Implementing Gradient Descent

Let's try using Batch Gradient Descent (introduced in Chapter 4) instead of the Normal Equation. First we will do this by manually computing the gradients, then we will use TensorFlow's autodiff feature to let TensorFlow compute the gradients automatically, and finally we will use a couple of TensorFlow's out-of-the-box optimizers.

When using Gradient Descent, remember that it is important to first normalize the input feature vectors, or else training may be much slower. You can do this using TensorFlow, NumPy, Scikit-Learn's `StandardScaler`, or any other solution you prefer. The following code assumes that this normalization has already been done.

Manually Computing the Gradients

The following code should be fairly self-explanatory, except for a few new elements:

- The `random_uniform()` function creates a node in the graph that will generate a tensor containing random values, given its shape and value range, much like NumPy's `rand()` function.
- The `assign()` function creates a node that will assign a new value to a variable. In this case, it implements the Batch Gradient Descent step $\theta^{(\text{next step})} = \theta - \eta\nabla_\theta \text{MSE}(\theta)$.
- The main loop executes the training step over and over again (`n_epochs` times), and every 100 iterations it prints out the current Mean Squared Error (`mse`). You should see the MSE go down at every iteration.

```
n_epochs = 1000
learning_rate = 0.01

X = tf.constant(scaled_housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0), name="theta")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
gradients = 2/m * tf.matmul(tf.transpose(X), error)
training_op = tf.assign(theta, theta - learning_rate * gradients)

init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(n_epochs):
```

```
        if epoch % 100 == 0:
            print("Epoch", epoch, "MSE =", mse.eval())
        sess.run(training_op)

best_theta = theta.eval()
```

Using autodiff

The preceding code works fine, but it requires mathematically deriving the gradients from the cost function (MSE). In the case of Linear Regression, it is reasonably easy, but if you had to do this with deep neural networks you would get quite a headache: it would be tedious and error-prone. You could use *symbolic differentiation* to automatically find the equations for the partial derivatives for you, but the resulting code would not necessarily be very efficient.

To understand why, consider the function $f(x)= \exp(\exp(\exp(x)))$. If you know calculus, you can figure out its derivative $f'(x) = \exp(x) \times \exp(\exp(x)) \times \exp(\exp(\exp(x)))$. If you code $f(x)$ and $f'(x)$ separately and exactly as they appear, your code will not be as efficient as it could be. A more efficient solution would be to write a function that first computes $\exp(x)$, then $\exp(\exp(x))$, then $\exp(\exp(\exp(x)))$, and returns all three. This gives you $f(x)$ directly (the third term), and if you need the derivative you can just multiply all three terms and you are done. With the naïve approach you would have had to call the `exp` function nine times to compute both $f(x)$ and $f'(x)$. With this approach you just need to call it three times.

It gets worse when your function is defined by some arbitrary code. Can you find the equation (or the code) to compute the partial derivatives of the following function? Hint: don't even try.

```
def my_func(a, b):
    z = 0
    for i in range(100):
        z = a * np.cos(z + i) + z * np.sin(b - i)
    return z
```

Fortunately, TensorFlow's autodiff feature comes to the rescue: it can automatically and efficiently compute the gradients for you. Simply replace the `gradients = ...` line in the Gradient Descent code in the previous section with the following line, and the code will continue to work just fine:

```
gradients = tf.gradients(mse, [theta])[0]
```

The `gradients()` function takes an op (in this case `mse`) and a list of variables (in this case just `theta`), and it creates a list of ops (one per variable) to compute the gradients of the op with regards to each variable. So the `gradients` node will compute the gradient vector of the MSE with regards to `theta`.