Chapter 19
Learning from Examples

In which we describe agents that can improve their behavior through diligent study of past experiences and predictions about the future.

An agent is **learning** if it improves its performance after making observations about the world. Learning can range from the trivial, such as jotting down a shopping list, to the profound, as when Albert Einstein inferred a new theory of the universe. When the agent is a computer, we call it **machine learning**: a computer observes some data, builds a **model** based on the data, and uses the model as both a **hypothesis** about the world and a piece of software that can solve problems.

Machine learning

Why would we want a machine to learn? Why not just program it the right way to begin with? There are two main reasons. First, the designers cannot anticipate all possible future situations. For example, a robot designed to navigate mazes must learn the layout of each new maze it encounters; a program for predicting stock market prices must learn to adapt when conditions change from boom to bust. Second, sometimes the designers have no idea how to program a solution themselves. Most people are good at recognizing the faces of family members, but they do it subconsciously, so even the best programmers don't know how to program a computer to accomplish that task, except by using machine learning algorithms.

In this chapter, we interleave a discussion of various model classes—decision trees (Section 19.3), linear models (Section 19.6), nonparametric models such as nearest neighbors (Section 19.7), ensemble models such as random forests (Section 19.8)—with practical advice on building machine learning systems (Section 19.9), and discussion of the theory of machine learning (Sections 19.1 to 19.5).

19.1 Forms of Learning

Any component of an agent program can be improved by machine learning. The improvements, and the techniques used to make them, depend on these factors:

- Which *component* is to be improved.
- What *prior knowledge* the agent has, which influences the *model* it builds.
- What *data* and *feedback* on that data is available.

Chapter 2 described several agent designs. The **components** of these agents include:

1. A direct mapping from conditions on the current state to actions.
2. A means to infer relevant properties of the world from the percept sequence.
3. Information about the way the world evolves and about the results of possible actions the agent can take.
4. Utility information indicating the desirability of world states.
5. Action-value information indicating the desirability of actions.
6. Goals that describe the most desirable states.
7. A problem generator, critic, and learning element that enable the system to improve.

Each of these components can be learned. Consider a self-driving car agent that learns by observing a human driver. Every time the driver brakes, the agent might learn a condition–action rule for when to brake (component 1). By seeing many camera images that it is told contain buses, it can learn to recognize them (component 2). By trying actions and observing the results—for example, braking hard on a wet road—it can learn the effects of its actions (component 3). Then, when it receives complaints from passengers who have been thoroughly shaken up during the trip, it can learn a useful component of its overall utility function (component 4).

The technology of machine learning has become a standard part of software engineering. Any time you are building a software system, even if you don’t think of it as an AI agent, components of the system can potentially be improved with machine learning. For example, software to analyze images of galaxies under gravitational lensing was speeded up by a factor of 10 million with a machine-learned model (Hezaveh *et al.*, 2017), and energy use for cooling data centers was reduced by 40% with another machine-learned model (Gao, 2014). Turing Award winner David Patterson and Google AI head Jeff Dean declared the dawn of a "Golden Age" for computer architecture due to machine learning (Dean *et al.*, 2018).

We have seen several examples of models for agent components: atomic, factored, and relational models based on logic or probability, and so on. Learning algorithms have been devised for all of these.

This chapter assumes little **prior knowledge** on the part of the agent: it starts from scratch and learns from the data. In Section 21.7.2 we consider **transfer learning**, in which knowledge from one domain is transferred to a new domain, so that learning can proceed faster with less data. We do assume, however, that the designer of the system chooses a model framework that can lead to effective learning.

Prior knowledge

Going from a specific set of observations to a general rule is called **induction**; from the observations that the sun rose every day in the past, we induce that the sun will come up tomorrow. This differs from the **deduction** we studied in Chapter 7 because the inductive conclusions may be incorrect, whereas deductive conclusions are guaranteed to be correct if the premises are correct.

This chapter concentrates on problems where the input is a **factored representation**—a vector of attribute values. It is also possible for the input to be any kind of data structure, including atomic and relational.

When the output is one of a finite set of values (such as *sunny/cloudy/rainy* or *true/false*), the learning problem is called **classification**. When it is a number (such as tomorrow's temperature, measured either as an integer or a real number), the learning problem has the (admittedly obscure[1]) name **regression**.

1 A better name would have been *function approximation* or *numeric prediction*. But in 1886 Francis Galton wrote an influential article on the concept of *regression to the mean* (e.g., the children of tall parents are likely to be taller than average, but not as tall as the parents). Galton showed plots with what he called “regression lines,” and readers came to associate the word “regression” with the statistical technique of function approximation rather than with the topic of regression to the mean.

Classification

Regression

There are three types of **feedback** that can accompany the inputs, and that determine the three main types of learning:

- In **supervised learning** the agent observes input-output pairs and learns a function that maps from input to output. For example, the inputs could be camera images, each one accompanied by an output saying “bus” or “pedestrian,” etc. An output like this is called a **label**. The agent learns a function that, when given a new image, predicts the appropriate label. In the case of braking actions (component 1 above), an input is the current state (speed and direction of the car, road condition), and an output is the distance it took to stop. In this case a set of output values can be obtained by the agent from its own percepts (after the fact); the environment is the teacher, and the agent learns a function that maps states to stopping distance.

Supervised learning

Label

- In **unsupervised learning** the agent learns patterns in the input without any explicit feedback. The most common unsupervised learning task is **clustering**: detecting potentially useful clusters of input examples. For example, when shown millions of images taken from the Internet, a computer vision system can identify a large cluster of similar images which an English speaker would call "cats."

Unsupervised learning

- In **reinforcement learning** the agent learns from a series of reinforcements: rewards and punishments. For example, at the end of a chess game the agent is told that it has won (a reward) or lost (a punishment). It is up to the agent to decide which of the actions prior to the reinforcement were most responsible for it, and to alter its actions to aim towards more rewards in the future.

Reinforcement learning

Feedback

19.2 Supervised Learning

More formally, the task of supervised learning is this:

> Given a **training set** of N example input–output pairs
>
> $$(x_1, y_1), (x_2, y_2), \ldots (x_N, y_N)\,,$$
>
> *Training set*
>
> where each pair was generated by an unknown function $y = f(x)$, discover a function h that approximates the true function f.

The function h is called a **hypothesis** about the world. It is drawn from a **hypothesis space** H of possible functions. For example, the hypothesis space might be the set of polynomials of degree 3; or the set of Javascript functions; or the set of 3-SAT Boolean logic formulas.

Hypothesis space

With alternative vocabulary, we can say that h is a **model** of the data, drawn from a **model class** H, or we can say a **function** drawn from a **function class**. We call the output y_i the **ground truth**—the true answer we are asking our model to predict.

Model class

Ground truth

How do we choose a hypothesis space? We might have some prior knowledge about the process that generated the data. If not, we can perform **exploratory data analysis**: examining the data with statistical tests and visualizations—histograms, scatter plots, box plots—to get a feel for the data, and some insight into what hypothesis space might be appropriate. Or we can just try multiple hypothesis spaces and evaluate which one works best.

Exploratory data analysis

Consistent hypothesis

How do we choose a good hypothesis from within the hypothesis space? We could hope for a **consistent hypothesis**: and h such that each x_i in the training set has $h(x_i) = y_i$. With continuous-valued outputs we can't expect an exact match to the ground truth; instead we look for a **best-fit function** for which each $h(x_i)$ is close to y_i (in a way that we will formalize in Section 19.4.2).

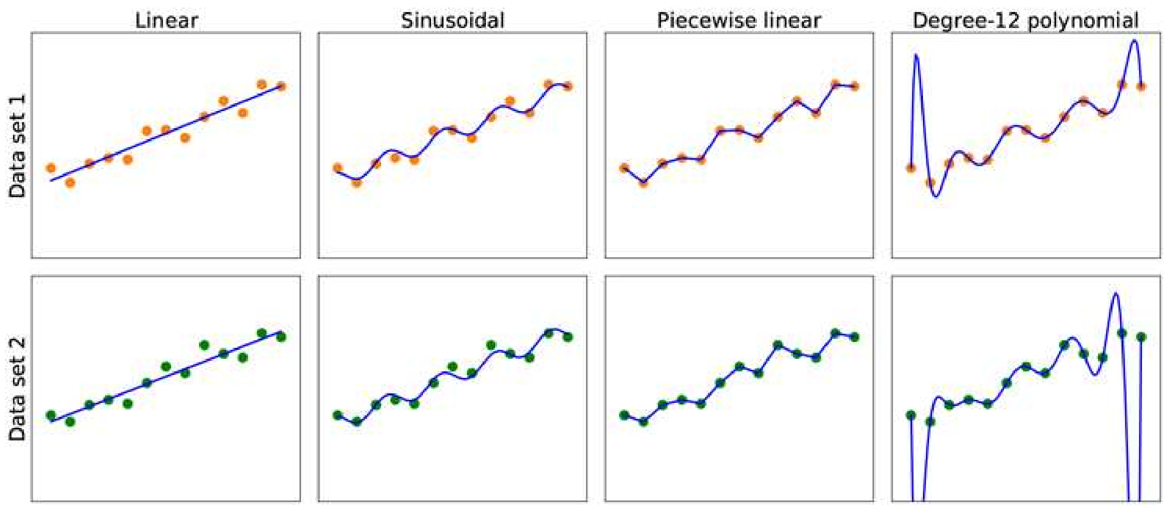
The true measure of a hypothesis is not how it does on the training set, but rather how well it handles inputs it has not yet seen. We can evaluate that with a second sample of (x_i, y_i) pairs called a **test set**. We say that h **generalizes** well if it accurately predicts the outputs of the test set.

Test set

Figure 19.1 shows that the function h that a learning algorithm discovers depends on the hypothesis space H it considers and on the training set it is given. Each of the four plots in the top row have the same training set of 13 data points in the (x, y) plane. The four plots in the bottom row have a second set of 13 data points; both sets are representative of the same unknown function $f(x)$. Each column shows the best-fit hypothesis h from a different hypothesis space:

- **COLUMN 1:** Straight lines; functions of the form $h(x) = w_1x + w_0$. There is no line that would be a consistent hypothesis for the data points.
- **COLUMN 2:** Sinusoidal functions of the form $h(x) = w_1x + \sin(w_0x)$. This choice is not quite consistent, but fits both data sets very well.
- **COLUMN 3:** Piecewise-linear functions where each line segment connects the dots from one data point to the next. These functions are always consistent.
- **COLUMN 4:** Degree-12 polynomials, $h(x) = \sum_{i=0}^{12} w_i x^i$. These are consistent: we can always get a degree-12 polynomial to perfectly fit 13 distinct points. But just because the hypothesis is consistent does not mean it is a good guess.

Figure 19.1



Finding hypotheses to fit data. **Top row**: four plots of best-fit functions from four different hypothesis spaces trained on data set 1. **Bottom row**: the same four functions, but trained on a slightly different data set (sampled from the same $f(x)$ function).

One way to analyze hypothesis spaces is by the bias they impose (regardless of the training data set) and the variance they produce (from one training set to another).

Bias

By **bias** we mean (loosely) the tendency of a predictive hypothesis to deviate from the expected value when averaged over different training sets. Bias often results from restrictions imposed by the hypothesis space. For example, the hypothesis space of linear functions induces a strong bias: it only allows functions consisting of straight lines. If there are any patterns in the data other than the overall slope of a line, a linear function will not be able to represent those patterns. We say that a hypothesis is **underfitting** when it fails to find a pattern in the data. On the other hand, the piecewise linear function has low bias; the shape of the function is driven by the data.

Underfitting

By **variance** we mean the amount of change in the hypothesis due to fluctuation in the training data. The two rows of Figure 19.1 represent data sets that were each sampled from the same $f(x)$ function. The data sets turned out to be slightly different. For the first three columns, the small difference in the data set translates into a small difference in the hypothesis. We call that low variance. But the degree-12 polynomials in the fourth column have high variance: look how different the two functions are at both ends of the x-axis. Clearly, at least one of these polynomials must be a poor approximation to the true $f(x)$. We say a function is **overfitting** the data when it pays too much attention to the particular data set it is trained on, causing it to perform poorly on unseen data.

Variance

Overfitting

Often there is a **bias–variance tradeoff**: a choice between more complex, low-bias hypotheses that fit the training data well and simpler, low-variance hypotheses that may generalize better. Albert Einstein said in 1933, "the supreme goal of all theory is to make the irreducible basic elements as simple and as few as possible without having to surrender the adequate representation of a single datum of experience." In other words, Einstein recommends choosing the simplest hypothesis that matches the data. This principle can be traced further back to the 14th-century English philosopher William of Ockham.[2] His principle that "plurality [of entities] should not be posited without necessity" is called **Ockham's razor** because it is used to "shave off" dubious explanations.

2 The name is often misspelled as "Occam."

Bias–variance tradeoff

Defining simplicity is not easy. It seems clear that a polynomial with only two parameters is simpler than one with thirteen parameters. We will make this intuition more precise in Section 19.3.4. However, in Chapter 21 we will see that deep neural network models can often generalize quite well, even though they are very complex—some of them have billions of parameters. So the number of parameters by itself is not a good measure of a model's fitness. Perhaps we should be aiming for "appropriateness," not "simplicity" in a model class. We will consider this issue in Section 19.4.1.

Which hypothesis is best in Figure 19.1? We can't be certain. If we knew the data represented, say, the number of hits to a Web site that grows from day to day, but also cycles depending on the time of day, then we might favor the sinusoidal function. If we knew the data was definitely not cyclic but had high noise, that would favor the linear function.

In some cases, an analyst is willing to say not just that a hypothesis is possible or impossible, but rather how probable it is. Supervised learning can be done by choosing the hypothesis h^* that is most probable given the data:

$$h^* = \underset{h \in H}{\operatorname{argmax}} \, P\left(h|data\right).$$

By Bayes' rule this is equivalent to

$$h^* = \underset{h \in H}{\operatorname{argmax}} \, P(data \middle| h) \, P(h).$$

Then we can say that the prior probability $P(h)$ is high for a smooth degree-1 or -2 polynomial and lower for a degree-12 polynomial with large, sharp spikes. We allow unusual-looking functions when the data say we really need them, but we discourage them by giving them a low prior probability.

Why not let H be the class of all computer programs, or all Turing machines? The problem is that *there is a tradeoff between the expressiveness of a hypothesis space and the computational complexity of finding a good hypothesis within that space.* For example, fitting a straight line to data is an easy computation; fitting high-degree polynomials is somewhat harder; and fitting Turing machines is undecidable. A second reason to prefer simple hypothesis spaces is that presumably we will want to use h after we have learned it, and computing $h(x)$ when h is a linear function is guaranteed to be fast, while computing an arbitrary Turing machine program is not even guaranteed to terminate.

For these reasons, most work on learning has focused on simple representations. In recent years there has been great interest in deep learning (Chapter 21), where representations are not simple but where the $h(x)$ computation still takes only a *bounded number of steps* to compute with appropriate hardware.

We will see that the expressiveness–complexity tradeoff is not simple: it is often the case, as we saw with first-order logic in Chapter 8, that an expressive language makes it possible for a *simple* hypothesis to fit the data, whereas restricting the expressiveness of the language means that any consistent hypothesis must be complex.

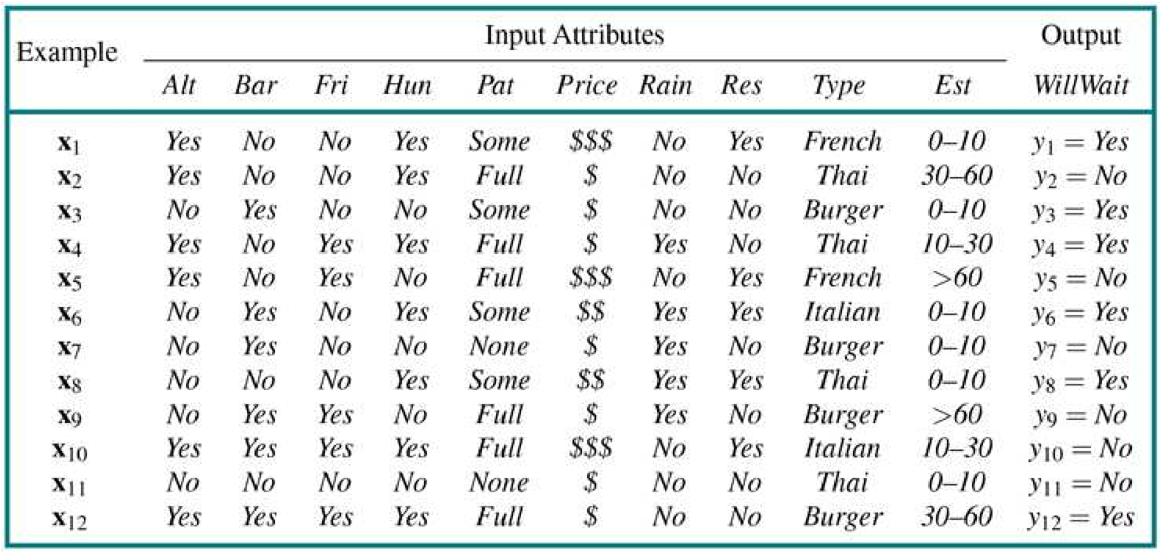
19.2.1 Example problem: Restaurant waiting

We will describe a sample supervised learning problem in detail: the problem of deciding whether to wait for a table at a restaurant. This problem will be used throughout the chapter to demonstrate different model classes. For this problem the output, y, is a Boolean variable that we will call *WillWait*; it is true for examples where we do wait for a table. The input, x, is a vector of ten attribute values, each of which has discrete values:

1. **ALTERNATE:** whether there is a suitable alternative restaurant nearby.
2. **BAR:** whether the restaurant has a comfortable bar area to wait in.
3. **FRI/SAT:** true on Fridays and Saturdays.
4. **HUNGRY:** whether we are hungry right now.
5. **PATRONS:** how many people are in the restaurant (values are *None*, *Some*, and *Full*).
6. **PRICE:** the restaurant's price range ($, $$, $$$).
7. **RAINING:** whether it is raining outside.
8. **RESERVATION:** whether we made a reservation.
9. **TYPE:** the kind of restaurant (French, Italian, Thai, or burger).
10. **WAITESTIMATE:** host's wait estimate: $0-10$, $10-30$, $30-60$, or >60 minutes.

A set of 12 examples, taken from the experience of one of us (SR), is shown in Figure 19.2. Note how skimpy these data are: there are $2^6 \times 3^2 \times 4^2 = 9,216$ possible combinations of values for the input attributes, but we are given the correct output for only 12 of them; each of the other 9,204 could be either true or false; we don't know. This is the essence of induction: we need to make our best guess at these missing 9,204 output values, given only the evidence of the 12 examples.

Figure 19.2



Example	Input Attributes										Output
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
$\mathbf{x}_1$	*Yes*	*No*	*No*	*Yes*	*Some*	*\$\$\$*	*No*	*Yes*	*French*	*0–10*	$y_1 = \textit{Yes}$
$\mathbf{x}_2$	*Yes*	*No*	*No*	*Yes*	*Full*	*\$*	*No*	*No*	*Thai*	*30–60*	$y_2 = \textit{No}$
$\mathbf{x}_3$	*No*	*Yes*	*No*	*No*	*Some*	*\$*	*No*	*No*	*Burger*	*0–10*	$y_3 = \textit{Yes}$
$\mathbf{x}_4$	*Yes*	*No*	*Yes*	*Yes*	*Full*	*\$*	*Yes*	*No*	*Thai*	*10–30*	$y_4 = \textit{Yes}$
$\mathbf{x}_5$	*Yes*	*No*	*Yes*	*No*	*Full*	*\$\$\$*	*No*	*Yes*	*French*	*>60*	$y_5 = \textit{No}$
$\mathbf{x}_6$	*No*	*Yes*	*No*	*Yes*	*Some*	*\$\$*	*Yes*	*Yes*	*Italian*	*0–10*	$y_6 = \textit{Yes}$
$\mathbf{x}_7$	*No*	*Yes*	*No*	*No*	*None*	*\$*	*Yes*	*No*	*Burger*	*0–10*	$y_7 = \textit{No}$
$\mathbf{x}_8$	*No*	*No*	*No*	*Yes*	*Some*	*\$\$*	*Yes*	*Yes*	*Thai*	*0–10*	$y_8 = \textit{Yes}$
$\mathbf{x}_9$	*No*	*Yes*	*Yes*	*No*	*Full*	*\$*	*Yes*	*No*	*Burger*	*>60*	$y_9 = \textit{No}$
$\mathbf{x}_{10}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*\$\$\$*	*No*	*Yes*	*Italian*	*10–30*	$y_{10} = \textit{No}$
$\mathbf{x}_{11}$	*No*	*No*	*No*	*No*	*None*	*\$*	*No*	*No*	*Thai*	*0–10*	$y_{11} = \textit{No}$
$\mathbf{x}_{12}$	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*\$*	*No*	*No*	*Burger*	*30–60*	$y_{12} = \textit{Yes}$

Examples for the restaurant domain.

19.5 The Theory of Learning

How can we be sure that our learned hypothesis will predict well for previously unseen inputs? That is, how do we know that the hypothesis h is close to the target function f if we don't know what f is? These questions have been pondered for centuries, by Ockham, Hume, and others. In recent decades, other questions have emerged: how many examples do we need to get a good h? What hypothesis space should we use? If the hypothesis space is very complex, can we even find the best h, or do we have to settle for a local maximum? How complex should h be? How do we avoid overfitting? This section examines these questions.

We'll start with the question of how many examples are needed for learning. We saw from the learning curve for decision tree learning on the restaurant problem (Figure 19.7 on page 661) that accuracy improves with more training data. Learning curves are useful, but they are specific to a particular learning algorithm on a particular problem. Are there some more general principles governing the number of examples needed?

Questions like this are addressed by **computational learning theory**, which lies at the intersection of AI, statistics, and theoretical computer science. The underlying principle is that any hypothesis that is seriously wrong will almost certainly be "found out" with high probability after a small number of examples, because it will make an incorrect prediction. Thus, any hypothesis that is consistent with a sufficiently large set of training examples is unlikely to be seriously wrong: that is, it must be **probably approximately correct (PAC)**.

Computational learning theory

Probably approximately correct (PAC)

Any learning algorithm that returns hypotheses that are probably approximately correct is called a **PAC learning** algorithm; we can use this approach to provide bounds on the performance of various learning algorithms.

PAC learning

PAC-learning theorems, like all theorems, are logical consequences of axioms. When a theorem (as opposed to, say, a political pundit) states something about the future based on the past, the axioms have to provide the "juice" to make that connection. For PAC learning, the juice is provided by the stationarity assumption introduced on page 665, which says that future examples are going to be drawn from the same fixed distribution $\mathbf{P}(E) = \mathbf{P}(X, Y)$ as past examples. (Note that we do not have to know what distribution that is, just that it doesn't change.) In addition, to keep things simple, we will assume that the true function f is deterministic and is a member of the hypothesis space H that is being considered.

The simplest PAC theorems deal with Boolean functions, for which the 0/1 loss is appropriate. The **error rate** of a hypothesis h, defined informally earlier, is defined formally here as the expected generalization error for examples drawn from the stationary distribution:

$$\text{error}(h) = GenLoss_{L_{0/1}}(h) = \sum_{x,y} L_{0/1}(y, h(x))\, P(x, y).$$

In other words, error(h) is the probability that h misclassifies a new example. This is the same quantity being measured experimentally by the learning curves shown earlier.

A hypothesis h is called **approximately correct** if error$(h) \leq \epsilon$, where ϵ is a small constant. We will show that we can find an N such that, after training on N examples, with high probability, all consistent hypotheses will be approximately correct. One can think of an approximately correct hypothesis as being "close" to the true function in hypothesis space: it lies inside what is called the **ϵ-ball** around the true function f. The hypothesis space outside this ball is called H_{bad}.

$\epsilon-$ ball

We can derive a bound on the probability that a "seriously wrong" hypothesis $h_b \in H_{\text{bad}}$ is consistent with the first N examples as follows. We know that $\text{error}(h_b) > \epsilon$. Thus, the probability that it agrees with a given example is at most $1-\epsilon$. Since the examples are independent, the bound for N examples is:

$$P(h_b \text{ agrees with } N \text{ examples}) \leq (1-\epsilon)^N .$$

The probability that H_{bad} contains at least one consistent hypothesis is bounded by the sum of the individual probabilities:

$$P(H_{\text{bad}} \text{ contains a consistent hypothesis}) \leq \left|H_{\text{bad}}\right|(1-\epsilon)^N \leq \left|H\right|(1-\epsilon)^N ,$$

where we have used the fact that H_{bad} is a subset of H and thus $|H_{\text{bad}}| \leq |H|$. We would like to reduce the probability of this event below some small number δ:

$$P(H_{\text{bad}} \text{ contains a consistent hypothesis}) \leq |H|(1-\epsilon)^N \leq \delta .$$

Given that $1-\epsilon \leq e^{-\epsilon}$, we can achieve this if we allow the algorithm to see

(19.1)

$$N \geq \frac{1}{\epsilon}\left(\ln\frac{1}{\delta} + \ln|H|\right)$$

examples. Thus, with probability at least $1-\delta$, after seeing this many examples, the learning algorithm will return a hypothesis that has error at most ϵ. In other words, it is probably approximately correct. The number of required examples, as a function of ϵ and δ, is called the **sample complexity** of the learning algorithm.

Sample complexity

As we saw earlier, if H is the set of all Boolean functions on n attributes, then $|H| = 2^{2^n}$. Thus, the sample complexity of the space grows as 2^n. Because the number of possible examples is also 2^n, this suggests that PAC-learning in the class of all Boolean functions requires seeing all, or nearly all, of the possible examples. A moment's thought reveals the reason for this: H contains enough hypotheses to classify any given set of examples in all possible ways. In particular, for any set of N examples, the set of hypotheses consistent with those examples contains equal numbers of hypotheses that predict x_{N+1} to be positive and hypotheses that predict x_{N+1} to be negative.

To obtain real generalization to unseen examples, then, it seems we need to restrict the hypothesis space H in some way; but of course, if we do restrict the space, we might eliminate the true function altogether. There are three ways to escape this dilemma.

The first is to bring prior knowledge to bear on the problem.

The second, which we introduced in Section 19.4.3, is to insist that the algorithm return not just any consistent hypothesis, but preferably a simple one (as is done in decision tree learning). In cases where finding simple consistent hypotheses is tractable, the sample complexity results are generally better than for analyses based only on consistency.

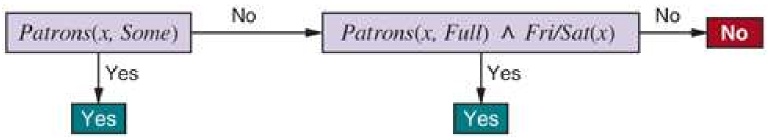
The third, which we pursue next, is to focus on learnable subsets of the entire hypothesis space of Boolean functions. This approach relies on the assumption that the restricted hypothesis space contains a hypothesis h that is close enough to the true function f; the benefits are that the restricted hypothesis space allows for effective generalization and is typically easier to search. We now examine one such restricted hypothesis space in more detail.

19.5.1 PAC learning example: Learning decision lists

We now show how to apply PAC learning to a new hypothesis space: **decision lists**. A decision list consists of a series of tests, each of which is a conjunction of literals. If a test succeeds when applied to an example description, the decision list specifies the value to be returned. If the test fails, processing continues with the next test in the list. Decision lists resemble decision trees, but their overall structure is simpler: they branch only in one direction. In contrast, the individual tests are more complex. Figure 19.10 shows a decision list that represents the following hypothesis:

$$WillWait \;\Leftrightarrow\; (Patrons = Some) \vee (Patrons = Full \wedge Fri/Sat).$$

Figure 19.10



A decision list for the restaurant problem.

Decision lists

If we allow tests of arbitrary size, then decision lists can represent any Boolean function (Exercise 19.DLEX). On the other hand, if we restrict the size of each test to at most k literals, then it is possible for the learning algorithm to generalize successfully from a small number of examples. We use the notation k-DL for a decision list with up to k conjunctions. The example in Figure 19.10 is in 2-DL. It is easy to show (Exercise 19.DLEX) that k-DL includes as a subset $k-$**DT**, the set of all decision trees of depth at most k. We will use the notation k-DL(n) to denote a k-DL using n Boolean attributes.

K-DT

The first task is to show that k-DL is learnable—that is, that any function in k-DL can be approximated accurately after training on a reasonable number of examples. To do this, we need to calculate the number of possible hypotheses. Let the set of conjunctions of at most k literals using n attributes be $Conj(n,k)$. Because a decision list is constructed from tests, and because each test can be attached to either a *Yes* or a *No* outcome or can be absent from the decision list, there are at most $3^{|Conj(n,k)|}$ distinct sets of component tests. Each of these sets of tests can be in any order, so

$$|k\text{-DL}(n)| \leq 3^c c! \text{ where } c = |Conj(n,k)|.$$

The number of conjunctions of at most k literals from n attributes is given by

$$\left|Conj(n,k)\right| = \sum_{i=0}^{k} \binom{2n}{i} = O(n^k).$$

Hence, after some work, we obtain

$$\left|k\text{-DL}(n)\right| = 2^{O(n^k \log_2(n^k))}.$$

We can plug this into Equation (19.1) to show that the number of examples needed for PAC-learning a k-DL(n) function is polynomial in n:

$$N \geq \frac{1}{\epsilon}\left(\ln \frac{1}{\delta} + O(n^k \log_2(n^k))\right).$$

Therefore, any algorithm that returns a consistent decision list will PAC-learn a k-DL function in a reasonable number of examples, for small k.

The next task is to find an efficient algorithm that returns a consistent decision list. We will use a greedy algorithm called DECISION-LIST-LEARNING that repeatedly finds a test that agrees exactly with some subset of the training set. Once it finds such a test, it adds it to the decision list under construction and removes the corresponding examples. It then constructs the remainder of the decision list, using just the remaining examples. This is repeated until there are no examples left. The algorithm is shown in Figure 19.11.

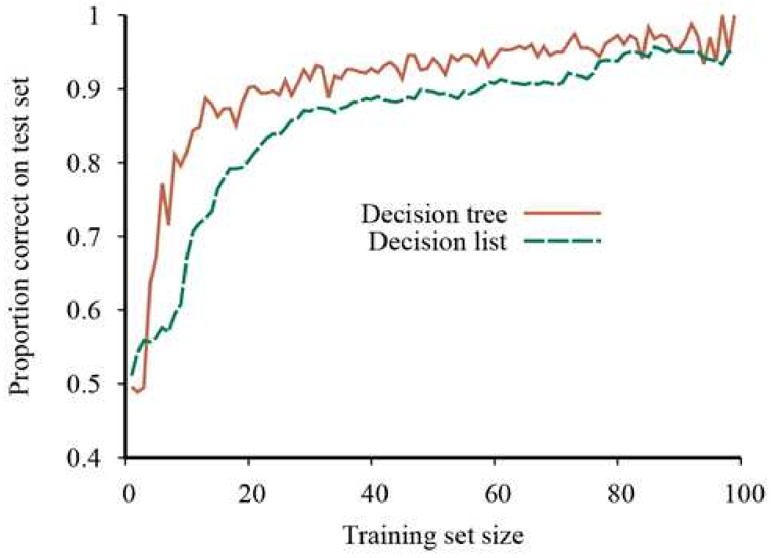
Figure 19.11

```
function DECISION-LIST-LEARNING(examples) returns a decision list, or failure
    if examples is empty then return the trivial decision list No
    t ← a test that matches a nonempty subset examples_t of examples
        such that the members of examples_t are all positive or all negative
    if there is no such t then return failure
    if the examples in examples_t are positive then o ← Yes else o ← No
    return a decision list with initial test t and outcome o and remaining tests given by
        DECISION-LIST-LEARNING(examples − examples_t)
```

An algorithm for learning decision lists.

This algorithm does not specify the method for selecting the next test to add to the decision list. Although the formal results given earlier do not depend on the selection method, it would seem reasonable to prefer small tests that match large sets of uniformly classified examples, so that the overall decision list will be as compact as possible. The simplest strategy is to find the smallest test *t* that matches any uniformly classified subset, regardless of the size of the subset. Even this approach works quite well, as Figure 19.12 suggests. For this problem, the decision tree learns a bit faster than the decision list, but has more variation. Both methods are over 90% accurate after 100 trials.

Figure 19.12



Learning curve for Decision-List-Learning algorithm on the restaurant data. The curve for Learn-Decision-Tree is shown for comparison; decision trees do slightly better on this particular problem.

19.6 Linear Regression and Classification

Now it is time to move on from decision trees and lists to a different hypothesis space, one that has been used for hundreds of years: the class of **linear functions** of continuous-valued inputs. We'll start with the simplest case: regression with a univariate linear function, otherwise known as "fitting a straight line." Section 19.6.3 covers the multivariable case. Sections 19.6.4 and 19.6.5 show how to turn linear functions into classifiers by applying hard and soft thresholds.

Linear function

19.6.1 Univariate linear regression

A univariate linear function (a straight line) with input x and output y has the form $y = w_1x + w_0$, where w_0 and w_1 are real-valued coefficients to be learned. We use the letter w because we think of the coefficients as **weights**; the value of y is changed by changing the relative weight of one term or another. We'll define $\mathbf{w}$ to be the vector $\langle w_0, w_1 \rangle$, and define the linear function with those weights as

$$h_{\mathbf{w}}(x) = w_1x + w_0.$$

Weight

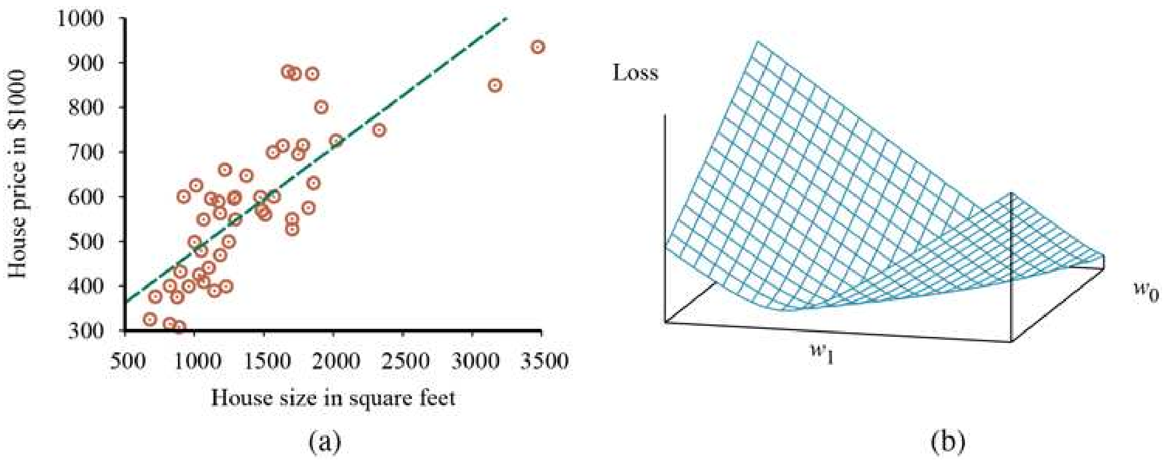
Figure 19.13(a) shows an example of a training set of n points in the x, y plane, each point representing the size in square feet and the price of a house offered for sale. The task of finding the $h_{\mathbf{w}}$ that best fits these data is called **linear regression**. To fit a line to the data, all we have to do is find the values of the weights $\langle w_0, w_1 \rangle$ that minimize the empirical loss. It is traditional (going back to Gauss[6]) to use the squared-error loss function, L_2, summed over all the training examples:

[6] Gauss showed that if the y_j values have normally distributed noise, then the most likely values of w_1 and w_0 are obtained by using L_2 loss, minimizing the sum of the squares of the errors. (If the values have noise that follows a Laplace (double exponential) distribution, then L_1 loss is appropriate.)

$$Loss(h_{\mathbf{w}}) = \sum_{j=1}^{N} L_2\left(y_j, h_{\mathbf{w}}\left(x_j\right)\right) = \sum_{j=1}^{N}\left(y_j - h_{\mathbf{w}}\left(x_j\right)\right)^2 = \sum_{j=1}^{N}\left(y_j - \left(w_1 x_j + w_0\right)\right)^2.$$

Linear regression

Figure 19.13



(a) Data points of price versus floor space of houses for sale in Berkeley, CA, in July 2009, along with the linear function hypothesis that minimizes squared-error loss: $y = 0.232x + 246$. (b) Plot of the loss function $\sum_j (y_j - w_1 x_j + w_0)^2$ for various values of w_0, w_1. Note that the loss function is convex, with a single global minimum.

We would like to find $\mathbf{w}^* = \operatorname{argmin}_{\mathbf{w}} Loss(h_{\mathbf{w}})$. The sum $\sum_{j=1}^{N}(y_j - (w_1 x_j + w_0))^2$ is minimized when its partial derivatives with respect to w_0 and w_1 are zero:

(19.2)

$$\frac{\partial}{\partial w_0}\sum_{j=1}^{N}(y_j - (w_1 x_j + w_0))^2 = 0 \text{ and } \frac{\partial}{\partial w_1}\sum_{j=1}^{N}(y_j - (w_1 x_j + w_0))^2 = 0.$$

These equations have a unique solution:

(19.3)

$$w_1 = \frac{N\left(\sum x_j y_j\right) - \left(\sum x_j\right)\left(\sum y_j\right)}{N\left(\sum x_j^2\right) - \left(\sum x_j\right)^2}; \qquad w_0 = \left(\sum y_j - w_1\left(\sum x_j\right)\right)/N.$$

For the example in Figure 19.13(a), the solution is $w_1 = 0.232$, $w_0 = 246$, and the line with those weights is shown as a dashed line in the figure.

Many forms of learning involve adjusting weights to minimize a loss, so it helps to have a mental picture of what's going on in **weight space**—the space defined by all possible settings of the weights. For univariate linear regression, the weight space defined by w_0 and w_1 is two-dimensional, so we can graph the loss as a function of w_0 and w_1 in a 3D plot (see Figure 19.13(b)). We see that the loss function is **convex**, as defined on page 122; this is true for *every* linear regression problem with an L_2 loss function, and implies that there are no local minima. In some sense that's the end of the story for linear models; if we need to fit lines to data, we apply Equation (19.3).[7]

[7] With some caveats: the L_2 loss function is appropriate when there is normally distributed noise that is independent of x; all results rely on the stationarity assumption; etc.

Weight space

19.6.2 Gradient descent

The univariate linear model has the nice property that it is easy to find an optimal solution where the partial derivatives are zero. But that won't always be the case, so we introduce here a method for minimizing loss that does not depend on solving to find zeroes of the derivatives, and can be applied to any loss function, no matter how complex.

Gradient descent

As discussed in Section 4.2 (page 119) we can search through a continuous weight space by incrementally modifying the parameters. There we called the algorithm **hill climbing**, but here we are minimizing loss, not maximizing gain, so we will use the term **gradient descent**. We choose any starting point in weight space—here, a point in the (w_0, w_1) plane—and then compute an estimate of the gradient and move a small amount in the steepest downhill direction, repeating until we converge on a point in weight space with (local) minimum loss. The algorithm is as follows:

(19.4)

```
$\mathbf{w} \leftarrow$ any point in the parameter space
**while not** converged **do**
    **for each** $w_i$ **in w do**
        $w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Loss(\mathbf{w})$
```

Gradient descent

The parameter α, which we called the **step size** in Section 4.2, is usually called the **learning rate** when we are trying to minimize loss in a learning problem. It can be a fixed constant, or it can decay over time as the learning process proceeds.

Learning rate

For univariate regression, the loss is quadratic, so the partial derivative will be linear. (The only calculus you need to know is the **chain rule**: $\partial g(f(x))/\partial x = g'(f(x))\,\partial f(x)/\partial x$, plus the facts that $\frac{\partial}{\partial x}x^2 = 2x$ and $\frac{\partial}{\partial x}x = 1$.) Let's first work out the partial derivatives—the slopes—in the simplified case of only one training example, (x, y):

(19.5)

$$\begin{aligned}\frac{\partial}{\partial w_i}Loss(\mathbf{w}) &= \frac{\partial}{\partial w_i}(y-h_{\mathbf{w}}(x))^2 = 2(y-h_{\mathbf{w}}(x))\times\frac{\partial}{\partial w_i}(y-h_{\mathbf{w}}(x))\\ &= 2(y-h_{\mathbf{w}}(x))\times\frac{\partial}{\partial w_i}(y-(w_1x+w_0)).\end{aligned}$$

Chain rule

Applying this to both w_0 and w_1 we get:

$$\frac{\partial}{\partial w_0}Loss(\mathbf{w}) = -2\left(y-h_{\mathbf{w}}\left(x\right)\right); \qquad \frac{\partial}{\partial w_1}Loss\left(\mathbf{w}\right) = -2\left(y-h_{\mathbf{w}}\left(x\right)\right)\times x.$$

Plugging this into Equation (19.4), and folding the 2 into the unspecified learning rate α, we get the following learning rule for the weights:

$$w_0 \leftarrow w_0 + \alpha\ \left(y-h_{\mathbf{w}}\left(x\right)\right)\ ; \quad w_1 \leftarrow w_1 + \alpha\ \left(y-h_{\mathbf{w}}\left(x\right)\ \right)\times\ x.$$

These updates make intuitive sense: if $h_{\mathbf{w}}(x) > y$ (i.e., the output is too large), reduce w_0 a bit, and reduce w_1 if x was a positive input but increase w_1 if x was a negative input.

The preceding equations cover one training example. For N training examples, we want to minimize the sum of the individual losses for each example. The derivative of a sum is the sum of the derivatives, so we have:

$$w_0 \leftarrow w_0 + \alpha\sum_j(y_j-h_{\mathbf{w}}(x_j))\ ; \quad w_1 \leftarrow w_1 + \alpha\sum_j(y_j-h_{\mathbf{w}}(x_j))\ \times\ x_j.$$

These updates constitute the **batch gradient descent** learning rule for univariate linear regression (also called **deterministic gradient descent**). The loss surface is convex, which means that there are no local minima to get stuck in, and convergence to the global minimum is guaranteed (as long as we don't pick an α that is so large that it overshoots), but may be very slow: we have to sum over all N training examples for every step, and there may be many steps. The problem is compounded if N is larger than the processor's memory size. A step that covers all the training examples is called an **epoch**.

Batch gradient descent

A faster variant is called **stochastic gradient descent** or **SGD**: it randomly selects a small number of training examples at each step, and updates according to Equation (19.5). The original version of SGD selected only one training example for each step, but it is now more common to select a **minibatch** of m out of the N examples. Suppose we have $N = 10,000$ examples and choose a minibatch of size $m = 100$. Then on each step we have reduced the amount of computation by a factor of 100; but because the standard error of the estimated mean gradient is proportional to the square root of the number of examples, the standard error increases by only a factor of 10. So even if we have to take 10 times more steps before convergence, minibatch SGD is still 10 times faster than full batch SGD in this case.

Epoch

Stochastic gradient descent

SGD

Minibatch

With some CPU or GPU architectures, we can choose m to take advantage of parallel vector operations, making a step with m examples almost as fast as a step with only a single example. Within these constraints, we would treat m as a hyperparameter that should be tuned for each learning problem.

Convergence of minibatch SGD is not strictly guaranteed; it can oscillate around the minimum without settling down. We will see on page 684 how a schedule of decreasing the learning rate, α, (as in simulated annealing) does guarantee convergence.

SGD can be helpful in an online setting, where new data are coming in one at a time, and the stationarity assumption may not hold. (In fact, SGD is also known as **online gradient descent**.) With a good choice for α a model will slowly evolve, remembering some of what it learned in the past, but also adapting to the changes represented by the new data.

Online gradient descent

SGD is widely applied to models other than linear regression, in particular neural networks. Even when the loss surface is not convex, the approach has proven effective in finding good local minima that are close to the global minimum.

19.6.3 Multivariable linear regression

We can easily extend to **multivariable linear regression** problems, in which each example $\mathbf{x}_j$ is an n-element vector.[8] Our hypothesis space is the set of functions of the form

[8] The reader may wish to consult Appendix A for a brief summary of linear algebra. Also, note that we use the term "multivariable regression" to mean that the input is a vector of multiple values, but the output is a single variable. We will use the term "multivariate regression" for the case where the output is also a vector of multiple variables. However, other authors use the two terms interchangeably.

$$h_{\mathbf{w}}(\mathbf{x}_j) = w_0 + w_1 x_{j,1} + \cdots + w_n x_{j,n} = w_0 + \sum_i w_i x_{j,i}.$$

Multivariable linear regression

The w_0 term, the intercept, stands out as different from the others. We can fix that by inventing a dummy input attribute, $x_{j,0}$, which is defined as always equal to 1. Then h is simply the dot product of the weights and the input vector (or equivalently, the matrix product of the transpose of the weights and the input vector):

$$h_{\mathbf{w}}(\mathbf{x}_j) = \mathbf{w} \cdot \mathbf{x}_j = \mathbf{w}^\top \mathbf{x}_j = \sum_i w_i x_{j,i}.$$

The best vector of weights, $\mathbf{w}^*$, minimizes squared-error loss over the examples:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_j L_2(y_j, \mathbf{w} \cdot \mathbf{x}_j).$$

Multivariable linear regression is actually not much more complicated than the univariate case we just covered. Gradient descent will reach the (unique) minimum of the loss function; the update equation for each weight w_i is

(19.6)

$$w_i \leftarrow w_i + \alpha \sum_j (y_j - h_{\mathbf{w}}(\mathbf{x}_j)) \times x_{j,i}.$$

With the tools of linear algebra and vector calculus, it is also possible to solve analytically for the $\mathbf{w}$ that minimizes loss. Let $\mathbf{y}$ be the vector of outputs for the training examples, and $\mathbf{X}$ be the **data matrix**—that is, the matrix of inputs with one n-dimensional example per row. Then the vector of predicted outputs is $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w}$ and the squared-error loss over all the training data is

$$L(\mathbf{w}) = \| \hat{\mathbf{y}} - \mathbf{y} \|^2 = \| \mathbf{X}\mathbf{w} - \mathbf{y} \|^2.$$

Data matrix

We set the gradient to zero:

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = 2\mathbf{X}^\top (\mathbf{X}\mathbf{w} - \mathbf{y}) = 0.$$

Rearranging, we find that the minimum-loss weight vector is given by

(19.7)

$$\mathbf{w}^* = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}.$$

We call the expression $(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top$ the **pseudoinverse** of the data matrix, and Equation (19.7) is called the **normal equation**.

Pseudoinverse

Normal equation

With univariate linear regression we didn't have to worry about overfitting. But with multivariable linear regression in high-dimensional spaces it is possible that some dimension that is actually irrelevant appears by chance to be useful, resulting in overfitting.

Thus, it is common to use **regularization** on multivariable linear functions to avoid overfitting. Recall that with regularization we minimize the total cost of a hypothesis, counting both the empirical loss and the complexity of the hypothesis:

$$Cost(h) = EmpLoss(h) + \lambda\, Complexity(h).$$

For linear functions the complexity can be specified as a function of the weights. We can consider a family of regularization functions:

$$Complexity(h_{\mathbf{w}}) = L_q(\mathbf{w}) = \sum_i |w_i|^q.$$

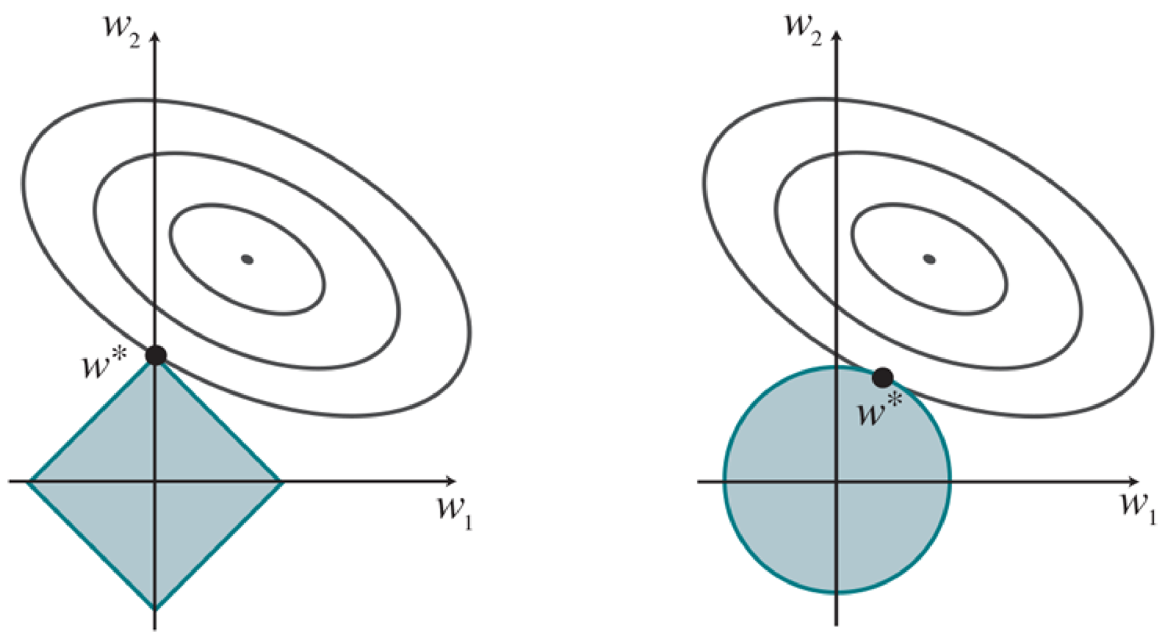
As with loss functions, with $q = 1$ we have L_1 regularization[9], which minimizes the sum of the absolute values; with $q = 2$, L_2 regularization minimizes the sum of squares. Which regularization function should you pick? That depends on the specific problem, but L_1 regularization has an important advantage: it tends to produce a **sparse model**. That is, it often sets many weights to zero, effectively declaring the corresponding attributes to be completely irrelevant—just as LEARN-DECISION-TREE does (although by a different mechanism). Hypotheses that discard attributes can be easier for a human to understand, and may be less likely to overfit.

9 It is perhaps confusing that the notation L_1 and L_2 is used for both loss functions and regularization functions. They need not be used in pairs: you could use L_2 loss with L_1 regularization, or vice versa.

Sparse model

Figure 19.14 gives an intuitive explanation of why L_1 regularization leads to weights of zero, while L_2 regularization does not. Note that minimizing $Loss(\mathbf{w}) + \lambda Complexity(\mathbf{w})$ is equivalent to minimizing $Loss(\mathbf{w})$ subject to the constraint that $Complexity(\mathbf{w}) \leq c$, for some constant c that is related to λ. Now, in Figure 19.14(a) the diamond-shaped box represents the set of points $\mathbf{w}$ in two-dimensional weight space that have L_1 complexity less than c; our solution will have to be somewhere inside this box. The concentric ovals represent contours of the loss function, with the minimum loss at the center. We want to find the point in the box that is closest to the minimum; you can see from the diagram that, for an arbitrary position of the minimum and its contours, it will be common for the corner of the box to find its way closest to the minimum, just because the corners are pointy. And of course the corners are the points that have a value of zero in some dimension.

Figure 19.14



Why L_1 regularization tends to produce a sparse model. Left: With L_1 regularization (box), the minimal achievable loss (concentric contours) often occurs on an axis, meaning a weight of zero. Right: With L_2 regularization (circle), the minimal loss is likely to occur anywhere on the circle, giving no preference to zero weights.

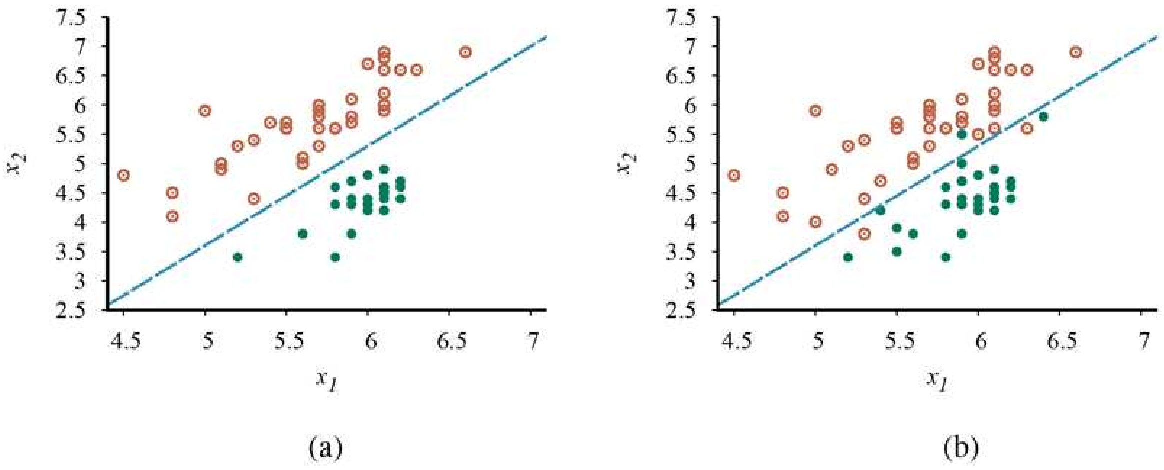
In Figure 19.14(b), we've done the same for the L_2 complexity measure, which represents a circle rather than a diamond. Here you can see that, in general, there is no reason for the intersection to appear on one of the axes; thus L_2 regularization does not tend to produce zero weights. The result is that the number of examples required to find a good h is linear in the number of irrelevant features for L_2 regularization, but only logarithmic with L_1 regularization. Empirical evidence on many problems supports this analysis.

Another way to look at it is that L_1 regularization takes the dimensional axes seriously, while L_2 treats them as arbitrary. The L_2 function is spherical, which makes it rotationally invariant: Imagine a set of points in a plane, measured by their x and y coordinates. Now imagine rotating the axes by 45°. You'd get a different set of (x', y') values representing the same points. If you apply L_2 regularization before and after rotating, you get exactly the same point as the answer (although the point would be described with the new (x', y') coordinates). That is appropriate when the choice of axes really is arbitrary—when it doesn't matter whether your two dimensions are distances north and east; or distances northeast and southeast. With L_1 regularization you'd get a different answer, because the L_1 function is not rotationally invariant. That is appropriate when the axes are not interchangeable; it doesn't make sense to rotate "number of bathrooms" 45° towards "lot size."

19.6.4 Linear classifiers with a hard threshold

Linear functions can be used to do classification as well as regression. For example, Figure 19.15(a) shows data points of two classes: earthquakes (which are of interest to seismologists) and underground explosions (which are of interest to arms control experts). Each point is defined by two input values, x_1 and x_2, that refer to body and surface wave magnitudes computed from the seismic signal. Given these training data, the task of classification is to learn a hypothesis h that will take new (x_1, x_2) points and return either 0 for earthquakes or 1 for explosions.

Figure 19.15



(a) Plot of two seismic data parameters, body wave magnitude x_1 and surface wave magnitude x_2, for earthquakes (open orange circles) and nuclear explosions (green circles) occurring between 1982 and 1990 in Asia and the Middle East (Kebeasy *et al.*, 1998). Also shown is a decision boundary between the classes. (b) The same domain with more data points. The earthquakes and explosions are no longer linearly separable.

A **decision boundary** is a line (or a surface, in higher dimensions) that separates the two classes. In Figure 19.15(a), the decision boundary is a straight line. A linear decision boundary is called a **linear separator** and data that admit such a separator are called **linearly separable**. The linear separator in this case is defined by

$$x_2 = 1.7x_1 - 4.9 \quad \text{or} \quad -4.9 + 1.7x_1 - x_2 = 0.$$

Decision boundary

Linear separator

Linear separability

The explosions, which we want to classify with value 1, are below and to the right of this line; they are points for which $-4.9 + 1.7x_1 - x_2 > 0$, while earthquakes have $-4.9 + 1.7x_1 - x_2 < 0$.. We can make the equation easier to deal with by changing it into the vector dot product form—with $x_0 = 1$ we have

$$-4.9x_0 + 1.7x_1 - x_2 = 0\,,$$

and we can define the vector of weights,

$$\mathbf{w} = \langle -4.9, 1.7, -1 \rangle,$$

and write the classification hypothesis

$$h_{\mathbf{w}}(\mathbf{x}) = 1 \text{ if } \mathbf{w} \cdot \mathbf{x} \geq 0 \text{ and } 0 \text{ otherwise.}$$

Alternatively, we can think of h as the result of passing the linear function $\mathbf{w} \cdot \mathbf{x}$ through a **threshold function**:

$$h_{\mathbf{w}}(\mathbf{x}) = Threshold(\mathbf{w} \cdot \mathbf{x}) \text{ where } Threshold(z) = 1 \text{ if } z \geq 0 \text{ and } 0 \text{ otherwise.}$$

Threshold function

The threshold function is shown in Figure 19.17(a).

Now that the hypothesis $h_{\mathbf{w}}(\mathbf{x})$ has a well-defined mathematical form, we can think about choosing the weights $\mathbf{w}$ to minimize the loss. In Sections 19.6.1 and 19.6.3, we did this both in closed form (by setting the gradient to zero and solving for the weights) and by gradient descent in weight space. Here we cannot do either of those things because the gradient is zero almost everywhere in weight space except at those points where $\mathbf{w} \cdot \mathbf{x} = 0$, and at those points the gradient is undefined.

There is, however, a simple weight update rule that converges to a solution—that is, to a linear separator that classifies the data perfectly—provided the data are linearly separable. For a single example $(\mathbf{x}, y)$, we have

(19.8)

$$w_i \leftarrow w_i + \alpha\,(y - h_{\mathbf{w}}(\mathbf{x})) \times x_i$$

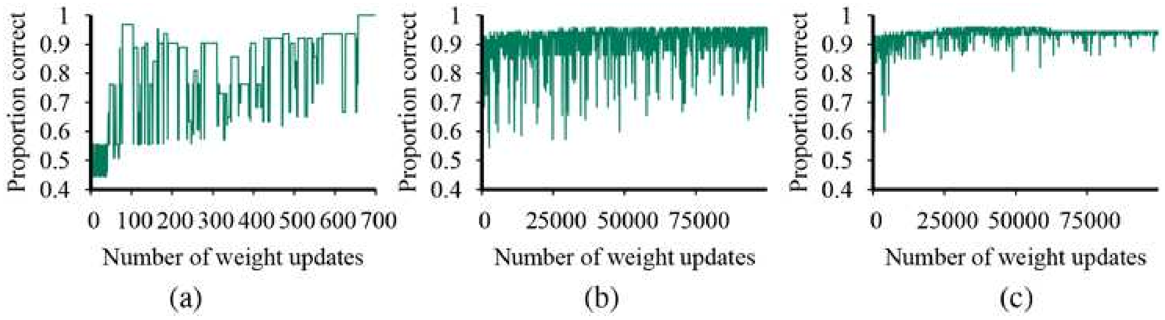
which is essentially identical to Equation (19.6), the update rule for linear regression! This rule is called the **perceptron learning rule**, for reasons that will become clear in Chapter 21. Because we are considering a 0/1 classification problem, however, the behavior is somewhat different. Both the true value y and the hypothesis output $h_{\mathbf{w}}(\mathbf{x})$ are either 0 or 1, so there are three possibilities:

- If the output is correct (i.e., $y = h_{\mathbf{w}}(\mathbf{x})$) then the weights are not changed.
- If y is 1 but $h_{\mathbf{w}}(\mathbf{x})$ is 0, then w_i is *increased* when the corresponding input x_i is positive and *decreased* when x_i is negative. This makes sense, because we want to make $\mathbf{w} \cdot \mathbf{x}$ bigger so that $h_{\mathbf{w}}(\mathbf{x})$ outputs a 1.
- If y is 0 but $h_{\mathbf{w}}(\mathbf{x})$ is 1, then w_i is *decreased* when the corresponding input x_i is positive and *increased* when x_i is negative. This makes sense, because we want to make $\mathbf{w} \cdot \mathbf{x}$ smaller so that $h_{\mathbf{w}}(\mathbf{x})$ outputs a 0.

Perceptron learning rule

Typically the learning rule is applied one example at a time, choosing examples at random (as in stochastic gradient descent). Figure 19.16(a) shows a **training curve** for this learning rule applied to the earthquake/explosion data shown in Figure 19.15(a). A training curve measures the classifier performance on a fixed training set as the learning process proceeds one update at a time on that training set. The curve shows the update rule converging to a zero-error linear separator. The "convergence" process isn't exactly pretty, but it always works. This particular run takes 657 steps to converge, for a data set with 63 examples, so each example is presented roughly 10 times on average. Typically, the variation across runs is large.

Figure 19.16



(a) Plot of total training-set accuracy vs. number of iterations through the training set for the perceptron learning rule, given the earthquake/explosion data in Figure 19.15(a). (b) The same plot for the noisy, nonseparable data in Figure 19.15(b); note the change in scale of the x-axis. (c) The same plot as in (b), with a learning rate schedule $\alpha(t) = 1000/(1000 + t)$.

Training curve

We have said that the perceptron learning rule converges to a perfect linear separator when the data points are linearly separable; but what if they are not? This situation is all too common in the real world. For example, Figure 19.15(b) adds back in the data points left out by Kebeasy *et al.*, (1998) when they plotted the data shown in Figure 19.15(a). In Figure 19.16(b), we show the perceptron learning rule failing to converge even after 10,000 steps: even though it hits the minimum-error solution (three errors) many times, the algorithm keeps changing the weights. In general, the perceptron rule may not converge to a stable solution for fixed learning rate α, but if α decays as $O(1/t)$ where t is the iteration number, then the rule can be shown to converge to a minimum-error solution when examples are presented in a random sequence.[10] It can also be shown that finding the minimum-error solution is NP-hard, so one expects that many presentations of the examples will be required for convergence to be achieved. Figure 19.16(c) shows the training process with a learning rate schedule $\alpha(t) = 1000/(1000 + t)$: convergence is not perfect after 100,000 iterations, but it is much better than the fixed-α case.

10 Technically, we require that $\sum_{t=1}^{\infty} \alpha(t) = \infty$ and $\sum_{t=1}^{\infty} \alpha^2(t) < \infty$. The learning rate $\alpha(t) = O(1/t)$ satisfies these conditions. Often we use $c/(c + t)$ for some fairly large constant c.

19.6.5 Linear classification with logistic regression

We have seen that passing the output of a linear function through the threshold function creates a linear classifier; yet the hard nature of the threshold causes some problems: the hypothesis $h_{\mathbf{w}}(\mathbf{x})$ is not differentiable and is in fact a discontinuous function of its inputs and its weights. This makes learning with the perceptron rule a very unpredictable adventure. Furthermore, the linear classifier always announces a completely confident prediction of 1 or 0, even for examples that are very close to the boundary; it would be better if it could classify some examples as a clear 0 or 1, and others as unclear borderline cases.

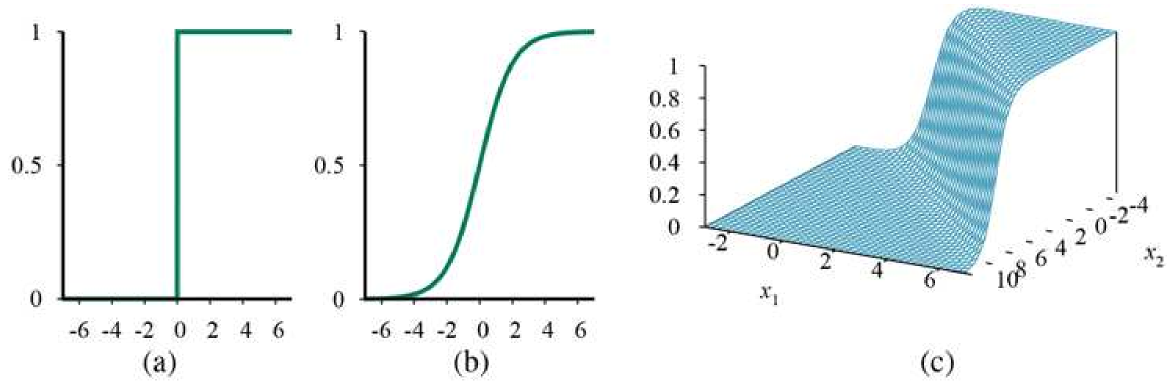
All of these issues can be resolved to a large extent by softening the threshold function—approximating the hard threshold with a continuous, differentiable function. In Chapter 13 (page 424), we saw two functions that look like soft thresholds: the integral of the standard normal distribution (used for the probit model) and the logistic function (used for the logit model). Although the two functions are very similar in shape, the logistic function

$$Logistic(z) = \frac{1}{1 + e^{-z}}$$

has more convenient mathematical properties. The function is shown in Figure 19.17(b). With the logistic function replacing the threshold function, we now have

$$h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w} \cdot \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}}}.$$

Figure 19.17



(a) The hard threshold function $Threshold(z)$ with 0/1 output. Note that the function is nondifferentiable at $z = 0$. (b) The logistic function, $Logistic(z) = \frac{1}{1+e^{-z}}$, also known as the sigmoid function. (c) Plot of a logistic regression hypothesis $h_{\mathbf{w}}(\mathbf{x}) = Logistic(\mathbf{w} \cdot \mathbf{x})$ for the data shown in Figure 19.15(b).

An example of such a hypothesis for the two-input earthquake/explosion problem is shown in Figure 19.17(c). Notice that the output, being a number between 0 and 1, can be interpreted as a *probability* of belonging to the class labeled 1. The hypothesis forms a soft boundary in the input space and gives a probability of 0.5 for any input at the center of the boundary region, and approaches 0 or 1 as we move away from the boundary.

The process of fitting the weights of this model to minimize loss on a data set is called **logistic regression**. There is no easy closed-form solution to find the optimal value of $\mathbf{w}$ with this model, but the gradient descent computation is straightforward. Because our hypotheses no longer output just 0 or 1, we will use the L_2 loss function; also, to keep the formulas readable, we'll use g to stand for the logistic function, with g' its derivative.

Logistic regression

For a single example $(\mathbf{x}, y)$, the derivation of the gradient is the same as for linear regression (Equation (19.5)) up to the point where the actual form of h is inserted. (For this derivation, we again need the chain rule.) We have

$$
\begin{aligned}
\frac{\partial}{\partial w_i} Loss(\mathbf{w}) &= \frac{\partial}{\partial w_i}(y - h_{\mathbf{w}}(\mathbf{x}))^2 \\
&= 2(y - h_{\mathbf{w}}(\mathbf{x})) \times \frac{\partial}{\partial w_i}(y - h_{\mathbf{w}}(\mathbf{x})) \\
&= -2(y - h_{\mathbf{w}}(\mathbf{x})) \times g'(\mathbf{w}\cdot\mathbf{x}) \times \frac{\partial}{\partial w_i}\mathbf{w}\cdot\mathbf{x} \\
&= -2(y - h_{\mathbf{w}}(\mathbf{x})) \times g'(\mathbf{w}\cdot\mathbf{x}) \times x_i .
\end{aligned}
$$

The derivative g' of the logistic function satisfies $g'(z) = g(z)(1 - g(z))$, so we have

$$g'(\mathbf{w}\cdot\mathbf{x}) = g(\mathbf{w}\cdot\mathbf{x})(1 - g(\mathbf{w}\cdot\mathbf{x})) = h_{\mathbf{w}}(\mathbf{x})(1 - h_{\mathbf{w}}(\mathbf{x}))$$

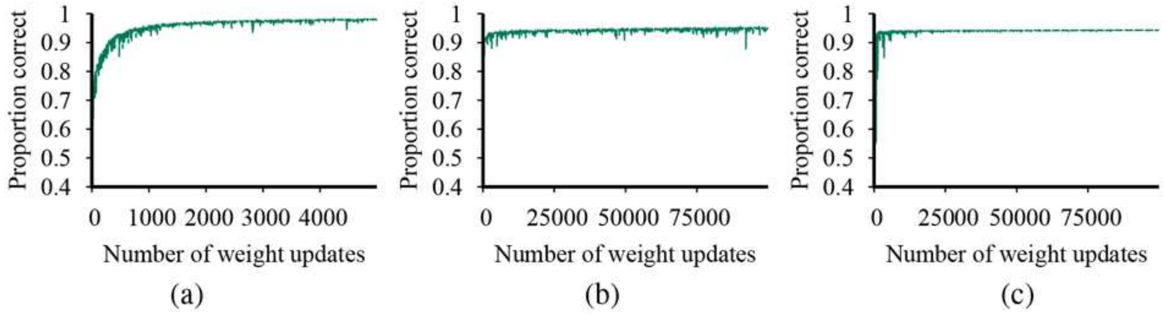
so the weight update for minimizing the loss takes a step in the direction of the difference between input and prediction, $(y - h_{\mathbf{w}}(\mathbf{x}))$, and the length of that step depends on the constant α and g':

(19.9)

$$w_i \leftarrow w_i + \alpha\,(y - h_{\mathbf{w}}(\mathbf{x})) \times h_{\mathbf{w}}(\mathbf{x})(1 - h_{\mathbf{w}}(\mathbf{x})) \times x_i .$$

Repeating the experiments of Figure 19.16 with logistic regression instead of the linear threshold classifier, we obtain the results shown in Figure 19.18. In (a), the linearly separable case, logistic regression is somewhat slower to converge, but behaves much more predictably. In (b) and (c), where the data are noisy and nonseparable, logistic regression converges far more quickly and reliably. These advantages tend to carry over into real-world applications, and logistic regression has become one of the most popular classification techniques for problems in medicine, marketing, survey analysis, credit scoring, public health, and other applications.

Figure 19.18



Repeat of the experiments in Figure 19.16 using logistic regression. The plot in (a) covers 5000 iterations rather than 700, while the plots in (b) and (c) use the same scale as before.

19.9 Developing Machine Learning Systems

In this chapter we have concentrated on explaining the *theory* of machine learning. The *practice* of using machine learning to solve practical problems is a separate discipline. Over the last 50 years, the software industry has evolved a software development methodology that makes it more likely that a (traditional) software project will be a success. But we are still in the early stages of defining a methodology for machine learning projects; the tools and techniques are not as well-developed. Here is a breakdown of typical steps in the process.

19.9.1 Problem formulation

The first step is to figure out what problem you want to solve. There are two parts to this. First ask, "what problem do I want to solve for my users?" An answer such as "make it easier for users to organize and access their photos" is too vague; "help a user find all photos that match a specific term, such as *Paris*" is better. Then ask, "what part(s) of the problem can be solved by machine learning?" perhaps settling on "learn a function that maps a photo to a set of labels; then, when given a label as a query, retrieve all photos with that label."

To make this concrete, you need to specify a loss function for your machine learning component, perhaps measuring the system's accuracy at predicting a correct label. This objective should be correlated with your true goals, but usually will be distinct—the true goal might be to maximize the number of users you gain and keep on your system, and the revenue that they produce. Those are metrics you should track, but not necessarily ones that you can directly build a machine learning model for.

When you have decomposed your problem into parts, you may find that there are multiple components that can be handled by old-fashioned software engineering, not machine learning. For example, for a user who asks for "best photos," you could implement a simple procedure that sorts photos by the number of likes and views. Once you have developed your overall system to the point where it is viable, you can then go back and optimize, replacing the simple components with more sophisticated machine learning models.

Part of problem formulation is deciding whether you are dealing with supervised, unsupervised, or reinforcement learning. The distinctions are not always so crisp. In **semisupervised learning** we are given a few labeled examples and use them to mine more information from a large collection of unlabeled examples. This has become a common approach, with companies emerging whose missions are to quickly label some examples, in order to help machine learning systems make better use of the remaining unlabeled examples.

Semisupervised learning

Sometimes you have a choice of which approach to use. Consider a system to recommend songs or movies to customers. We could approach this as a supervised learning problem, where the inputs include a representation of the customer and the labeled output is whether or not they liked the recommendation, or we could approach it as a reinforcement learning problem, where the system makes a series of recommendation actions, and occasionally gets a reward from the customer for making a good suggestion.

The labels themselves may not be the oracular truths that we hope for. Imagine that you are trying to build a system to guess a person's age from a photo. You gather some labeled examples by having people upload photos and state their age. That's supervised learning. But in reality some of the people lied about their age. It's not just that there is random noise in the data; rather the inaccuracies are systematic, and to uncover them is an unsupervised learning problem involving images, self-reported ages, and true (unknown) ages. Thus, both noise and lack of labels create a continuum between supervised and unsupervised learning. The field of **weakly supervised learning** focuses on using labels that are noisy, imprecise, or supplied by non-experts.

Weakly supervised learning

19.9.2 Data collection, assessment, and management

Every machine learning project needs data; in the case of our photo identification project there are freely available image data sets, such as **ImageNet**, which has over 14 million photos with about 20,000 different labels. Sometimes we may have to manufacture our own data, which can be done by our own labor, or by **crowdsourcing** to paid workers or unpaid volunteers operating over an Internet service. Sometimes data come from your users. For example, the Waze navigation service encourages users to upload data about traffic jams, and uses that to provide up-to-date navigation directions for all users. Transfer learning (see Section 21.7.2) can be used when you don't have enough of your own data: start with a publicly available general-purpose data set (or a model that has been pretrained on this data), and then add specific data from your users and retrain.

ImageNet

If you deploy a system to users, your users will provide feedback—perhaps by clicking on one item and ignoring the others. You will need a strategy for dealing with this data. That involves a review with privacy experts (see Section 27.3.2) to make sure that you get the proper permission for the data you collect, and that you have processes for insuring the integrity of the user's data, and that they understand what you will do with it. You also need to ensure that your processes are fair and unbiased (see Section 27.3.3). If there is data that you feel is too sensitive to collect but that would be useful for a machine learning model, consider a federated learning approach where the data stays on the user's device, but model parameters are shared in a way that does not reveal private data.

It is good practice to maintain **data provenance** for all your data. For each column in your data set, you should know the exact definition, where the data come from, what the possible values are, and who has worked on it. Were there periods of time in which a data feed was interrupted? Did the definition of some data source evolve over time? You'll need to know this if you want to compare results across time periods.

This is particularly true if you are relying on data that are produced by someone else—their needs and yours might diverge, and they might end up changing the way the data are produced, or might stop updating it all together. You need to monitor your data feeds to catch this. Having a reliable, flexible, secure, data-handling pipeline is more critical to success than the exact details of the machine learning algorithm. Provenance is also important for legal reasons, such as compliance with privacy law.

For any task there will be questions about the data: Is this the right data for my task? Does it capture enough of the right inputs to give us a chance of learning a model? Does it contain the outputs I want to predict? If not, can I build an unsupervised model? Or can I label a portion of the data and then do semisupervised learning? Is it relevant data? It is great to have 14 million photos, but if all your users are specialists interested in a specific topic, then a general database won't help—you'll need to collect photos on the specific topic. How much training data is enough? (Do I need to collect more data? Can I discard some data to make computation faster?) The best way to answer this is to reason by analogy to a similar project with known training set size.

Once you get started you can draw a learning curve (see Figure 19.7) to see if more data will help, or if learning has already plateaued. There are endless ad hoc, unjustified rules of thumb for the number of training examples you'll need: millions for hard problems; thousands for average problems; hundreds or thousands for each class in a classification problem; 10 times more examples than parameters of the model; 10 times more examples than input features; $O(d \log d)$ examples for d input features; more examples for nonlinear models than for linear models; more examples if greater accuracy is required; fewer examples if you use regularization; enough examples to achieve the statistical power necessary to reject the null hypothesis in classification. All these rules come with caveats—as does the sensible rule that suggests trying what has worked in the past for similar problems.

You should think defensively about your data. Could there be data entry errors? What can be done with missing data fields? If you collect data from your customers (or other people) could some of the people be adversaries out to game the system? Are there spelling errors or inconsistent terminology in text data? (For example, do “Apple,” “AAPL,” and “Apple Inc.” all refer to the same company?) You will need a process to catch and correct all these potential sources of data error.

When data are limited, **data augmentation** can help. For example, with a data set of images, you can create multiple versions of each image by rotating, translating, cropping, or scaling each image, or by changing the brightness or color balance or adding noise. As long as these are small changes, the image label should remain the same, and a model trained on such augmented data will be more robust.

Data augmentation

Sometimes data are plentiful but are classified into **unbalanced classes**. For example, a training set of credit card transactions might consist of 10,000,000 valid transactions and 1,000 fraudulent ones. A classifier that says “valid” regardless of the input will achieve 99.99% accuracy on this data set. To go beyond that, a classifier will have to pay more attention to the fraudulent examples. To help it do that, you can **undersample** the majority class (i.e., ignore some of the “valid” class examples) or **over-sample** the minority class (i.e., duplicate some of the “fraudulent” class examples). You can use a weighted loss function that gives a larger penalty to missing a fraudulent case.

Unbalanced classes

Undersampling

Over-sample

Boosting can also help you focus on the minority class. If you are using an ensemble method, you can change the rules by which the ensemble votes and give "fraudulent" as the response even if only a minority of the ensemble votes for "fraudulent." You can help balance unbalanced classes by generating synthetic data with techniques such as SMOTE (Chawla *et al.*, 2002) or ADASYN (He *et al.*, 2008).

You should carefully consider **outliers** in your data. An outlier is a data point that is far from other points. For example, in the restaurant problem, if price were a numeric value rather than a categorical one, and if one example had a price of $316 while all the others were $30 or less, that example would be an outlier. Methods such as linear regression are susceptible to outliers because they must form a single global linear model that takes all inputs into account—they can't treat the outlier differently from other example points, and thus a single outlier can have a large effect on all the parameters of the model.

Outlier

With attributes like price that are positive numbers, we can diminish the effect of outliers by transforming the data, taking the logarithm of each value, so $20, $25, and $316 become 1.3, 1.4, and 2.5. This makes sense from a practical point of view because the high value now has less influence on the model, and from a theoretical point of view because, as we saw in Section 16.3.2, the utility of money is logarithmic.

Methods such as decision trees that are built from multiple local models can treat outliers individually: it doesn't matter if the biggest value is $300 or $31; either way it can be treated in its own local node after a test of the form $cost \leq 30$. That makes decision trees (and thus random forests and gradient boosting) more robust to outliers.

Feature engineering

After correcting overt errors, you may also want to preprocess your data to make it easier to digest. We have already seen the process of quantization: forcing a continuous valued input, such as the wait time, into fixed bins $(0-10\,\text{minutes}, 10-30, 30-60, \text{or} > 60)$. Domain knowledge can tell you what thresholds are important, such as comparing $age \geq 18$ when studying voting patterns. We also saw (page 688) that nearest-neighbor algorithms perform better when data are normalized to have a standard deviation of 1. With categorical attributes such as sunny/cloudy/rainy, it is often helpful to transform the data into three separate Boolean attributes, exactly one of which is true (we call this a **one-hot encoding**). This is particularly useful when the machine learning model is a neural network.

One-hot encoding

You can also introduce new attributes based on your domain knowledge. For example, given a data set of customer purchases where each entry has a date attribute, you might want to augment the data with new attributes saying whether the date is a weekend or holiday.

As another example, consider the task of estimating the true value of houses that are for sale. In Figure 19.13 we showed a toy version of this problem, doing linear regression of house size to asking price. But we really want to estimate the selling price of a house, not the asking price. To solve this task we'll need data on actual sales. But that doesn't mean we should throw away the data about asking price—we can use it as one of the input features. Besides the size of the house, we'll need more information: the number of rooms, bedrooms, and bathrooms; whether the kitchen and bathrooms have been recently remodeled; the age of the house and perhaps its state of repair; whether it has central heating and air conditioning; the size of the yard and the state of the landscaping.

We'll also need information about the lot and the neighborhood. But how do we define neighborhood? By zip code? What if a zip code straddles a desirable and an undesirable neighborhood? What about the school district? Should the *name* of the school district be a feature, or the *average test scores*? The ability to do a good job of feature engineering is critical to success. As Pedro Domingos (2012) says, "At the end of the day, some machine learning projects succeed and some fail. What makes the difference? Easily the most important factor is the features used."

Exploratory data analysis and visualization

John Tukey (1977) coined the term **exploratory data analysis** (EDA) for the process of exploring data in order to gain an understanding of it, not to make predictions or test hypotheses. This is done mostly with visualizations, but also with summary statistics. Looking at a few histograms or scatter plots can often help determine if data are missing or erroneous; whether your data are normally distributed or heavy-tailed; and what learning model might be appropriate.

It can be helpful to cluster your data and then visualize a prototype data point at the center of each cluster. For example, in the data set of images, I can identify that here is a cluster of cat faces; nearby is a cluster of sleeping cats; other clusters depict other objects. Expect to iterate several times between visualizing and modeling—to create clusters you need a distance function to tell you which items are near each other, but to choose a good distance function you need some feel for the data.

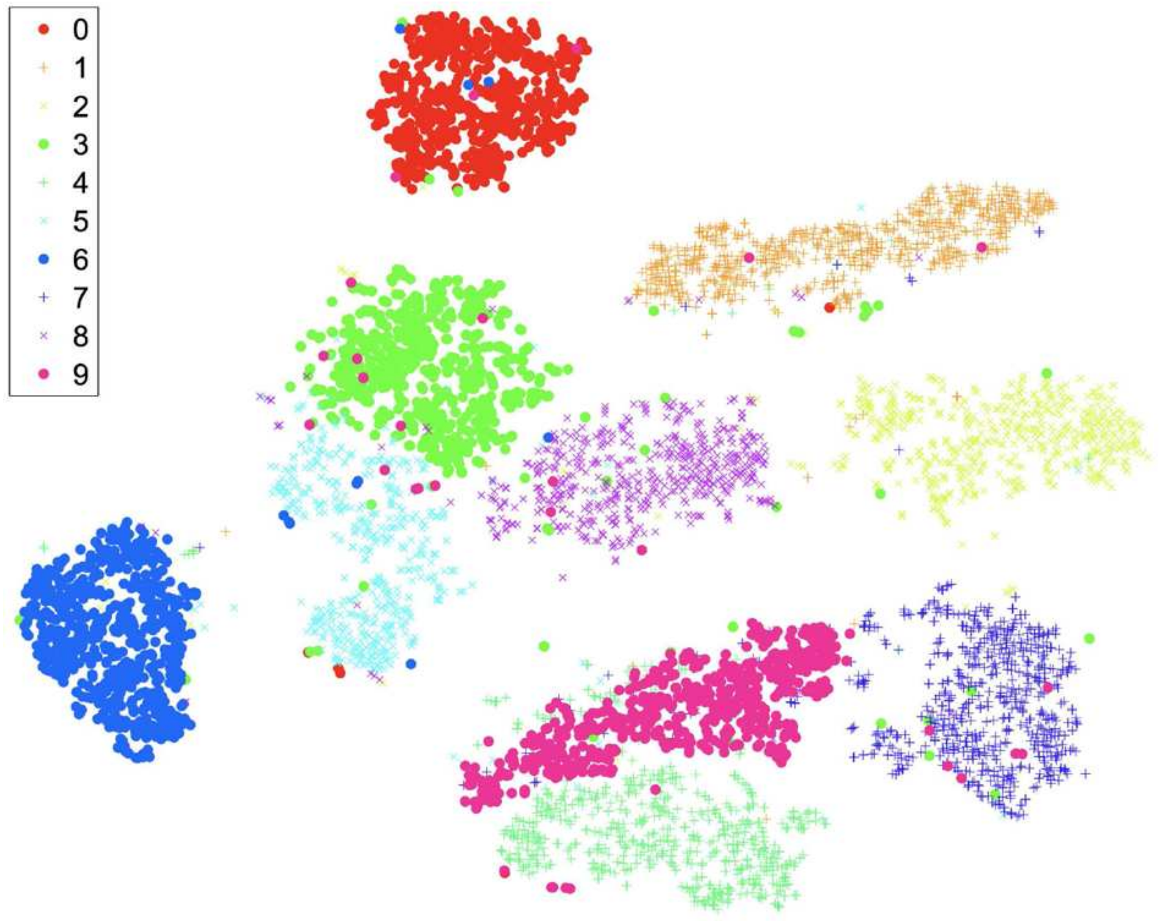
It is also helpful to detect outliers that are far from the prototypes; these can be considered **critics** of the prototype model, and can give you a feel for what type of errors your system might make. An example would be a cat wearing a lion costume.

Our computer display devices (screens or paper) are two-dimensional, which means that it is easy to visualize two-dimensional data. And our eyes are experienced at understanding three-dimensional data that has been projected down to two dimensions. But many data sets have dozens or even millions of dimensions. In order to visualize them we can do dimensionality reduction, projecting the data down to a **map** in two dimensions (or sometimes to three dimensions, which can then be explored interactively).[17]

17 Geoffrey Hinton provides the helpful advice "To deal with a 14-dimensional space, visualize a 3D space and say 'fourteen' to yourself very loudly."

The map can't maintain all relationships between data points, but should have the property that similar points in the original data set are close together in the map. A technique called **t-distributed stochastic neighbor embedding (t-SNE)** does just that. Figure 19.27 shows a t-SNE map of the MNIST digit recognition data set. Data analysis and visualization packages such as Pandas, Bokeh, and Tableau can make it easier to work with your data.

Figure 19.27



A two-dimensional t-SNE map of the MNIST data set, a collection of 60,000 images of handwritten digits, each 28 × 28 pixels and thus 784 dimensions. You can clearly see clusters for the ten digits, with a few confusions in each cluster; for example the top cluster is for the digit 0, but within the bounds of the cluster are a few data points representing the digits 3 and 6. The t-SNE algorithm finds a representation that accentuates the differences between clusters.

T-distributed stochastic neighbor embedding (t-SNE)

19.9.3 Model selection and training

With cleaned data in hand and an intuitive feel for it, it is time to build a model. That means choosing a model class (random forests? deep neural networks? an ensemble?), training your model with the training data, tuning any hyperparameters of the class (number of trees? number of layers?) with the validation data, debugging the process, and finally evaluating the model on the test data.

There is no guaranteed way to pick the best model class, but there are some rough guidelines. Random forests are good when there are a lot of categorical features and you believe that many of them may be irrelevant. Nonparametric methods are good when you have a lot of data and no prior knowledge, and when you don't want to worry too much about choosing just the right features (as long as there are fewer than 20 or so). However, nonparametric methods usually give you a function h that is more expensive to run.

Logistic regression does well when the data are linearly separable, or can be converted to be so with clever feature engineering. Support vector machines are a good method to try when the data set is not too large; they perform similarly to logistic regression on separable data and can be better for high-dimensional data. Problems dealing with pattern recognition, such as image or speech processing, are most often approached with deep neural networks (see Chapter 21).

Choosing hyperparameters can be done with a combination of experience—do what worked well in similar past problems—and search: run experiments with multiple possible values for hyperparameters. As you run more experiments you will get ideas for different models to try. However, if you measure performance on the validation data, get a new idea, and run more experiments, then you run the risk of overfitting on the validation data. If you have enough data, you may want to have several separate validation data sets to avoid this problem. This is especially true if you inspect the validation data by eye, rather than just run evaluations on it.

Suppose you are building a classifier—for example a system to classify spam email. Labeling a legitimate piece of mail as spam is called a **false positive**. There will be a tradeoff between false positives and false negatives (labeling a piece of spam as legitimate); if you want to keep more legitimate mail out of the spam folder, you will necessarily end up sending more spam to the inbox. But what is the best way to make the tradeoff? You can try different values of hyperparameters and get different rates for the two types of errors—different points on this tradeoff. A chart called the **receiver operating characteristic (ROC) curve** plots false positives versus true positives for each value of the hyperparameter, helping you visualize values that would be good choices for the tradeoff. A metric called the "area under the ROC curve" or **AUC** provides a single-number summary of the ROC curve, which is useful if you want to deploy a system and let each user choose their tradeoff point.

False positive

Receiver operating characteristic (ROC) curve

AUC

Another helpful visualization tool for classification problems is a **confusion matrix**: a two-dimensional table of counts of how often each category is classified or misclassified as each other category.

Confusion matrix

There can be tradeoffs in factors other than the loss function. If you can train a stock market prediction model that makes you $10 on every trade, that's great—but not if it costs you $20 in computation cost for each prediction. A machine translation program that runs on your phone and allows you to read signs in a foreign city is helpful—but not if it runs down the battery after an hour of use. Keep track of all the factors that lead to acceptance or rejection of your system, and design a process where you can quickly iterate the process of getting a new idea, running an experiment, and evaluating the results of the experiment to see if you have made progress. Making this iteration process fast is one of the most important factors for success in machine learning.

19.9.4 Trust, interpretability, and explainability

We have described a machine learning methodology where you develop your model with training data, choose hyperparameters with validation data, and get a final metric with test data. Doing well on that metric is a necessary but not sufficient condition for you to **trust** your model. And it is not just you—other stakeholders including regulators, lawmakers, the press, and your users are also interested in the trustworthiness of your system (as well as in related attributes such as reliability, accountability, and safety).

A machine learning system is still a piece of software, and you can build trust with all the typical tools for verifying and validating any software system:

- **SOURCE CONTROL:** Systems for version control, build, and bug/issue tracking.
- **TESTING:** Unit tests for all the components covering simple canonical cases as well as tricky adversarial cases, fuzz tests (where random inputs are generated), regression tests, load tests, and system integration tests: these are all important for any software system. For machine learning, we also have tests on the training, validation, and test data sets.
- **REVIEW:** Code walk-throughs and reviews, privacy reviews, fairness reviews (see Section 27.3.3), and other legal compliance reviews.
- **MONITORING:** Dashboards and alerts to make sure that the system is up and running and is continuing to performing at a high level of accuracy.
- **ACCOUNTABILITY:** What happens when the system is wrong? What is the process for complaining about or appealing the system's decision? How can we track who was responsible for the error? Society expects (but doesn't always get) accountability for important decisions made by banks, politicians, and the law, and they should expect accountability from software systems including machine learning systems.

In addition, there are some factors that are especially important for machine learning systems, as we shall detail below.

INTERPRETABILITY: We say that a machine learning model is **interpretable** if you can inspect the actual model and understand why it got a particular answer for a given input, and how the answer would change when the input changes.[18] Decision tree models are considered to be highly interpretable; we can understand that following the path $Patrons = Full$ and $WaitEstimate =$ 0–10 in a decision tree leads to a decision to $wait$. A decision tree is interpretable for two reasons. First, we humans have experience in understanding IF/THEN rules. (In contrast, it is very difficult for humans to get an intuitive understanding of the result of a matrix multiply followed by an activation function, as is done in some neural network models.) Second, the decision tree was in a sense constructed to be interpretable—the root of the tree was chosen to be the attribute with the highest information gain.

18 This terminology is not universally accepted; some authors use "interpretable" and "explainable" as synonyms, both referring to reaching some kind of understanding of a model.

Interpretability

Linear regression models are also considered to be interpretable; we can examine a model for predicting the rent on an apartment and see that for each bedroom added, the rent increases by $500, according to the model. This idea of "If I change X, how will the output change?" is at the core of interpretability. Of course, correlation is not causation, so interpretable models are answering *what* is the case, but not necessarily *why* it is the case.

Explainability

EXPLAINABILITY: An explainable model is one that can help you understand "*why* was this output produced for this input?" In our terminology, interpretability derives from inspecting the actual model, whereas explainability can be provided by a separate process. That is, the model itself can be a hard-to-understand black box, but an explanation module can summarize what the model does. For a neural network image-recognition system that classifies a picture as *dog*, if we tried to interpret the model directly, the best we could come away with would be something like "after processing the convolutional layers, the activation for the *dog* output in the softmax layer was higher than any other class." That's not a very compelling argument. But a separate explanation module might be able to examine the neural network model and come up with the explanation "it has four legs, fur, a tail, floppy ears, and a long snout; it is smaller than a wolf, and it is lying on a dog bed, so I think it is a dog." Explanations are one way to build trust, and some regulations such as the European GDPR (General Data Protection Regulation) require systems to provide explanations.

As an example of a separate explanation module, the local interpretable model-agnostic explanations (LIME) system works like this: no matter what model class you use, LIME builds an interpretable model—often a decision tree or linear model—that is an approximation of your model, and then interprets the linear model to create explanations that say how important each feature is. LIME accomplishes this by treating the machine-learned model as a black box, and probing it with different random input values to create a data set from which the interpretable model can be built. This approach is appropriate for structured data, but not for things like images, where each pixel is a feature, and no one pixel is "important" by itself.

Sometimes we choose a model class because of its explainability—we might choose decision trees over neural networks not because they have higher accuracy but because the explainability gives us more trust in them.

However, a simple explanation can lead to a false sense of security. After all, we typically choose to use a machine learning model (rather than a hand-written traditional program) because the problem we are trying to solve is inherently complex, and we don't know how to write a traditional program. In that case, we shouldn't expect that there will necessarily be a simple explanation for every prediction.

If you are building a machine learning model primarily for the purpose of understanding the domain, then interpretability and explainability will help you arrive at that understanding. But if you just want the best-performing piece of software then testing may give you more confidence and trust than explanations. Which would you trust: an experimental aircraft that has never flown before but has a detailed explanation of why it is safe, or an aircraft that safely completed 100 previous flights and has been carefully maintained, but comes with no guaranted explanation?

19.9.5 Operation, monitoring, and maintenance

Once you are happy with your model's performance, you can deploy it to your users. You'll face additional challenges. First, there is the problem of the **long tail** of user inputs. You may have tested your system on a large test set, but if your system is popular, you will soon see inputs that were never tested before. You need to know whether your model generalizes well for them, which means you need to **monitor** your performance on live data—tracking statistics, displaying a dashboard, and sending alerts when key metrics fall below a threshold. In addition to automatically updating statistics on user interactions, you may need to hire and train human raters to look at your system and grade how well it is doing.

Long tail

Monitoring

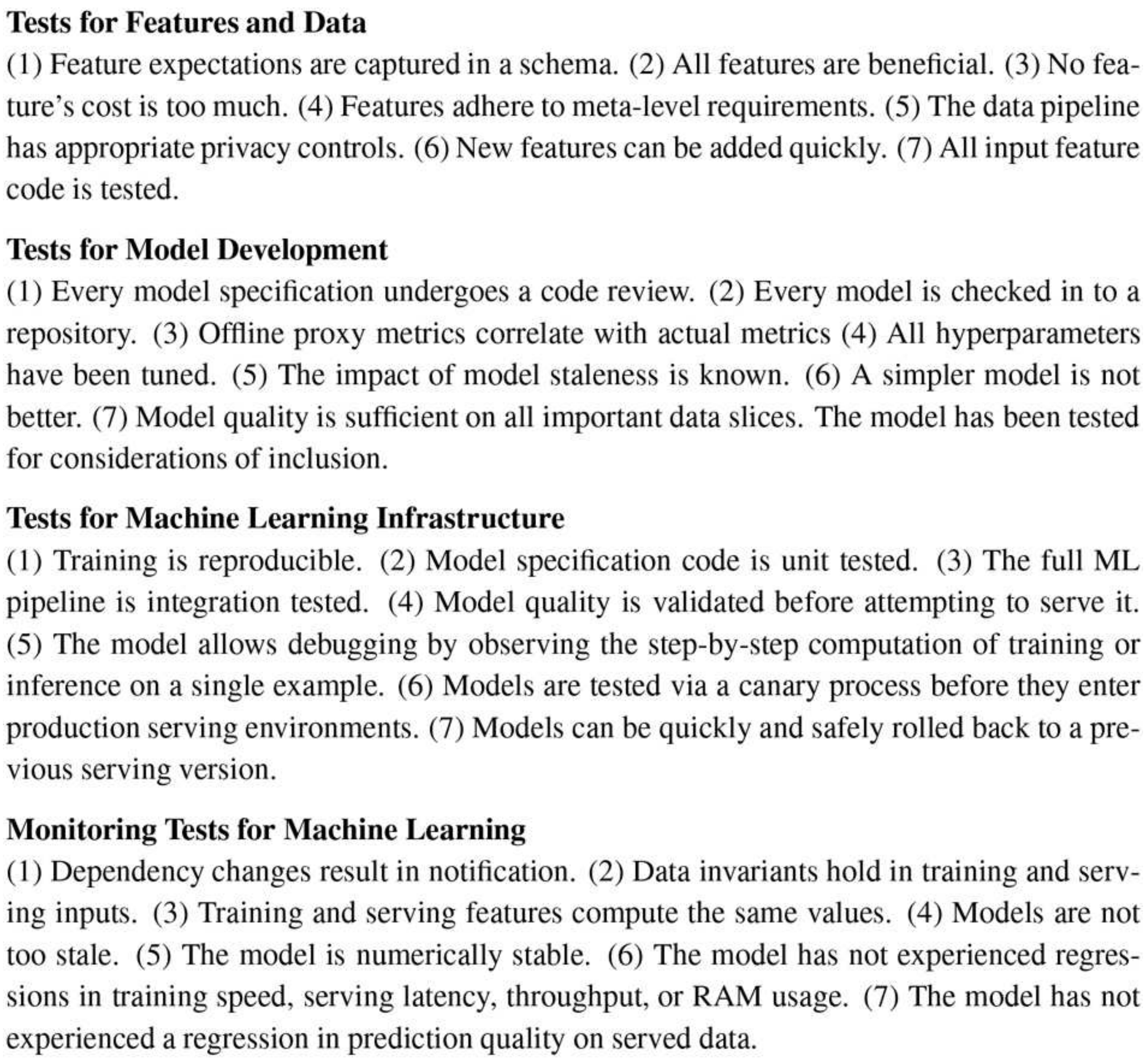
Second, there is the problem of **nonstationarity**—the world changes over time. Suppose your system classifies email as spam or non-spam. As soon as you successfully classify a batch of spam messages, the spammers will see what you have done and change their tactics, sending a new type of message you haven't seen before. Non-spam also evolves, as users change the mix of email versus messaging or desktop versus mobile services that they use.

Nonstationarity

You will continually face the question of what is better: a model that has been well tested but was built from older data, versus a model that is built from the latest data but has not been tested in actual use. Different systems have different requirements for freshness: some problems benefit from a new model every day, or even every hour, while other problems can keep the same model for months. If you are deploying a new model every hour, it will be impractical to run a heavy test suite and a manual review process for each update. You will need to automate the testing and release process so that small changes can be automatically approved, but larger changes trigger appropriate review. You can consider the tradeoff between an online model where new data incrementally modifies the existing model, versus an offline model where each new release requires building a new model from scratch.

It it is not just that the data will be changing—for example, new words will be used in spam email messages. It is also that the entire data schema may change—you might start out classifying spam email, and need to adapt to classify spam text messages, spam voice messages, spam videos, etc. Figure 19.28 gives a general rubric to guide the practitioner in choosing the appropriate level of testing and monitoring.

Figure 19.28



Tests for Features and Data

(1) Feature expectations are captured in a schema. (2) All features are beneficial. (3) No feature's cost is too much. (4) Features adhere to meta-level requirements. (5) The data pipeline has appropriate privacy controls. (6) New features can be added quickly. (7) All input feature code is tested.

Tests for Model Development

(1) Every model specification undergoes a code review. (2) Every model is checked in to a repository. (3) Offline proxy metrics correlate with actual metrics (4) All hyperparameters have been tuned. (5) The impact of model staleness is known. (6) A simpler model is not better. (7) Model quality is sufficient on all important data slices. The model has been tested for considerations of inclusion.

Tests for Machine Learning Infrastructure

(1) Training is reproducible. (2) Model specification code is unit tested. (3) The full ML pipeline is integration tested. (4) Model quality is validated before attempting to serve it. (5) The model allows debugging by observing the step-by-step computation of training or inference on a single example. (6) Models are tested via a canary process before they enter production serving environments. (7) Models can be quickly and safely rolled back to a previous serving version.

Monitoring Tests for Machine Learning

(1) Dependency changes result in notification. (2) Data invariants hold in training and serving inputs. (3) Training and serving features compute the same values. (4) Models are not too stale. (5) The model is numerically stable. (6) The model has not experienced regressions in training speed, serving latency, throughput, or RAM usage. (7) The model has not experienced a regression in prediction quality on served data.

A set of criteria to see how well you are doing at deploying your machine learning model with sufficient tests. Abridged from Breck *et al.* (2016), who also provide a scoring metric.

Summary

This chapter introduced machine learning, and focused on supervised learning from examples. The main points were:

- Learning takes many forms, depending on the nature of the agent, the component to be improved, and the available feedback.
- If the available feedback provides the correct answer for example inputs, then the learning problem is called **supervised learning.** The task is to learn a function $y = h(x)$. Learning a function whose output is a continuous or ordered value (like *weight*) is called **regression**; learning a function with a small number of possible output categories is called **classification**;
- We want to learn a function that not only agrees with the data but also is likely to agree with future data. We need to balance agreement with the data against simplicity of the hypothesis.
- **Decision trees** can represent all Boolean functions. The **information-gain** heuristic provides an efficient method for finding a simple, consistent decision tree.
- The performance of a learning algorithm can be visualized by a **learning curve**, which shows the prediction accuracy on the **test set** as a function of the **training set** size.
- When there are multiple models to choose from, **model selection** can pick good values of hyperparameters, as confirmed by **cross-validation** on validation data. Once the hyperparameter values are chosen, we build our best model using all the training data.
- Sometimes not all errors are equal. A **loss function** tells us how bad each error is; the goal is then to minimize loss over a validation set.
- **Computational learning theory** analyzes the sample complexity and computational complexity of inductive learning. There is a tradeoff between the expressiveness of the hypothesis space and the ease of learning.
- **Linear regression** is a widely used model. The optimal parameters of a linear regression model can be calculated exactly, or can be found by gradient descent search, which is a technique that can be applied to models that do not have a closed-form solution.
- A linear classifier with a hard threshold—also known as a **perceptron**—can be trained by a simple weight update rule to fit data that are **linearly separable**. In other cases, the rule fails to converge.

- **Logistic regression** replaces the perceptron's hard threshold with a soft threshold defined by a logistic function. Gradient descent works well even for noisy data that are not linearly separable.
- **Nonparametric models** use all the data to make each prediction, rather than trying to summarize the data with a few parameters. Examples include **nearest neighbors** and **locally weighted regression**.
- **Support vector machines** find linear separators with **maximum margin** to improve the generalization performance of the classifier. **Kernel methods** implicitly transform the input data into a high-dimensional space where a linear separator may exist, even if the original data are nonseparable.
- Ensemble methods such as **bagging** and **boosting** often perform better than individual methods. In **online learning** we can aggregate the opinions of experts to come arbitrarily close to the best expert's performance, even when the distribution of the data are constantly shifting.
- Building a good machine learning model requires experience in the complete development process, from managing data to model selection and optimization, to continued maintenance.

Chapter 21
Deep Learning

In which gradient descent learns multistep programs, with significant implications for the major subfields of artificial intelligence.

Deep learning is a broad family of techniques for machine learning in which hypotheses take the form of complex algebraic circuits with tunable connection strengths. The word "deep" refers to the fact that the circuits are typically organized into many **layers**, which means that computation paths from inputs to outputs have many steps. Deep learning is currently the most widely used approach for applications such as visual object recognition, machine translation, speech recognition, speech synthesis, and image synthesis; it also plays a significant role in reinforcement learning applications (see Chapter 22).

Deep learning

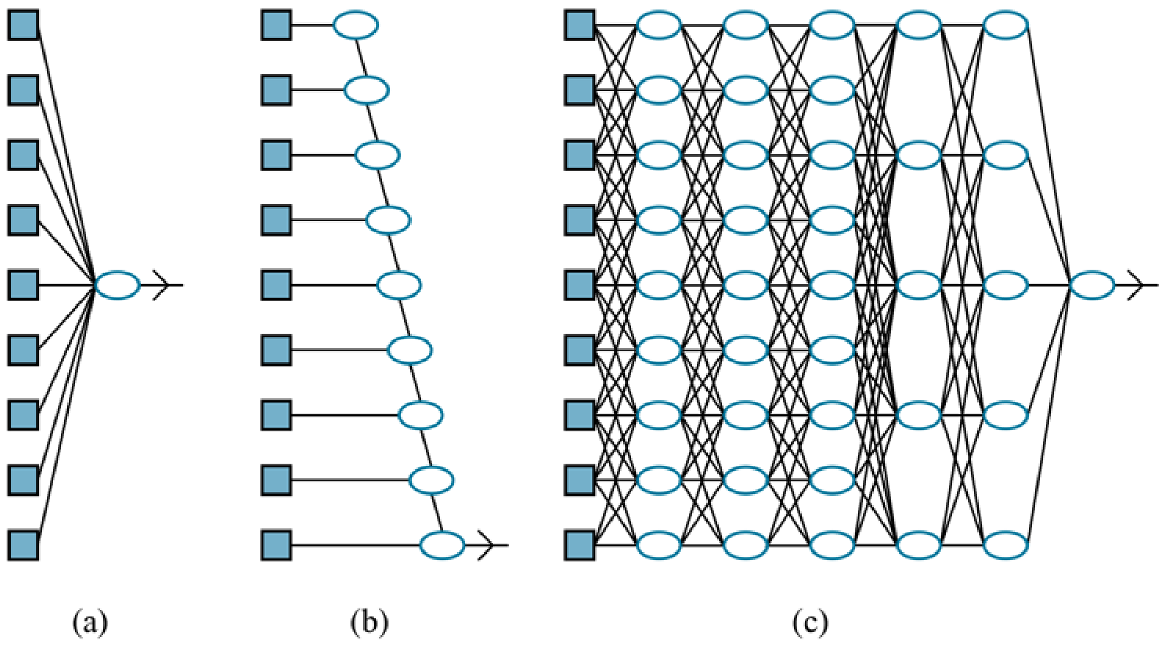
Layer

Deep learning has its origins in early work that tried to model networks of neurons in the brain (McCulloch and Pitts, 1943) with computational circuits. For this reason, the networks trained by deep learning methods are often called **neural networks**, even though the resemblance to real neural cells and structures is superficial.

Neural network

While the true reasons for the success of deep learning have yet to be fully elucidated, it has self-evident advantages over some of the methods covered in Chapter 19—particularly for high-dimensional data such as images. For example, although methods such as linear and logistic regression can handle a large number of input variables, the computation path from each input to the output is very short: multiplication by a single weight, then adding into the aggregate output. Moreover, the different input variables contribute independently to the output, without interacting with each other (Figure 21.1(a)). This significantly limits the expressive power of such models. They can represent only linear functions and boundaries in the input space, whereas most real-world concepts are far more complex.

Figure 21.1



(a) A shallow model, such as linear regression, has short computation paths between inputs and output. (b) A decision list network (page 674) has some long paths for some possible input values, but most paths are short. (c) A deep learning network has longer computation paths, allowing each variable to interact with all the others.

Decision lists and decision trees, on the other hand, allow for long computation paths that can depend on many input variables—but only for a relatively small fraction of the possible input vectors (Figure 21.1(b)). If a decision tree has long computation paths for a significant fraction of the possible inputs, it must be exponentially large in the number of input variables. The basic idea of deep learning is to train circuits such that the computation paths are long, allowing all the input variables to interact in complex ways (Figure

21.1(c)). These circuit models turn out to be sufficiently expressive to capture the complexity of real-world data for many important kinds of learning problems.

Section 21.1 describes simple feedforward networks, their components, and the essentials of learning in such networks. Section 21.2 goes into more detail on how deep networks are put together, and Section 21.3 covers a class of networks called convolutional neural networks that are especially important in vision applications. Sections 21.4 and 21.5 go into more detail on algorithms for training networks from data and methods for improving generalization. Section 21.6 covers networks with recurrent structure, which are well suited for sequential data. Section 21.7 describes ways to use deep learning for tasks other than supervised learning. Finally, Section 21.8 surveys the range of applications of deep learning.

21.1 Simple Feedforward Networks

A **feedforward network**, as the name suggests, has connections only in one direction—that is, it forms a directed acyclic graph with designated input and output nodes. Each node computes a function of its inputs and passes the result to its successors in the network. Information flows through the network from the input nodes to the output nodes, and there are no loops. A **recurrent network**, on the other hand, feeds its intermediate or final outputs back into its own inputs. This means that the signal values within the network form a dynamical system that has internal state or memory. We will consider recurrent networks in Section 21.6.

Feedforward network

Recurrent network

Boolean circuits, which implement Boolean functions, are an example of feedforward networks. In a Boolean circuit, the inputs are limited to 0 and 1, and each node implements a simple Boolean function of its inputs, producing a 0 or a 1. In neural networks, input values are typically continuous, and nodes take continuous inputs and produce continuous outputs. Some of the inputs to nodes are **parameters** of the network; the network learns by adjusting the values of these parameters so that the network as a whole fits the training data.

21.1.1 Networks as complex functions

Unit

Each node within a network is called a **unit**. Traditionally, following the design proposed by McCulloch and Pitts, a unit calculates the weighted sum of the inputs from predecessor nodes and then applies a nonlinear function to produce its output. Let a_j denote the output of unit j and let $w_{i,j}$ be the weight attached to the link from unit i to unit j; then we have

$$a_j = g_j\left(\textstyle\sum_i w_{i,j} a_i\right) \equiv g_j(in_j),$$

where g_j is a nonlinear **activation function** associated with unit j and in_j is the weighted sum of the inputs to unit j.

Activation function

As in Section 19.6.3 (page 679), we stipulate that each unit has an extra input from a dummy unit 0 that is fixed to $+1$ and a weight $w_{0,j}$ for that input. This allows the total weighted input in_j to unit j to be nonzero even when the outputs of the preceding layer are all zero. With this convention, we can write the preceding equation in vector form:

(21.1)

$$a_j = g_j(\mathbf{w}^\top \mathbf{x})$$

where $\mathbf{w}$ is the vector of weights leading into unit j (including $w_{0,j}$) and $\mathbf{x}$ is the vector of inputs to unit j (including the $+1$).

The fact that the activation function is nonlinear is important because if it were not, any composition of units would still represent a linear function. The nonlinearity is what allows sufficiently large networks of units to represent arbitrary functions. The **universal approximation** theorem states that a network with just two layers of computational units, the first nonlinear and the second linear, can approximate any continuous function to an arbitrary degree of accuracy. The proof works by showing that an exponentially large network can represent exponentially many "bumps" of different heights at different locations in the input space, thereby approximating the desired function. In other words, sufficiently large networks can implement a lookup table for continuous functions, just as sufficiently large decision trees implement a lookup table for Boolean functions.

A variety of different activation functions are used. The most common are the following:

- The logistic or **sigmoid** function, which is also used in logistic regression (see page 685):

$$\sigma(x) = 1/(1+e^{-x}).$$

Sigmoid

- The **ReLU** function, whose name is an abbreviation for **rectified linear unit**:

$$\mathrm{ReLU}(x) = \max(0{,}x).$$

ReLU

- The **softplus** function, a smooth version of the ReLU function:

$$\mathrm{softplus}(x) = \log(1+e^{x}).$$

Softplus

  The derivative of the softplus function is the sigmoid function.
- The **tanh** function:

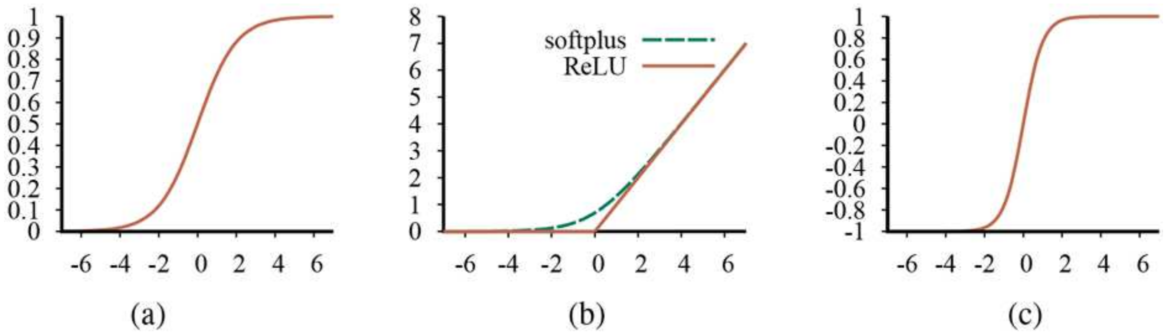
$$\tanh(x) = \tfrac{e^{2x}-1}{e^{2x}+1}.$$

Tanh

Note that the range of tanh is $(-1, +1)$. Tanh is a scaled and shifted version of the sigmoid, as $\tanh(x) = 2\sigma(2x) - 1$.

These functions are shown in Figure 21.2. Notice that all of them are monotonically nondecreasing, which means that their derivatives g' are nonnegative. We will have more to say about the choice of activation function in later sections.

Figure 21.2



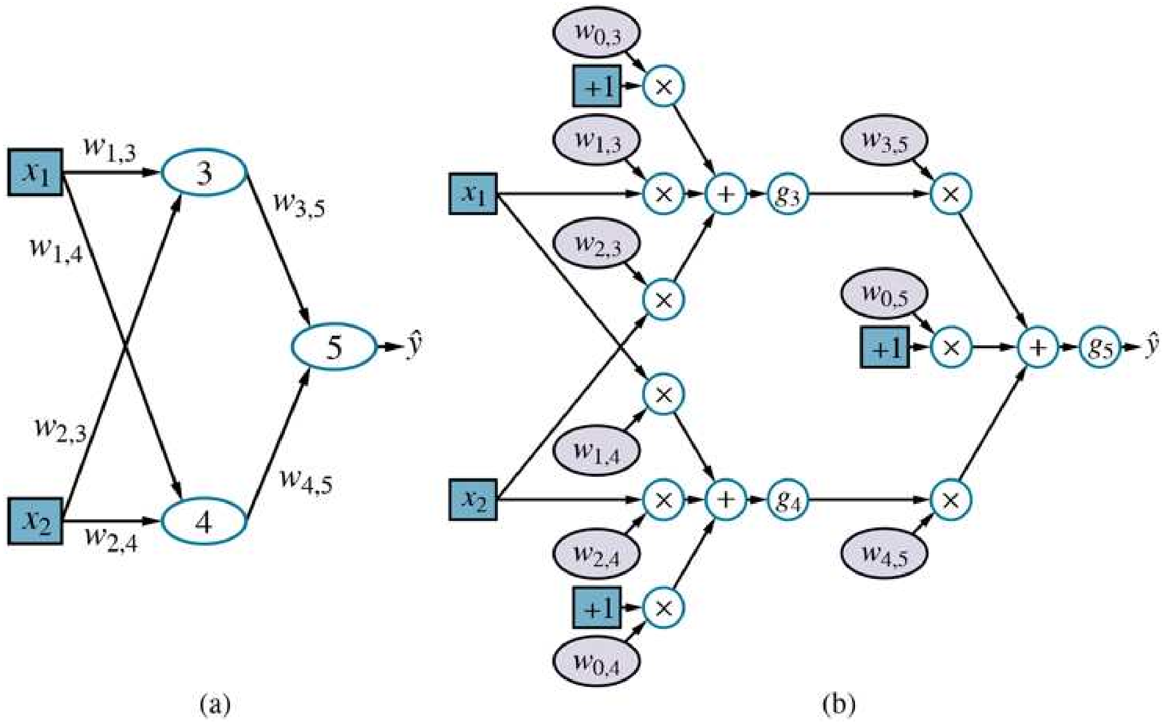
Activation functions commonly used in deep learning systems: (a) the logistic or sigmoid function; (b) the ReLU function and the softplus function; (c) the tanh function.

Coupling multiple units together into a network creates a complex function that is a composition of the algebraic expressions represented by the individual units. For example, the network shown in Figure 21.3(a) represents a function $h_{\mathbf{w}}(\mathbf{x})$, parameterized by weights $\mathbf{w}$, that maps a two-element input vector $\mathbf{x}$ to a scalar output value $\hat{y}$. The internal structure of the function mirrors the structure of the network. For example, we can write an expression for the output $\hat{y}$ as follows:

(21.2)

$$\begin{aligned}\hat{y} &= g_5(in_5) = g_5(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4)\\ &= g_5(w_{0,5} + w_{3,5}g_3(in_3) + w_{4,5}g_4(in_4))\\ &= g_5(w_{0,5} + w_{3,5}g_3(w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2)\\ &\qquad\qquad + w_{4,5}g_4(w_{0,4} + w_{1,4}x_1 + w_{2,4}x_2)).\end{aligned}$$

Figure 21.3



(a) A neural network with two inputs, one hidden layer of two units, and one output unit. Not shown are the dummy inputs and their associated weights. (b) The network in (a) unpacked into its full computation graph.

Thus, we have the output $\hat{y}$ expressed as a function $h_{\mathbf{w}}(\mathbf{x})$ of the inputs and the weights.

Figure 21.3(a) shows the traditional way a network might be depicted in a book on neural networks. A more general way to think about the network is as a **computation graph** or **dataflow graph**—essentially a circuit in which each node represents an elementary computation. Figure 21.3(b) shows the computation graph corresponding to the network in Figure 21.3(a); the graph makes each element of the overall computation explicit. It also distinguishes between the inputs (in blue) and the weights (in light mauve): the weights can be adjusted to make the output $\hat{y}$ agree more closely with the true value y in the training data. Each weight is like a volume control knob that determines how much the next node in the graph hears from that particular predecessor in the graph.

Computation graph

Dataflow graph

Just as Equation (21.1) described the operation of a unit in vector form, we can do something similar for the network as a whole. We will generally use $\mathbf{W}$ to denote a weight matrix; for this network, $\mathbf{W}^{(1)}$ denotes the weights in the first layer ($w_{1,3}$, $w_{1,4}$, etc.) and $\mathbf{W}^{(2)}$ denotes the weights in the second layer ($w_{3,5}$ etc.). Finally, let $\mathbf{g}^{(1)}$ and $\mathbf{g}^{(2)}$ denote the activation functions in the first and second layers. Then the entire network can be written as follows:

(21.3)

$$h_{\mathbf{w}}(\mathbf{x}) = \mathbf{g}^{(2)}(\mathbf{W}^{(2)}\mathbf{g}^{(1)}(\mathbf{W}^{(1)}\mathbf{x})).$$

Like Equation (21.2), this expression corresponds to a computation graph, albeit a much simpler one than the graph in Figure 21.3(b): here, the graph is simply a chain with weight matrices feeding into each layer.

The computation graph in Figure 21.3(b) is relatively small and shallow, but the same idea applies to all forms of deep learning: we construct computation graphs and adjust their weights to fit the data. The graph in Figure 21.3(b) is also **fully connected**, meaning that every node in each layer is connected to every node in the next layer. This is in some sense the default, but we will see in Section 21.3 that choosing the connectivity of the network is also important in achieving effective learning.

Fully connected

21.1.2 Gradients and learning

In Section 19.6, we introduced an approach to supervised learning based on **gradient descent**: calculate the gradient of the loss function with respect to the weights, and adjust the weights along the gradient direction to reduce the loss. (If you have not already read

Section 19.6, we recommend strongly that you do so before continuing.) We can apply exactly the same approach to learning the weights in computation graphs. For the weights leading into units in the **output layer**—the ones that produce the output of the network, the gradient calculation is essentially identical to the process in Section 19.6. For weights leading into units in the **hidden layers**, which are not directly connected to the outputs, the process is only slightly more complicated.

Output layer

Hidden layer

For now, we will use the squared loss function, L_2, and we will calculate the gradient for the network in Figure 21.3 with respect to a single training example $(\mathbf{x}, y)$. (For multiple examples, the gradient is just the sum of the gradients for the individual examples.) The network outputs a prediction $\hat{y} = h_{\mathbf{w}}(\mathbf{x})$ and the true value is y, so we have

$$Loss(h_{\mathbf{w}}) = L_2(y, h_{\mathbf{w}}(\mathbf{x})) = \| y - h_{\mathbf{w}}(\mathbf{x}) \|^2 = (y - \hat{y})^2.$$

To compute the gradient of the loss with respect to the weights, we need the same tools of calculus we used in Chapter 19—principally the **chain rule**, $\partial g(f(x))/\partial x = g'(f(x))\partial f(x)/\partial x$. We'll start with the easy case: a weight such as $w_{3,5}$ that is connected to the output unit. We operate directly on the network-defining expressions from Equation (21.2):

(21.4)

$$\begin{aligned}\frac{\partial}{\partial w_{3,5}} Loss(h_{\mathbf{w}}) &= \frac{\partial}{\partial w_{3,5}}(y-\hat{y})^2 = -2(y-\hat{y})\frac{\partial \hat{y}}{\partial w_{3,5}} \\ &= -2(y-\hat{y})\frac{\partial}{\partial w_{3,5}} g_5(in_5) = -2(y-\hat{y})g_5'(in_5)\frac{\partial}{\partial w_{3,5}} in_5 \\ &= -2(y-\hat{y})g_5'(in_5)\frac{\partial}{\partial w_{3,5}}\left(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4\right) \\ &= -2(y-\hat{y})g_5'(in_5)a_3.\end{aligned}$$

The simplification in the last line follows because $w_{0,5}$ and $w_{4,5}a_4$ do not depend on $w_{3,5}$, nor does the coefficient of $w_{3,5}$, a_3.

The slightly more difficult case involves a weight such as $w_{1,3}$ that is not directly connected to the output unit. Here, we have to apply the chain rule one more time. The first few steps are identical, so we omit them:

(21.5)

$$\begin{aligned}\frac{\partial}{\partial w_{1,3}} Loss(h_{\mathbf{w}}) &= -2(y-\hat{y})g_5'(in_5)\frac{\partial}{\partial w_{1,3}}\left(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4\right) \\ &= -2(y-\hat{y})g_5'(in_5)w_{3,5}\frac{\partial}{\partial w_{1,3}} a_3 \\ &= -2(y-\hat{y})g_5'(in_5)w_{3,5}\frac{\partial}{\partial w_{1,3}} g_3(in_3) \\ &= -2(y-\hat{y})g_5'(in_5)w_{3,5}g_3'(in_3)\frac{\partial}{\partial w_{1,3}} in_3 \\ &= -2(y-\hat{y})g_5'(in_5)w_{3,5}g_3'(in_3)\frac{\partial}{\partial w_{1,3}}\left(w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2\right) \\ &= -2(y-\hat{y})g_5'(in_5)w_{3,5}g_3'(in_3)x_1.\end{aligned}$$

So, we have fairly simple expressions for the gradient of the loss with respect to the weights $w_{3,5}$ and $w_{1,3}$.

If we define $\Delta_5 = 2(\hat{y} - y)g_5'(in_5)$ as a sort of "perceived error" at the point where unit 5 receives its input, then the gradient with respect to $w_{3,5}$ is just $\Delta_5 a_3$. This makes perfect sense: if Δ_5 is positive, that means $\hat{y}$ is too big (recall that g' is always nonnegative); if a_3 is also positive, then increasing $w_{3,5}$ will only make things worse, whereas if a_3 is negative, then increasing $w_{3,5}$ will reduce the error. The magnitude of a_3 also matters: if a_3 is small for this training example, then $w_{3,5}$ didn't play a major role in producing the error and doesn't need to be changed much.

If we also define $\Delta_3 = \Delta_5 w_{3,5} g'_3(in_3)$, then the gradient for $w_{1,3}$ becomes just $\Delta_3 x_1$. Thus, the perceived error at the input to unit 3 is the perceived error at the input to unit 5, multiplied by information along the path from 5 back to 3. This phenomenon is completely general, and gives rise to the term **back-propagation** for the way that the error at the output is passed back through the network.

Back-propagation

Another important characteristic of these gradient expressions is that they have as factors the local derivatives $g'_j(in_j)$. As noted earlier, these derivatives are always nonnegative, but they can be very close to zero (in the case of the sigmoid, softplus, and tanh functions) or exactly zero (in the case of ReLUs), if the inputs from the training example in question happen to put unit j in the flat operating region. If the derivative g'_j is small or zero, that means that changing the weights leading into unit j will have a negligible effect on its output. As a result, deep networks with many layers may suffer from a **vanishing gradient**—the error signals are extinguished altogether as they are propagated back through the network. Section 21.3.3 provides one solution to this problem.

Vanishing gradient

We have shown that gradients in our tiny example network are simple expressions that can be computed by passing information back through the network from the output units. It turns out that this property holds more generally. In fact, as we show in Section 21.4.1, the gradient computations for *any* feedforward computation graph have the same structure as the underlying computation graph. This property follows straightforwardly from the rules of differentiation.

We have shown the gory details of a gradient calculation, but worry not: there is no need to redo the derivations in Equations (21.4) and (21.5) for each new network structure! All such gradients can be computed by the method of **automatic differentiation**, which applies the rules of calculus in a systematic way to calculate gradients for any numeric program.[1] In fact, the method of back-propagation in deep learning is simply an application of **reverse mode** differentiation, which applies the chain rule "from the outside in" and gains the efficiency advantages of dynamic programming when the network in question has many inputs and relatively few outputs.

1 Automatic differentiation methods were originally developed in the 1960s and 1970s for optimizing the parameters of systems defined by large, complex Fortran programs.

Automatic differentiation

Reverse mode

All of the major packages for deep learning provide automatic differentiation, so that users can experiment freely with different network structures, activation functions, loss functions, and forms of composition without having to do lots of calculus to derive a new learning algorithm for each experiment. This has encouraged an approach called **end-to-end learning**, in which a complex computational system for a task such as machine translation can be composed from several trainable subsystems; the entire system is then trained in an end-to-end fashion from input/output pairs. With this approach, the designer need have only a vague idea about how the overall system should be structured; there is no need to know in advance exactly what each subsystem should do or how to label its inputs and outputs.

End-to-end learning