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`.

But now, suppose that we have encoded our learning problem with a suitable loss function? How can we actually find minima of this loss function in practice? The key trick we will use is minimization by gradient descent. Suppose that f is a function that depends on some weights W. Then ∇W denotes the direction change in W that would maximally increase f. It follows that taking a step in the opposite direction would get us closer to the minima of f.

Notation for Gradients

We have written the gradient for learnable weight W as ∇W. At times, it will be convenient to use the following alternative notation for the gradient:

$$\nabla W = \frac{\partial \mathscr{L}}{\partial W}$$

Read this equation as saying that gradient ∇W encodes the direction that maximally changes the loss $\mathscr{L}$.

TheI idea of gradient descent is to find the minima of functions by repeatedly following the negative gradient. Algorithmically, this update rule can be expressed as

$$W = W - \alpha \nabla W$$

where α is the *step-size* and dictates how much weight is given to new gradient ∇W. The idea is to take many little steps each in the direction of ∇W. Note that ∇W is itself a function of W, so the actual step changes at each iteration. Each step performs a little update to the weight matrix W. The iterative process of performing updates is typically called *learning* the weight matrix W.

Computing Gradients Efficiently with Minibatches

One issue is that computing ∇W can be very slow. Implicitly, ∇W depends on the loss function $\mathscr{L}$. Since $\mathscr{L}$ depends on the entire dataset, computing ∇W can become very slow for large datasets. In practice, people usually estimate ∇W on a fraction of the dataset called a *minibatch*. Each minibatch is of size typically 50–100. The size of the minibatch is a *hyperparameter* in a deep learning algorithm. The step-size for each step α is another hyperparameter. Deep learning algorithms typically have clusters of hyperparameters, which are not themselves learned via the stochastic gradient descent.

This tension between learnable parameters and hyperparameters is one of the weaknesses and strengths of deep architectures. The presence of hyperparameters provides much room for utilizing the expert's strong intuition, while the learnable parameters allow the data to speak for itself. However, this flexibility itself quickly becomes a weakness, with understanding of the behavior of hyperparameters something of a black art that blocks beginners from widely deploying deep learning. We will spend significant effort discussing hyperparameter optimization later in this book.

We end this section by introducing the notion of an *epoch*. An epoch is a full pass of the gradient descent algorithm over the data x. More particularly, an epoch consists of however many gradient descent steps are required to view all the data at a given minibatch size. For example, suppose that a dataset has 1,000 datapoints and training uses a minibatch of size 50. Then an epoch will consist of 20 gradient descent updates. Each epoch of training increases the amount of useful knowledge the model has gained. Mathematically, this will correspond to reductions in the value of the loss function on the training set.

Early epochs will cause dramatic drops in the loss function. This process is often referred to as *learning the prior* on that dataset. While it appears that the model is learning rapidly, it is in fact only adjusting itself to reside in the portion of parameter space that is pertinent to the problem at hand. Later epochs will correspond to much smaller drops in the loss function, but it is often in these later epochs that meaningful learning will happen. A few epochs is usually too little time for a nontrivial model to learn anything useful; models are usually trained from 10–1,000 epochs or until convergence. While this appears large, it's important to note that the number of epochs required usually doesn't scale with the size of the dataset at hand. Consequently, gradient descent scales linearly with the size of data and not quadratically! This is one of the greatest strengths of the stochastic gradient descent method versus other learning algorithms. More complicated learning algorithms may only require a single pass

over a dataset, but may use total compute that scales quadratically with the number of datapoints. In this era of big datasets, quadratic runtimes are a fatal weakness.

Tracking the drop in the loss function as a function of the number of epochs can be an extremely useful visual shorthand for understanding the learning process. These plots are often referred to as loss curves (see Figure 3-4). With time, an experienced practitioner can diagnose common failures in learning with just a quick glance at the loss curve. We will pay significant attention to the loss curves for various deep learning models over the course of this book. In particular, later in this chapter, we will introduce TensorBoard, a powerful visualization suite that TensorFlow provides for tracking quantities such as loss functions.

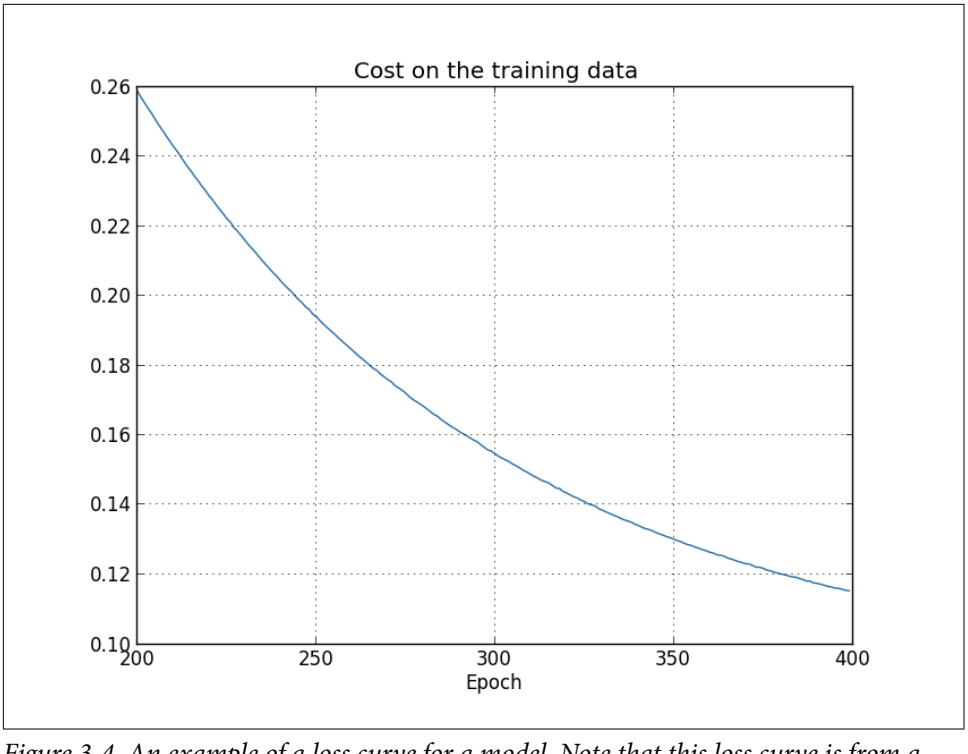

Figure 3-4. An example of a loss curve for a model. Note that this loss curve is from a model trained with the true gradient (that is, not a minibatch estimate) and is consequently smoother than other loss curves you will encounter later in this book.

Automatic Differentiation Systems

Machine learning is the art of defining loss functions suited to datasets and then minimizing them. In order to minimize loss functions, we need to compute their gradients and use the gradient descent algorithm to iteratively reduce the loss. However, we still need to discuss how gradients are actually computed. Until recently, the