Chapter 5

Machine Learning Basics

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

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

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

5.1 Learning Algorithms

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

5.1.1 The Task, T

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.1.2 The Performance Measure, P

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

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

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

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

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

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

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

5.1.3 The Experience, E

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.1.4 Example: Linear Regression

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

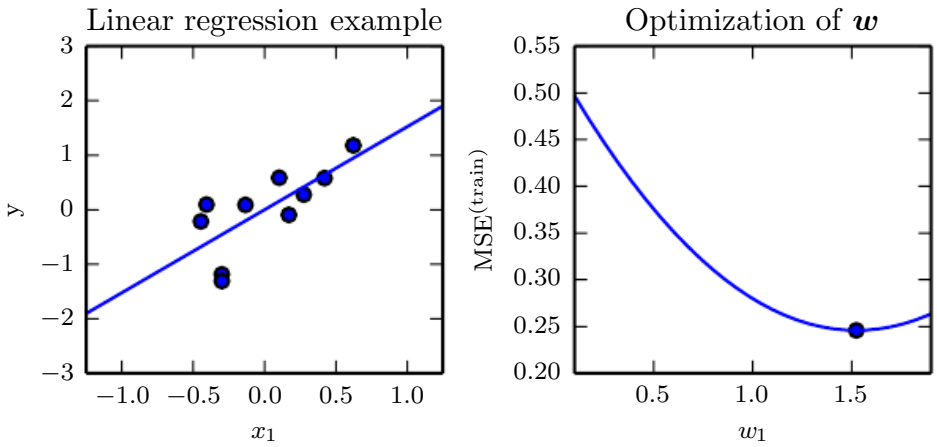

Figure 5.1: A linear regression problem, with a training set consisting of ten data points, each containing one feature. Because there is only one feature, the weight vector $\boldsymbol{w}$ contains only a single parameter to learn, w_1. *(Left)*Observe that linear regression learns to set w_1 such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

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

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

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

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

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

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

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

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

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

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

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

5.2 Capacity, Overfitting and Underfitting

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

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

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

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

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

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

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

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

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

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

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

1. Make the training error small.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

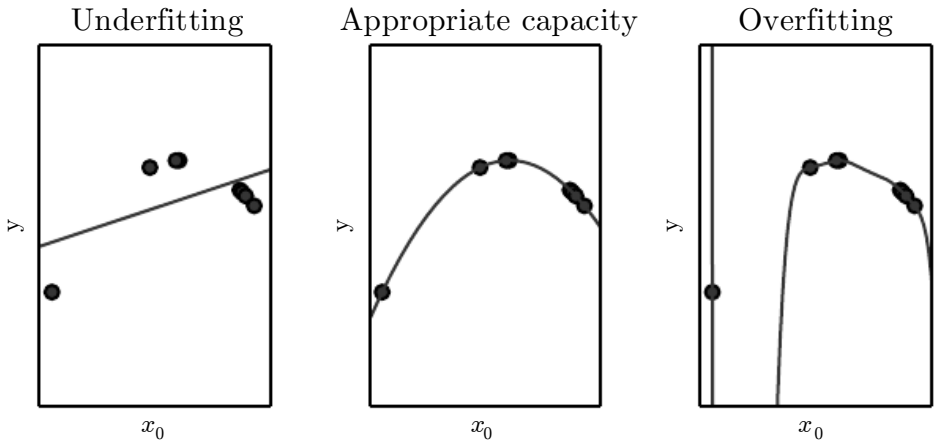

Figure 5.2: We fit three models to this example training set. The training data was generated synthetically, by randomly sampling x values and choosing y deterministically by evaluating a quadratic function. *(Left)*A linear function fit to the data suffers from underfitting—it cannot capture the curvature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

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

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

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

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

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

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

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

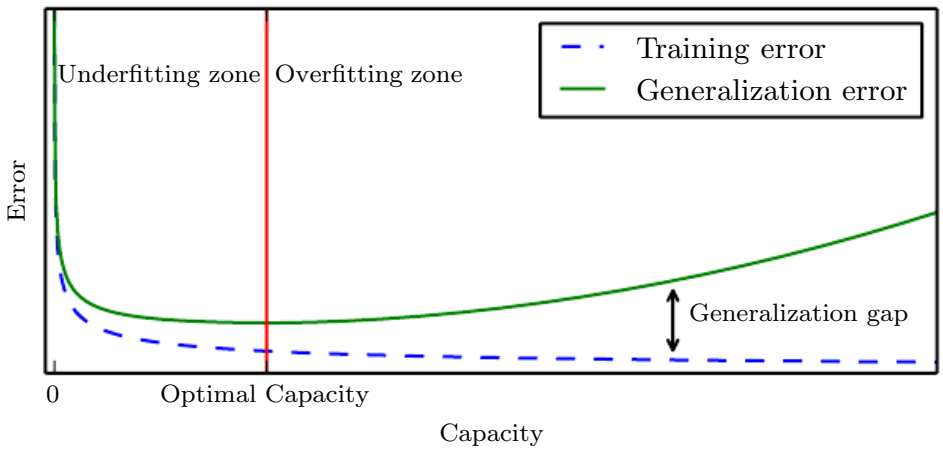

Figure 5.3: Typical relationship between capacity and error. Training and test error behave differently. At the left end of the graph, training error and generalization error are both high. This is the **underfitting regime**. As we increase capacity, training error decreases, but the gap between training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

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

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

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

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

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

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

5.2.1 The No Free Lunch Theorem

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

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

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

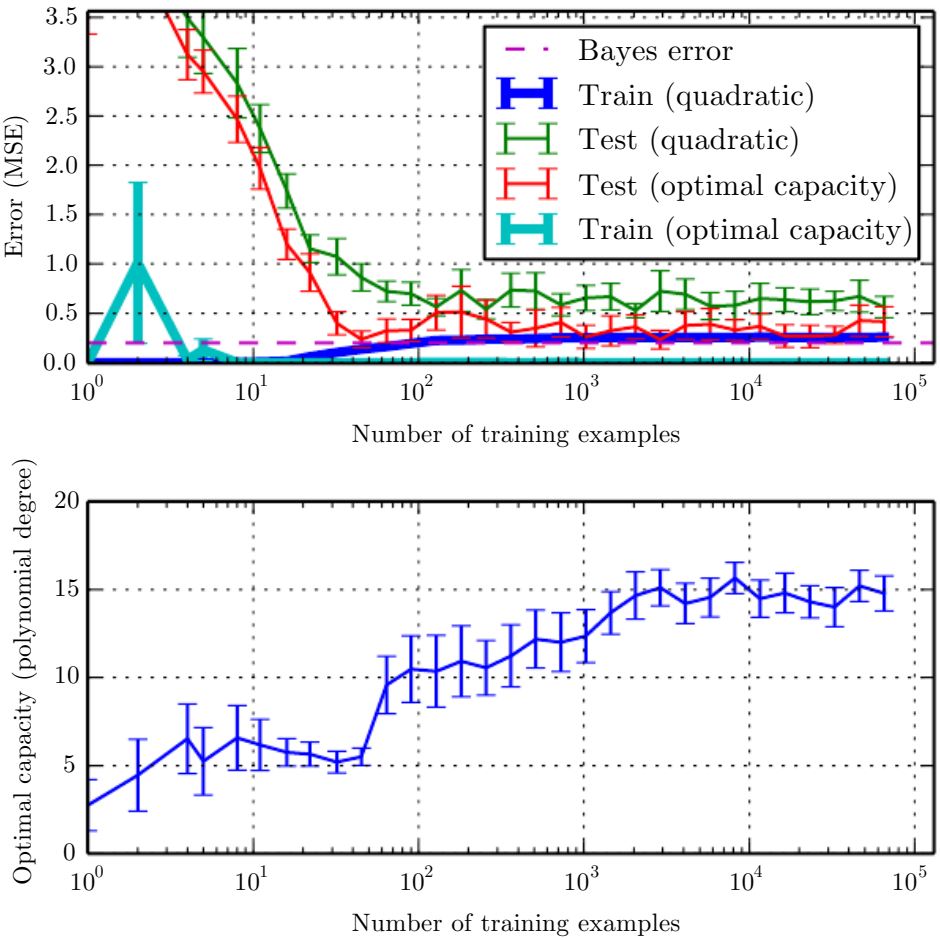

Figure 5.4: The effect of the training dataset size on the train and test error, as well as on the optimal model capacity. We constructed a synthetic regression problem based on adding a moderate amount of noise to a degree-5 polynomial, generated a single test set, and then generated several different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

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

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

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

5.2.2 Regularization

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

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

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

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