to solve for the PCA components of $\boldsymbol{X}$ (i.e. elements of $\boldsymbol{z}$) reveals an interesting connection to the eigen-decomposition of a matrix related to $\boldsymbol{X}$. Specifically, the columns of $\boldsymbol{W}$ are the eigenvectors of the $n \times n$-dimensional matrix $\boldsymbol{X}^\top \boldsymbol{X}$.

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 attempt 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 (mediated via the transformation $\boldsymbol{W}$) that aligns the principal axes of variance with the basis of the new representation space associated with $\boldsymbol{z}$, as illustrated in Fig. 5.8. 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. These issues are discussed below in Sec. 5.12 and later in detail in Chapter 16.

5.10 Weakly Supervised Learning

Weakly supervised learning is another class of learning methods that stands between supervised and unsupervised learning. It refers to a setting where the datasets consists of $(\boldsymbol{x}, y)$ pairs, as in supervised learning, but where the labels y are either unreliably present (i.e. with missing values) or noisy (i.e. where the label given is not the true label).

Methods for working with weakly labeled data have recently grown in importance due to the—largely untapped—potential for using large quantities of readily

available weakly labeled data in a transfer learning paradigm to help solve problems where large, clean datasets are hard to come-by. The Internet has become a major source of this kind of noisy data.

For example, although we would like to train a computer vision system with labels indicating the presence and location of every object (and which pixels correspond to which object) in every image, such labeling is very human-labor intensive. Instead, we want to take advantage of images for which only the main object is identified, like the ImageNet dataset (Deng *et al.*, 2009), or worse, of video for which some general and high-level semantic spoken caption is approximately temporally aligned with the corresponding frames of the video, like the DVS data (Descriptive Video service) which has recently been released (Torabi *et al.*, 2015).

5.11 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

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

and the model specification $p_{\text{model}}(y \mid \boldsymbol{x}) = \mathcal{N}(y \mid \boldsymbol{x}^\top \boldsymbol{w} + b, 1)$. Typically we then observe that J simplifies to the mean squared error and we can choose to optimize this in closed form by solving the normal equations with the Moore-Penrose pseudo-inverse.

By realizing that we can modify any of these components, 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 maxmimum likelihood estimation. This main term of the cost function 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

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

where y is the per-example loss $L(\boldsymbol{x}, y, \boldsymbol{\theta}) = \log p(y \mid \boldsymbol{x}; \boldsymbol{\theta})$. We can design many different cost functions just by taking the expectation across the training set of different per-example loss functions.

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

$$J(\boldsymbol{w}, b) = \lambda ||\boldsymbol{w}||_2^2 - \mathbb{E}_{\boldsymbol{x}, y \sim p_{\text{data}}} p_{\text{model}} \left(y \mid \boldsymbol{x} \right).$$

This still allows closed-form optimization.

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

This recipe 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

$$J(\boldsymbol{w}) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}} ||\boldsymbol{x} - r(x; \boldsymbol{w})||_2^2$$

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

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.

5.12 The Curse of Dimensionality and Statistical Limitations of Local Generalization

The number of variable configurations grows exponentially with the number of variables, i.e., with dimension, which brings up a statistical form of the *curse of dimensionality*, introduced in the next section. Many non-parametric learning algorithms, such as kernel machines with a Gaussian kernel, rely on a simple preference over functions which corresponds to an assumption of smoothness or local

constancy. As argued in Section 5.12.2 that follows, this allows these algorithms to generalize near the training examples, but does not allow them to generalize in a non-trivial way far from them: the number of ups and downs that can be captured is limited by the number of training examples. This is particularly problematic with high-dimensional data, because of the curse of dimensionality. In order to reduce that difficulty, researchers have introduced the idea of dimensionality reduction and manifold learning, introduced in Section 5.12.3. This motivates the introduction of additional knowledge, i.e., a priori information, about the task to be learned, as well as the idea of learning to better represent the data, the topic which constitutes the bulk of the rest of this book.

5.12.1 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 the variables of interest increases exponentially as the dimensionality increases.

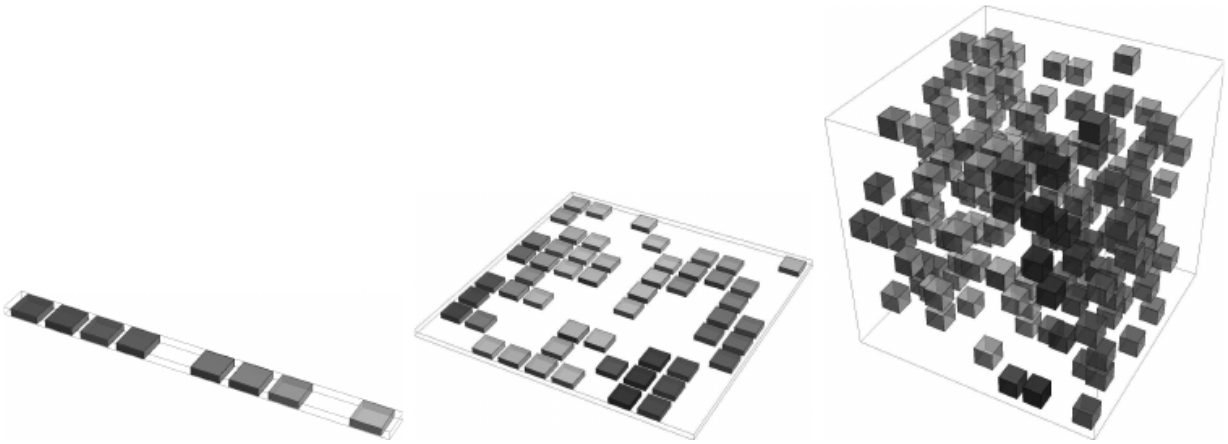

Figure 5.9: As the number of relevant dimensions of the data increases (from left to right), the number of configurations of interest may grow exponentially. In the figure we first consider one-dimensional data (left), i.e., one variable for which we only care to distinguish 10 regions of interest. With enough examples falling within each of these regions (cells, in the figure), learning algorithms can easily generalize correctly, i.e., estimate the value of the target function within each region (and possibly interpolate between neighboring regions). With 2 dimensions (center), but still caring 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. With 3 dimensions (right) 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, it looks like we need $O(V^d)$ regions and examples. This is an instance of the curse of dimensionality. However, note that if the data distribution is concentrated on a smaller set of regions, we may actually not need to cover all the possible regions, only those where probability is non-negligible. *Figure graciously provided by, and with authorization from, Nicolas Chapados.*

The curse of dimensionality rears its ugly head 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 the variables of interest 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. In low dimensions we can describe this space with a low number of grid cells that are mostly occupied by the data. The least we can assume about the data generating distribution is that our learner should provide the same answer to two examples falling in the same grid cell. It is a form of local constancy assumption, a notion that we develop further in the next section. 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 going to be huge, much larger than our number of examples, most configurations will have no training example associated with it. How could we possibly say something meaningful about these new configurations? A simple answer is to extend the local constancy assumption into a smoothness assumption, as explained next.

5.12.2 Smoothness and Local Constancy A Priori Preference

As argued previously, and especially in high-dimensional spaces (because of the curse of dimensionality introduced above), machine learning algorithms need priors, i.e., a preference over the space of solutions, in order to generalize to new configurations not seen in the training set. The specification of these preferences includes the choice of model family, as well as any regularizer or other aspects of the algorithm that influence the final outcome of training. We consider here a particular family of preferences which underlie many classical machine learning algorithms, and which we call the *smoothness prior* or the *local constancy prior*. We find that when the function to be learned has many ups and downs, and this is typically the case in high-dimensional spaces because of the curse of dimensionality (see above), then the smoothness prior is insufficient to achieve good generalization. We argue that more assumptions are needed in order to generalize better, in this setting. Deep learning algorithms typically introduce such additional assumptions. This starts with the classical multi-layer neural networks studied in the next chapter (Chapter 6), and in Chapter 16 we return to the advantages that representation learning, distributed representations and depth can bring towards generalization, even in high-dimensional spaces.

Different smoothness or local constancy priors can be expressed, but what they basically say is that the target function or distribution of interest f^* is such that

$$f^*(\boldsymbol{x}) \approx f^*(\boldsymbol{x} + \epsilon) \tag{5.48}$$

for most configurations $\boldsymbol{x}$ and small change ϵ. In other words, if we know a good answer (e.g., for an example $\boldsymbol{x}$) then that answer is probably good in the neighborhood of $\boldsymbol{x}$, and if we have several good answers in some neighborhood we would combine them (e.g., by some form of averaging or interpolation) to produce an answer that agrees with 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 R containing all the points $\boldsymbol{x}$ that have the same set of k nearest neighbors

in the training set. Note that for $k = 1$, the number of distinguishable regions cannot be more than the number of training examples.

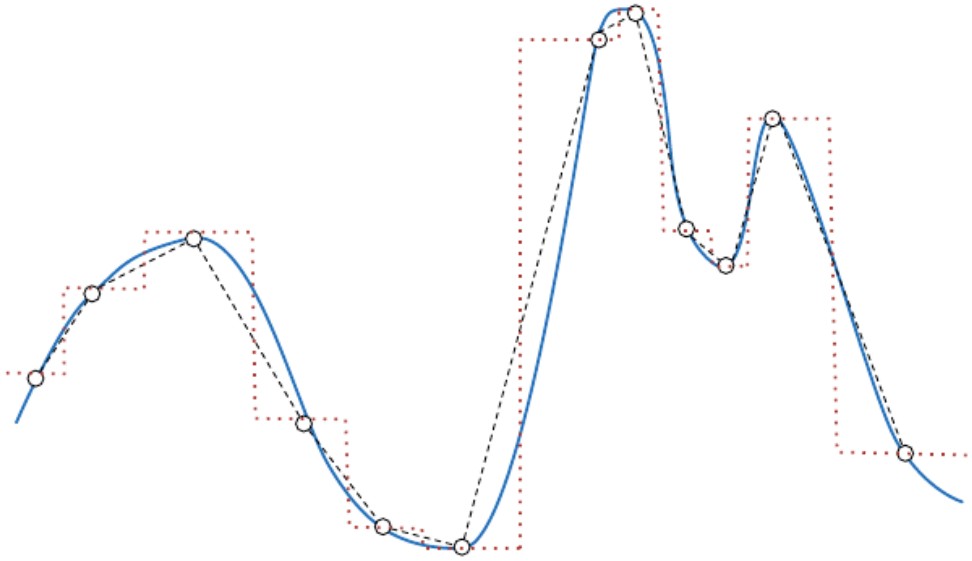

Figure 5.10: Illustration of interpolation and kernel-based methods, which construct a smooth function by interpolating in various ways between the training examples (circles), which act like knot points controlling the shape of the implicit regions that separate them as well as the values to output within each region. Depending on the type of kernel, one obtains a piecewise constant (histogram-like, in dotted red), a piecewise linear (dashed black) or a smoother kernel (bold blue). The underlying assumption is that the target function is as smooth or locally as constant as possible. This assumption allows to *generalize locally*, i.e., to extend the answer known at some point $\boldsymbol{x}$ to nearby points, and this works very well so long as, like in the figure, there are enough examples to cover most of the ups and downs of the target function.

To obtain even more smoothness, we can *interpolate* between neighboring training examples, as illustrated in Figure 5.10. For example, *non-parametric kernel density estimation methods* and *kernel regression* methods construct a learned function f of the form of Eq. 5.37 for classification or regression, or alternatively, e.g., in the Parzen regression estimator, of the form

$$f(\boldsymbol{x}) = b + \sum_{i=1}^{n} \alpha_i \frac{k(\boldsymbol{x}, \boldsymbol{x}^{(i)})}{\sum_{j=1}^{n} k(\boldsymbol{x}, x^{(j)})}.$$

If the kernel function k is discrete (e.g. 0 or 1), then this can include the above cases where f is piecewise constant and a discrete set of regions (no more than one per training example) can be distinguished. However, better results can often be obtained if k is smooth, e.g., the Gaussian kernel from Eq. 5.38. With k a *local kernel* (Bengio *et al.*, 2006b; Bengio and LeCun, 2007b; Bengio, 2009)[6] , we can

[6]i.e., with $k(\boldsymbol{u}, \boldsymbol{v})$ large when $\boldsymbol{u} = \boldsymbol{v}$ and decreasing as they get farther apart

think of each $\boldsymbol{x}^{(i)}$ as a *template* and the kernel function as a *similarity function* that *matches a template and a test example*.

With the Gaussian kernel, we do not have a piecewise constant function but instead a continuous and smooth function. In fact, the choice of k can be shown to correspond to a particular form of smoothness. Equivalently, we can think of many of these estimators as the result of smoothing the *empirical distribution* by convolving it with a function associated with the kernel, e.g., the Gaussian kernel density estimator is the empirical distribution convolved with the Gaussian density.

Although in classical non-parametric estimators the α_i of Eq. 5.37 are fixed (e.g. to $1/n$ for density estimation and to $y^{(i)}$ for supervised learning from examples $(\boldsymbol{x}^{(i)}, y^{(i)})$), they can be optimized, and this is the basis of more modern non-parametric kernel methods (Schölkopf and Smola, 2002) such as the Support Vector Machine (Boser *et al.*, 1992; Cortes and Vapnik, 1995) (see also Section 5.8.2).

However, as illustrated in Figure 5.10, even though these smooth kernel methods generalize better, the main thing that has changed is that one can basically interpolate between the neighboring examples, in some space associated with the kernel. One can then think of the training examples as control knots which locally specify the shape of each region and the associated output.

Decision trees also suffer from the limitations of exclusively smoothness-based learning. Because each example only informs the region in which it falls about which output to produce, *one cannot have more regions than training examples*. If the target function can be well approximated by cutting the input space into N regions (with a different answer in each region), then at least N examples are needed (and a multiple of N is needed to achieve some level of statistical confidence in the predicted output). All this is also true if the tree is used for density estimation (the output is simply an estimate of the density within the region, which can be obtained by the ratio of the number of training examples in the region by the region volume) or whether a non-constant (e.g. linear) predictor is associated with each leaf (then more examples are needed within each leaf node, but the relationship between number of regions and number of examples remains linear). We examine below how this may hurt the generalization ability of decision trees and other learning algorithms that are based only on the smoothness or local constancy priors, when the input is high-dimensional, i.e., because of the curse of dimensionality.

In all cases, the smoothness assumption (Eq. 5.48) allows the learner to *generalize locally*. Since we assume that the target function obeys $f^*(\boldsymbol{x}) \approx f^*(\boldsymbol{x} + \epsilon)$ most of the time for small ϵ, we can generalize the empirical distribution (or the $(\boldsymbol{x}, y)$ training pairs) to the neighborhood of the training examples. If $(\boldsymbol{x}^{(i)}, y^{(i)})$

is a supervised (input,target) training example, then we expect $f^*(\boldsymbol{x}^{(i)}) \approx y^{(i)}$, and therefore if $\boldsymbol{x}$ is a near neighbor of $\boldsymbol{x}^{(i)}$, we expect that $f^*(\boldsymbol{x}) \approx y^{(i)}$. By considering more neighbors, we can obtain better generalization, by better executing the smoothness assumption.

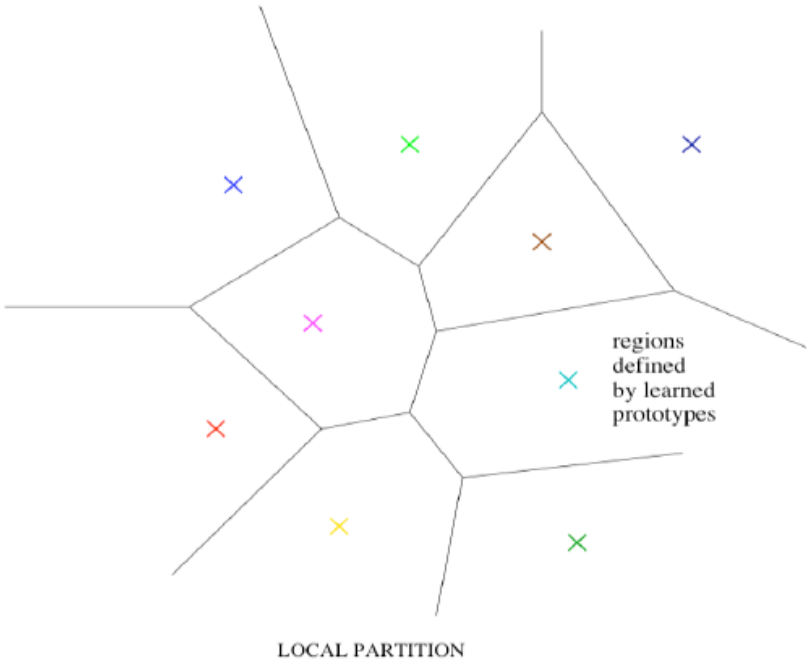

Figure 5.11: Illustration of how non-parametric learning algorithms that exploit only the smoothness or local constancy priors typically break up the input space into regions, with examples in those regions being used both to define the region boundaries and what the output should be within each region. The figure shows the case of clustering or 1-nearest-neighbor classifiers, for which each training example (cross of a different color) defines a region or a template (here, the different regions form a Voronoi tessellation). The number of these contiguous regions cannot grow faster than the number of training examples. In the case of a decision tree, the regions are recursively obtained by axis-aligned cuts within existing regions, but for these and for kernel machines with a local kernel (such as the Gaussian kernel), the same property holds, and generalization can only be *local*: each training example only informs the learner about how to generalize in some neighborhood around it.

In general, to distinguish $O(N)$ regions in input space, all of these methods require $O(N)$ examples (and typically there are $O(N)$ parameters associated with the $O(N)$ regions). This is illustrated in Figure 5.11 in the case of a nearest-neighbor or clustering scenario, where each training example can be used to define one region. 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,

i.e., with a lot of variations, but a simple structure to them, and imagine that the number of training examples is substantially less than the number of black and white regions. Based on local generalization and the smoothness or local constancy prior, we could get the correct answer within a constant-colour region, but we could not correctly predict the checkerboard pattern. The only thing that an example tells us, with this prior, is that nearby points should have the same colour, and the only way to get the checkerboard right is to cover all 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 to cover most of the ups and downs of the target function.* This is generally true when the function to be learned is smooth enough, which is typically the case for low-dimensional data. And if it is not very smooth (we want to distinguish a huge number of regions compared to the number of examples), is there any hope to generalize well?

Both of these questions are answered positively in Chapter 16. The key insight is that a very large number of regions, e.g., $O(2^N)$, can be defined with $O(N)$ examples, so long as we introduce some dependencies between the regions via additional priors about the underlying data generating distribution. In this way, we can actually generalize non-locally (Bengio and Monperrus, 2005; Bengio *et al.*, 2006c). A neural network can actually learn a checkerboard pattern. Similarly, some recurrent neural networks can learn the n-bit parity (at least for some not too large values of n). Of course we could also solve the checkerboard task by making a much stronger assumption, e.g., that the target function is periodic. However, neural networks can generalize to a much wider variety of structures, and indeed our AI tasks have structure that is much too complex to be limited to 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. 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, as discussed in Chapter 16. Priors that are based on compositionality, such as arising from learning distributed representations and from a deep composition of representations, can give an exponential advantage, which can hopefully counter the exponential curse of dimensionality. Chapter 16 discusses these questions from the angle of representation learning and the objective of *disentangling the underlying factors of variation.*

5.12.3 Manifold Learning and the Curse of Dimensionality

We consider here a particular type of machine learning task called *manifold learning*. Although they have been introduced to reduce the curse of dimensionality. We will argue that they allow one to visualize and highlight how the smoothness prior is not sufficient to generalize in high-dimensional spaces. Chapter 17 is devoted to the manifold perspective on representation learning and goes in much greater details in this topic as well as in actual manifold learning algorithms based on neural networks.

A *manifold* is a connected region, i.e., a set of points, associated with a neighborhood around each point, which makes it locally look like a Euclidean space. The notion of neighbor implies the existence of transformations that can be applied to *move on the manifold* from one position to a neighboring one. Although there is a formal mathematical meaning to this term, in machine learning it tends to be used more loosely to talk about a connected set of points that can be well approximated 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, i.e., moving along the manifold in some direction. The manifolds we talk about in machine learning are subsets of points, also called a submanifold, of the embedding space (which is also a manifold).

Manifold learning algorithms assume that the data distribution is concentrated in a small number of dimensions, i.e., that the set of high-probability configurations can be approximated by a low-dimensional manifold. Figure 5.8 (left) illustrates a distribution that is concentrated near a linear manifold (the manifold is along a 1-dimensional straight line). 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.

Is this assumption reasonable? It seems to be true for almost all of the AI tasks such as those involving images, sounds and text. To be convinced of this we will invoke (a) the observation that probability mass is concentrated and (b) the observed objects can generally be transformed into other plausible configurations via some small changes (which indicates a notion of direction of variation while staying on the "manifold"). For (a), consider that if the assumption of probability concentration was false, then sampling uniformly at random from in the set of all configurations (e.g., uniformly in $\mathbb{R}^n$) should produce probable (data-like) configurations reasonably often. But this is not what we observe in practice. For example, generate pixel configurations for an image by independently picking the grey level (or a binary 0 vs 1) for each pixel. What kind of images do you get? You get "white noise" images, that look like the old television sets when no signal

is coming in, as illustrated in Figure 5.12 (left). What is the probability that you would obtain something that looks like a natural image, with this procedure? Almost zero, because the set of probable configurations (near the manifold of natural images) occupies a very small volume out of the total set of pixel configurations. Similarly, if you generate a document by picking letters randomly, 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.

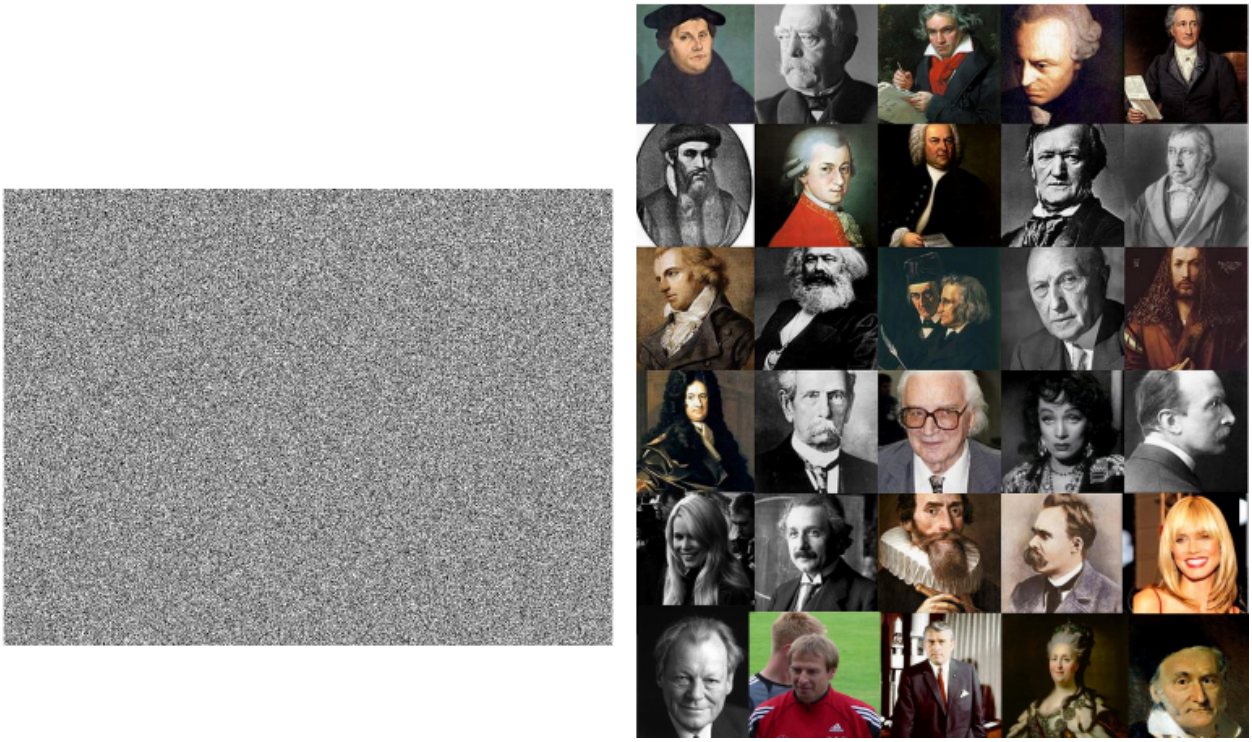

Figure 5.12: Sampling images uniformly at random, e.g., by randomly picking each pixel according to a uniform distribution, gives rise to white noise images such as illustrated on the left. Although there is a non-zero probability to generate something that looks like a natural image (like those on the right), that probability is exponentially tiny (exponential in the number of pixels!). This suggests that natural images are very "special", and that they occupy a tiny volume of the space of images.

The above thought experiments, which are in agreement with the many experimental results of the manifold learning literature, e.g. (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 establish that for a large class of datasets of interest in AI, the *manifold hypothesis* is true: the data generating distribution concentrates in a small number of dimensions, as in the cartoon of Figure 17.4,

from Chapter 17. That chapter explores the relationships between representation learning and manifold learning: if the data distribution concentrates on a smaller number of dimensions, then we can think of these dimensions as natural coordinates for the data, and we can think of representation learning algorithms as ways to map the input space to a new and often lower-dimensional space which captures the leading dimensions of variation present in the data as axes or dimensions of the representation.

An initial hope of early work on manifold learning (Roweis and Saul, 2000; Tenenbaum *et al.*, 2000) was to *reduce the effect of the curse of dimensionality*, by first reducing the data to a lower dimensional representation (e.g. mapping (x_1, x_2) to z_1 in Figure 5.8 (right)), and then applying ordinary machine learning in that transformed space. This dimensionality reduction can be achieved by learning a transformation (generally non-linear, unlike with PCA introduced in Section 5.9.1) of the data that is *invertible for most training examples*, i.e., that keeps the information in the input example. It is only possible to reconstruct input examples from their low-dimensional representation because they lie on a lower-dimensional manifold, of course. This is basically how *auto-encoders* (Chapter 15) are trained.

The hope was that by non-linearly projecting the data in a new space of lower dimension, we would reduce the curse of dimensionality by only looking at relevant dimensions, i.e., a smaller set of regions of interest (cells, in Figure 5.9). This can indeed be the case, however, as discussed in Chapter 17, the manifolds can be highly curved and have a very large number of twists, requiring still a very large number of regions to be distinguished (every up and down of each corner of the manifold). And even if we were to reduce the dimensionality of an input from 10000 (e.g. 100×100 binary pixels) to 100, 2^{100} is still too large to hope covering with a training set. This still rules out the use of purely local generalization (i.e., the smoothness prior only) to model such manifolds, as discussed in Chapter 17 around Figure 17.4 and 17.5. It may also be that although the effective dimensionality of the data could be small, some examples could fall outside of the main manifold and that we do not want to systematically lose that information. A *sparse representation* then becomes a possible way to represent data that is mostly low-dimensional, although occasionally occupying more dimensions. This can be achieved with a high-dimensional representation whose elements are 0 most of the time. We can see that the effective dimension (the number of non-zeros) then can change depending on where we are in input space, which can be useful. Sparse representations are discussed in Section 15.8.

The next part of the book introduces specific deep learning algorithms that aim at discovering representations that are useful for some task, i.e., trying to extract the directions of variations that matter for the task of interest, often in a

supervised setting. The last part of the book concentrates more on unsupervised representation learning algorithms, which attempt to capture all of the directions of variation that are salient in the data distribution.