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.

Why are toy datasets important?

In machine learning, it is often critical to learn to properly use toy datasets. Learning is challenging, and one of the most common mistakes beginners make is trying to learn nontrivial models on complex data too soon. These attempts often end in abject failure, and the would-be machine learner walks away dejected and convinced machine learning isn't for them.

The real culprit here of course isn't the student, but rather the fact that real-world datasets have many idiosyncrasies. Seasoned data scientists have learned that real-world datasets often require many clean-up and preprocessing transformations before becoming amenable to learning. Deep learning exacerbates this problem, since most deep learning models are notoriously sensitive to infelicities in data. Issues like a wide range of regression labels, or underlying strong noise patterns can throw off gradient-descent–based methods, even when other machine learning algorithms (such as random forests) would have no issues.

Luckily, it's almost always possible to deal with these issues, but doing so can require considerable sophistication on the part of the data scientist. These sensitivity issues are perhaps the biggest roadblock to the commoditization of machine learning as a technology. We will go into depth on data clean-up strategies, but for the time being, we recommend a much simpler alternative: use toy datasets!

Toy datasets are critical for understanding learning algorithms. Given very simple synthetic datasets, it is trivial to gauge whether the algorithm has learned the correct rule. On more complex datasets, this judgment can be highly challenging. Consequently, for the remainder of this chapter, we will only use toy datasets as we cover the fundamentals of gradient-descent–based learning with TensorFlow. We will dive deep into case studies with real-world data in the following chapters.

Adding noise with Gaussians

Earlier, we discussed discrete probability distributions as a tool for turning discrete choices into continuous values. We also alluded to the idea of a continuous probability distribution but didn't dive into it.

Continuous probability distributions (more accurately known as probability density functions) are a useful mathematical tool for modeling random events that may have a range of outcomes. For our purposes, it is enough to think of probability density functions as a useful tool for modeling some measurement error in gathering data. The Gaussian distribution is widely used for noise modeling.

As Figure 3-5 shows, note that Gaussians can have different *means* μ and *standard deviations* σ. The mean of a Gaussian is the average value it takes, while the standard deviation is a measure of the spread around this average value. In general, adding a Gaussian random variable onto some quantity provides a structured way to fuzz the

quantity by making it vary slighty. This is a very useful trick for coming up with non-trivial synthetic datasets.

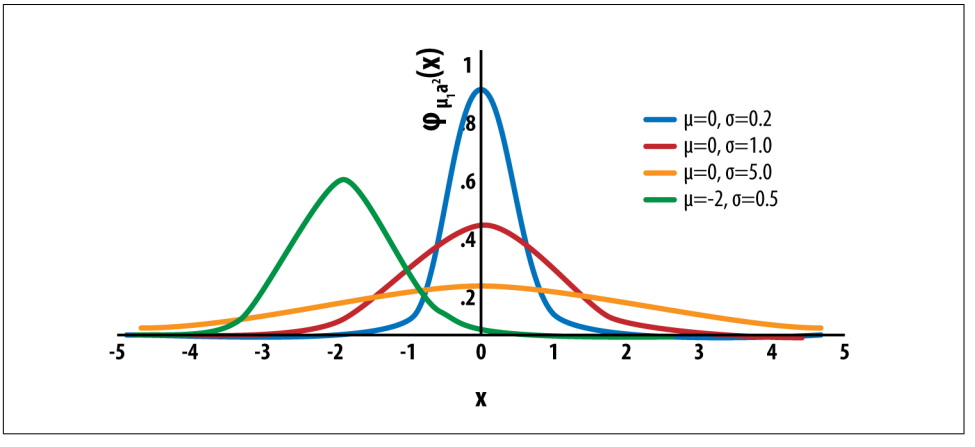

Figure 3-5. Illustrations of various Gaussian probability distributions with different means and standard deviations.

We quickly note that the Gaussian distribution is also called the Normal distribution. A Gaussian with mean μ and standard deviation σ is written $N(\mu, \sigma)$. This shorthand notation is convenient, and we will use it many times in the coming chapters.

Toy regression datasets

The simplest form of linear regression is learning the parameters for a one-dimensional line. Suppose that our datapoints x are one-dimensional. Then suppose that real-valued labels y are generated by a linear rule

$$y = wx + b$$

Here, w, b are the learnable parameters that must be estimated from data by gradient descent. In order to test that we can learn these parameters with TensorFlow, we will generate an artificial dataset consisting of points upon a straight line. To make the learning challenge a little more difficult, we will add a small amount of Gaussian noise to the dataset.

Let's write down the equation for our line perturbed by a small amount of Gaussian noise:

$$y = wx + b + N(0, \epsilon)$$

Here ϵ is the standard deviation of the noise term. We can then use NumPy to generate an artificial dataset drawn from this distribution, as shown in Example 3-2.

Example 3-2. Using NumPy to sample an artificial dataset

```
# Generate synthetic data
N = 100
w_true = 5
b_true = 2
noise_scale = .1
x_np = np.random.rand(N, 1)
noise = np.random.normal(scale=noise_scale, size=(N, 1))
# Convert shape of y_np to (N,)
y_np = np.reshape(w_true * x_np + b_true + noise, (-1))
```

We plot this dataset using Matplotlib in Figure 3-6. (you can find the code in the GitHub repo associated with this book to see the exact plotting code) to verify that synthetic data looks reasonable. As expected, the data distribution is a straight line, with a small amount of measurement error.

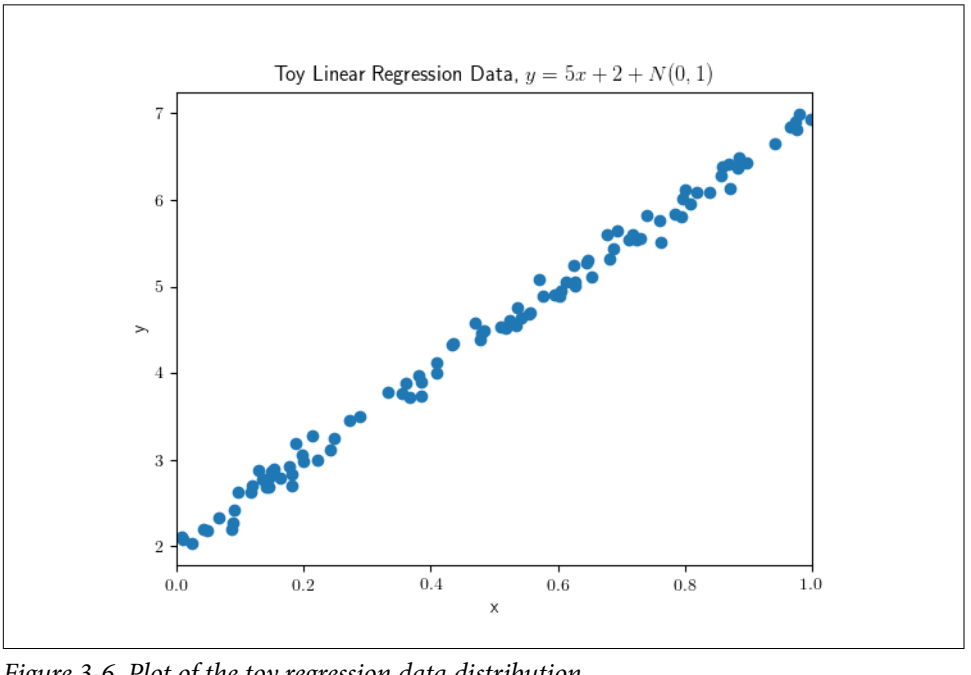

Figure 3-6. Plot of the toy regression data distribution.

Toy classification datasets

It's a little trickier to create a synthetic classification dataset. Logically, we want two distinct classes of points, which are easily separated. Suppose that the dataset consists

of only two types of points, (–1, –1) and (1, 1). Then a learning algorithm would have to learn a rule that separates these two data values.

$$y_0 = (-1, -1)$$
$$y_1 = (1, 1)$$

As before, let's make the challenge a little more difficult by adding some Gaussian noise to both types of points:

$$y_0 = (-1, -1) + N(0, \epsilon)$$
$$y_1 = (1, 1) + N(0, \epsilon)$$

However, there's a slight bit of trickiness here. Our points are two-dimensional, while the Gaussian noise we introduced previously is one-dimensional. Luckily, there exists a multivariate extension of the Gaussian. We won't discuss the intricacies of the multivariate Gaussian here, but you do not need to understand the intricacies to follow our discussion.

The NumPy code to generate the synthetic dataset in Example 3-3 is slightly trickier than that for the linear regression problem since we have to use the stacking function `np.vstack` to combine the two different types of datapoints and associate them with different labels. (We use the related function `np.concatenate` to combine the one-dimensional labels.)

Example 3-3. Sample a toy classification dataset with NumPy

```
# Generate synthetic data
N = 100
# Zeros form a Gaussian centered at (-1, -1)
# epsilon is .1
x_zeros = np.random.multivariate_normal(
    mean=np.array((-1, -1)), cov=.1*np.eye(2), size=(N/2,))
y_zeros = np.zeros((N/2,))
# Ones form a Gaussian centered at (1, 1)
# epsilon is .1
x_ones = np.random.multivariate_normal(
    mean=np.array((1, 1)), cov=.1*np.eye(2), size=(N/2,))
y_ones = np.ones((N/2,))

x_np = np.vstack([x_zeros, x_ones])
y_np = np.concatenate([y_zeros, y_ones])
```

Figure 3-7 plots the data generated by this code with Matplotlib to verify that the distribution is as expected. We see that the data resides in two classes that are neatly separated.

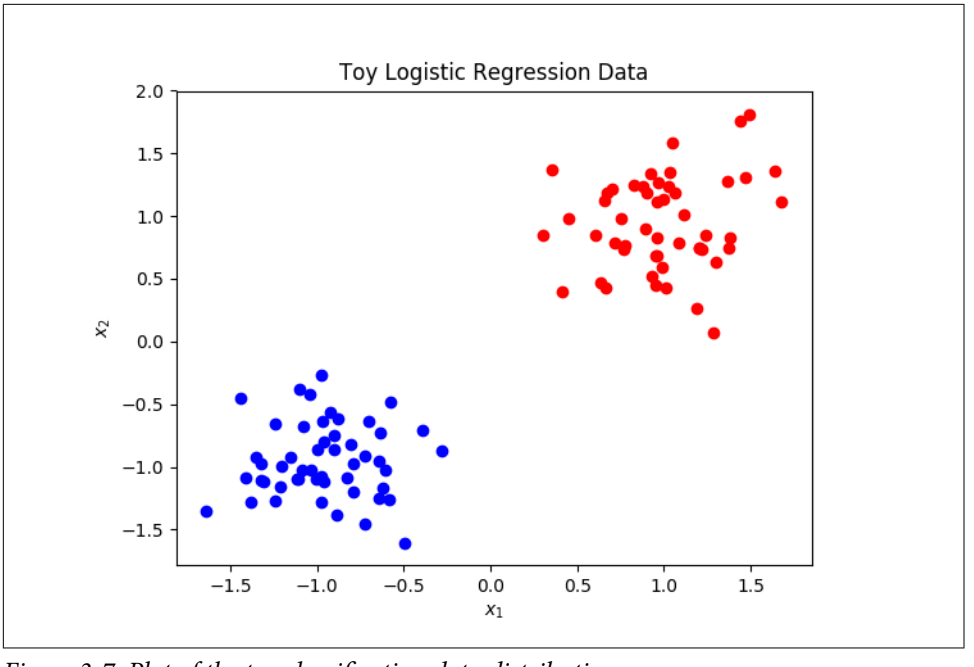

Figure 3-7. Plot of the toy classification data distribution.

New TensorFlow Concepts

Creating simple machine learning systems in TensorFlow will require that you learn some new TensorFlow concepts.

Placeholders

A placeholder is a way to input information into a TensorFlow computation graph. Think of placeholders as the input nodes through which information enters TensorFlow. The key function used to create placeholders is `tf.placeholder` (Example 3-4).

Example 3-4. Create a TensorFlow placeholder

```
>>> tf.placeholder(tf.float32, shape=(2,2))
<tf.Tensor 'Placeholder:0' shape=(2, 2) dtype=float32>
```

We will use placeholders to feed datapoints x and labels y to our regression and classification algorithms.

Feed dictionaries and Fetches

Recall that we can evaluate tensors in TensorFlow by using `sess.run(var)`. How do we feed in values for placeholders in our TensorFlow computations then? The answer