The key advantage of differentiable functions is that we can use the slope of the function at a particular point as a guide to find places where the function is higher or lower than our current position. This allows us to find the *minima* of the function. The *derivative* of differentiable function f, denoted f', is another function that provides the slope of the original function at all points. The conceptual idea is that the derivative of a function at a given point gives a signpost pointing to directions where the function is higher or lower than its current value. An optimization algorithm can follow this signpost to move closer to a minima of f. At the minima itself, the function will have derivative zero.

The power of derivative-driven optimization isn't apparent at first. Generations of calculus students have suffered through stultifying exercises minimizing tiny functions on paper. These exercises aren't useful since finding the minima of a function with only a small number of input parameters is a trivial exercise best done graphically. The power of derivative-driven optimization only becomes evident when there are hundreds, thousands, millions, or billions of variables. At these scales, understanding the function analytically is nigh impossible, and all visualizations are fraught exercises that may well miss the key attributes of the function. At these scales, the *gradient* of the function ∇f, a generalization of f' to multivariate functions, is likely the most powerful mathematical tool to understand the function and its behavior. We will dig into gradients in more depth later in this chapter. (Conceptually that is; we won't cover the technical details of gradients in this work.)

At a very high level, machine learning is simply the act of function minimization: learning algorithms are nothing more than minima finders for suitably defined functions. This definition has the advantage of mathematical simplicity. But, what are these special differentiable functions that encode useful solutions in their minima and how can we find them?

Loss Functions

In order to solve a given machine learning problem, a data scientist must find a way of constructing a function whose minima encode solutions to the real-world problem at hand. Luckily for our hapless data scientist, the machine learning literature has built up a rich history of *loss functions* that perform such encodings. Practical machine learning boils down to understanding the different types of loss functions available and knowing which loss function should be applied to which problems. Put another way, the loss function is the mechanism by which a data science project is transmuted into mathematics. All of machine learning, and much of artificial intelligence, boils down to the creation of the right loss function to solve the problem at hand. We will give you a whirlwind tour of some common families of loss functions.

We start by noting that a loss function $\mathscr{L}$ must satisfy some mathematical properties to be meaningful. First $\mathscr{L}$ must use both datapoints x and labels y. We denote this by

writing the loss function as $\mathscr{L}(x, y)$. Using our language from the previous chapter, both x and y are tensors, and $\mathscr{L}$ is a function from pairs of tensors to scalars. What should the functional form of the loss function be? A common assumption that people use is to make loss functions *additive*. Suppose that (x_i, y_i) are the data available for example i and that there are N total examples. Then the loss function can be decomposed as

$$\mathscr{L}(x, y) = \sum_{i=1}^{N} \mathscr{L}_i(x_i, y_i)$$

(In practice $\mathscr{L}_i$ is the same for every datapoint.) This additive decomposition allows for many useful advantages. The first is that derivatives factor through addition, so computing the gradient of the total loss simplifies as follows:

$$\nabla\mathscr{L}(x, y) = \sum_{i=1}^{N} \nabla\mathscr{L}_i(x_i, y_i)$$

This mathematical trick means that so long as the smaller functions $\mathscr{L}_i$ are differentiable, so too will the total loss function be. It follows that the problem of designing loss functions resolves into the problem of designing smaller functions $\mathscr{L}_i(x_i, y_i)$. Before we dive into designing the $\mathscr{L}_i$, it will be convenient to take a small detour that explains the difference between classification and regression problems.

Classification and regression

Machine learning algorithms can be broadly categorized as supervised or unsupervised problems. Supervised problems are those for which both datapoints x and labels y are available, while unsupervised problems have only datapoints x without labels y. In general, unsupervised machine learning is much harder and less well-defined (what does it mean to "understand" datapoints x?). We won't delve into unsupervised loss functions at this point since, in practice, most unsupervised losses are cleverly repurposed supervised losses.

Supervised machine learning can be broken up into the two subproblems of classification and regression. A classification problem is one in which you seek to design a machine learning system that assigns a discrete label, say 0/1 (or more generally $0, \cdots, n$) to a given datapoint. Regression is the problem of designing a machine learning system that attaches a real valued label (in $\mathbb{R}$) to a given datapoint.

At a high level, these problems may appear rather different. Discrete objects and continuous objects are typically treated differently by mathematics and common sense. However, part of the trickery used in machine learning is to use continuous, differen-

tiable loss functions to encode both classification and regression problems. As we've mentioned previously, much of machine learning is simply the art of turning complicated real-world systems into suitably simple differentiable functions.

In the following sections, we will introduce you to a pair of mathematical functions that will prove very useful for transforming classification and regression tasks into suitable loss functions.

L^2 Loss

The L^2 loss (pronounced *ell-two* loss) is commonly used for regression problems. The L^2 loss (or L^2-norm as it's commonly called elsewhere) provides for a measure of the magnitude of a vector:

$$\| a \|_2 = \sqrt{\sum_{i=1}^{N} a_i^2}$$

Here, a is assumed to be a vector of length N. The L^2 norm is commonly used to define the distance between two vectors:

$$\| a - b \|_2 = \sqrt{\sum_{i=1}^{N} (a_i - b_i)^2}$$

This idea of L^2 as a distance measurement is very useful for solving regression problems in supervised machine learning. Suppose that x is a collection of data and y the associated labels. Let f be some differentiable function that encodes our machine learning model. Then to encourage f to predict y, we create the L^2 loss function

$$\mathscr{L}(x, y) = \| f(x) - y \|_2$$

As a quick note, it's common in practice to not use the L^2 loss directly, but rather its square

$$\| a - b \|_2^2 = \sum_{i=1}^{N} (a_i - b_i)^2$$

in order to avoid dealing with terms of the form $1/\sqrt{(x)}$ in the gradient. We will use the squared L^2 loss repeatedly in the remainder of this chapter and book.

Failure Modes of L^2 Loss

The L^2 sharply penalizes large-scale deviances from true labels, but doesn't do a great job of rewarding exact matches for real-valued labels. We can understand this discrepancy mathematically, by studying the behavior of the functions x^2 and x near the origin (Figure 3-3).

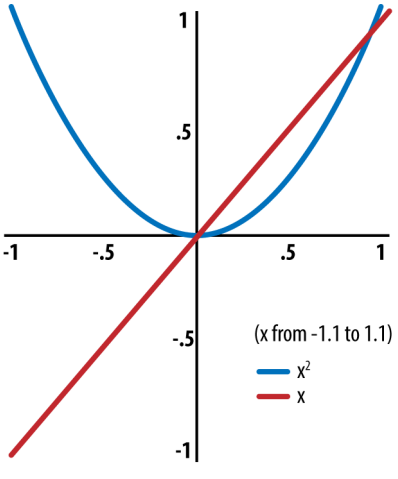

Figure 3-3. A comparison of the square and identity functions near the origin.

Notice how x^2 dwindles rapidly to 0 for small values of x. As a result, small deviations aren't penalized heavily by the L^2 loss. In low-dimensional regression, this isn't a major issue, but in high-dimensional regression, the L^2 becomes a poor loss function since there may be many small deviations that together make the regression output poor. For example, in image prediction, L^2 loss creates blurry images that are not visually appealing. Recent progress in machine learning has devised ways to learn loss functions. These learned loss functions, commonly styled Generative Adversarial Networks or GANs, are much more suitable for high-dimensional regression and are capable of generating nonblurry images.

Probability distributions

Before introducing loss functions for classification problems, it will be useful to take a quick aside to introduce probability distributions. To start, what is a probability distribution and why should we care about it for the purposes of machine learning? Probability is a deep subject, so we will only delve far enough into it for you to gain the required minimal understanding. At a high level, probability distributions provide a mathematical trick that allows you to relax a discrete set of choices into a con-

tinuum. Suppose, for example, you need to design a machine learning system that predicts whether a coin will fall heads up or heads down. It doesn't seem like heads up/down can be encoded as a continuous function, much less a differentiable one. How can you then use the machinery of calculus or TensorFlow to solve problems involving discrete choices?

Enter the probability distribution. Instead of hard choices, make the classifier predict the chance of getting heads up or heads down. For example, the classifier may learn to predict that heads has probability 0.75 and tails has probability 0.25. Note that probabilities vary continuously! Consequently by working with the probabilities of discrete events rather than with the events themselves, you can neatly sidestep the issue that calculus doesn't really work with discrete events.

A probability distribution p is simply a listing of the probabilities for the possible discrete events at hand. In this case, $p = (0.75, 0.25)$. Note, alternatively, you can view $p: \{0, 1\} \to \mathbb{R}$ as a function from the set of two elements to the real numbers. This viewpoint will be useful notationally at times.

We briefly note that the technical definition of a probability distribution is more involved. It is feasible to assign probability distributions to real-valued events. We will discuss such distributions later in the chapter.

Cross-entropy loss

Cross-entropy is a mathematical method for gauging the distance between two probability distributions:

$$H(p, q) = -\sum_x p(x) \log q(x)$$

Here p and q are two probability distributions. The notation $p(x)$ denotes the probability p accords to event x. This definition is worth discussing carefully. Like the L^2 norm, H provides a notion of distance. Note that in the case where $p = q$,

$$H(p, p) = -\sum_x p(x) \log p(x)$$

This quantity is the entropy of p and is usually written simply $H(p)$. It's a measure of how disordered the distribution is; the entropy is maximized when all events are equally likely. $H(p)$ is always less than or equal to $H(p, q)$. In fact, the "further away" distribution q is from p, the larger the cross-entropy gets. We won't dig deeply into the precise meanings of these statements, but the intuition of cross-entropy as a distance mechanism is worth remembering.

As an aside, note that unlike L^2 norm, H is asymmetric! That is, $H(p, q) \neq H(q, p)$. For this reason, reasoning with cross-entropy can be a little tricky and is best done with some caution.

Returning to concrete matters, now suppose that $p = (y, 1 - y)$ is the true data distribution for a discrete system with two outcomes, and $q = \left(y_{\text{pred}}, 1 - y_{\text{pred}}\right)$ is that predicted by a machine learning system. Then the cross-entropy loss is

$$H(p, q) = y \log y_{\text{pred}} + (1 - y) \log \left(1 - y_{\text{pred}}\right)$$

This form of the loss is used widely in machine learning systems to train classifiers. Empirically, minimizing $H(p, q)$ seems to construct classifiers that reproduce provided training labels well.

Gradient Descent

So far in this chapter, you have learned about the notion of function minimization as a proxy for machine learning. As a short recap, minimizing a suitable function is often sufficient to learn to solve a desired task. In order to use this framework, you need to use suitable loss functions, such as the L^2 or $H(p, q)$ cross-entropy in order to transform classification and regression problems into suitable loss functions.

Learnable Weights

So far in this chapter, we've explained that machine learning is the act of minimizing suitably defined loss function $\mathscr{L}(x, y)$. That is, we attempt to find arguments to the loss function $\mathscr{L}$ that minimize it. However, careful readers will recall that (x,y) are fixed quantities that cannot be changed. What arguments to $\mathscr{L}$ are we changing during learning then?

Enter learnable weights W. Suppose $f(x)$ is a differentiable function we wish to fit with our machine learning model. We will dictate that f be *parameterized* by choice of W. That is, our function actually has two arguments $f(W, x)$. Fixing the value of W results in a function that depends solely on datapoints x. These learnable weights are the quantities actually selected by minimization of the loss function. We will see later in the chapter how TensorFlow can be used to encode learnable weights using `tf.Variable`.