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.

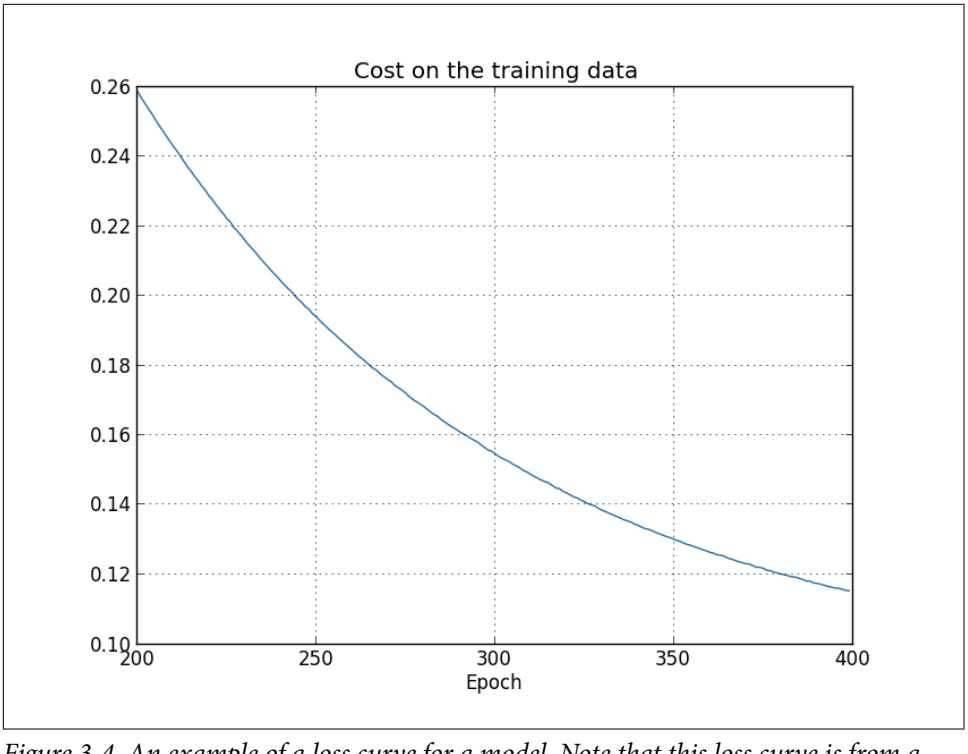

Figure 3-4. An example of a loss curve for a model. Note that this loss curve is from a model trained with the true gradient (that is, not a minibatch estimate) and is consequently smoother than other loss curves you will encounter later in this book.

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

answer was "by hand." Machine learning experts would break out pen and paper and compute matrix derivatives by hand to compute the analytical formulas for all gradients in a learning system. These formulas would then be manually coded to implement the learning algorithm. This process was notoriously buggy, and more than one machine learning expert has stories of accidental gradient errors in published papers and production systems going undiscovered for years.

This state of affairs has changed significantly with the widespread availability of automatic differentiation engines. Systems like TensorFlow are capable of automatically computing gradients for almost all loss functions. This automatic differentiation is one of the greatest advantages of TensorFlow and similar systems, since machine learning practitioners no longer need to be experts at matrix calculus. However, it's still worth understanding at a high level how TensorFlow can automatically take derivatives of complex functions. For those readers who suffered through an introductory class in calculus, you might remember that taking derivatives of functions is surprisingly mechanical. There are a series of simple rules that can be applied to take derivatives of most functions. For example:

$$\frac{d}{dx}x^n = nx^{n-1}$$

$$\frac{d}{dx}e^x = e^x$$

These rules can be combined through the power of the chain rule:

$$\frac{d}{dx}f(g(x)) = f'(g(x))g'(x)$$

where f' is used to denote the derivative of f and g' that of g. With these rules, it's straightforward to envision how one might program an automatic differentiation engine for one-dimensional calculus. Indeed, the creation of such a differentiation engine is often a first-year programming exercise in Lisp-based classes. (It turns out that correctly parsing functions is a much trickier problem than taking derivatives. Lisp makes it trivial to parse formulas using its syntax, while in other languages, waiting to do this exercise until you take a course on compilers is often easier).

How might these rules be extended to calculus of higher dimensions? Getting the math right is trickier, since there are many more numbers to consider. For example, given $X = AB$ where X, A, B are all matrices, the formula comes out to be

$$\nabla A = \frac{\partial L}{\partial A} = \frac{\partial L}{\partial X}B^T = (\nabla X)B^T$$

Formulas like this can be combined to provide a symbolic differentiation system for vectorial and tensorial calculus.

Learning with TensorFlow

In the rest of this chapter, we will cover the concepts that you need to learn basic machine learning models with TensorFlow. We will start by introducing the concept of toy datasets, and will explain how to create meaningful toy datasets using common Python libraries. Next, we will discuss new TensorFlow ideas such as placeholders, feed dictionaries, name scopes, optimizers, and gradients. The next section will show you how to use these concepts to train simple regression and classification models.

Creating Toy Datasets

In this section, we will discuss how to create simple but meaningful synthetic datasets, or toy datasets, that we will use to train simple supervised classification and regression models.

An (extremely) brief introduction to NumPy

We will make heavy use of NumPy in order to define useful toy datasets. NumPy is a Python package that allows for manipulation of tensors (called `ndarrays` in NumPy). Example 3-1 shows some basics.

Example 3-1. Some examples of basic NumPy usage

```
>>> import numpy as np
>>> np.zeros((2,2))
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> np.eye(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

You may notice that NumPy `ndarray` manipulation looks remarkably similar to TensorFlow tensor manipulation. This similarity was purposefully designed by TensorFlow's architects. Many key TensorFlow utility functions have similar arguments and forms to analogous functions in NumPy. For this purpose, we will not attempt to introduce NumPy in great depth, and will trust readers to use experimentation to work out NumPy usage. There are numerous online resources that provide tutorial introductions to NumPy.