There are four main approaches to computing gradients automatically. They are summarized in Table 9-2. TensorFlow uses *reverse-mode autodiff*, which is perfect (efficient and accurate) when there are many inputs and few outputs, as is often the case in neural networks. It computes all the partial derivatives of the outputs with regards to all the inputs in just $n_{\text{outputs}} + 1$ graph traversals.

*Table 9-2. Main solutions to compute gradients automatically*

| Technique | Nb of graph traversals to compute all gradients | Accuracy | Supports arbitrary code | Comment |
|---|---|---|---|---|
| Numerical differentiation | $n_{\text{inputs}} + 1$ | Low | Yes | Trivial to implement |
| Symbolic differentiation | N/A | High | No | Builds a very different graph |
| Forward-mode autodiff | $n_{\text{inputs}}$ | High | Yes | Uses *dual numbers* |
| Reverse-mode autodiff | $n_{\text{outputs}} + 1$ | High | Yes | Implemented by TensorFlow |

If you are interested in how this magic works, check out Appendix D.

## Using an Optimizer

So TensorFlow computes the gradients for you. But it gets even easier: it also provides a number of optimizers out of the box, including a Gradient Descent optimizer. You can simply replace the preceding `gradients = ...` and `training_op = ...` lines with the following code, and once again everything will just work fine:

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(mse)
```

If you want to use a different type of optimizer, you just need to change one line. For example, you can use a momentum optimizer (which often converges much faster than Gradient Descent; see Chapter 11) by defining the optimizer like this:

```
optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate,
                                       momentum=0.9)
```

# Feeding Data to the Training Algorithm

Let's try to modify the previous code to implement Mini-batch Gradient Descent. For this, we need a way to replace `X` and `y` at every iteration with the next mini-batch. The simplest way to do this is to use placeholder nodes. These nodes are special because they don't actually perform any computation, they just output the data you tell them to output at runtime. They are typically used to pass the training data to TensorFlow during training. If you don't specify a value at runtime for a placeholder, you get an exception.

To create a placeholder node, you must call the `placeholder()` function and specify the output tensor's data type. Optionally, you can also specify its shape, if you want to

enforce it. If you specify `None` for a dimension, it means "any size." For example, the following code creates a placeholder node `A`, and also a node `B = A + 5`. When we evaluate `B`, we pass a `feed_dict` to the `eval()` method that specifies the value of `A`. Note that `A` must have rank 2 (i.e., it must be two-dimensional) and there must be three columns (or else an exception is raised), but it can have any number of rows.

```
>>> A = tf.placeholder(tf.float32, shape=(None, 3))
>>> B = A + 5
>>> with tf.Session() as sess:
...     B_val_1 = B.eval(feed_dict={A: [[1, 2, 3]]})
...     B_val_2 = B.eval(feed_dict={A: [[4, 5, 6], [7, 8, 9]]})
...
>>> print(B_val_1)
[[ 6.  7.  8.]]
>>> print(B_val_2)
[[  9.  10.  11.]
 [ 12.  13.  14.]]
```

You can actually feed the output of *any* operations, not just placeholders. In this case TensorFlow does not try to evaluate these operations; it uses the values you feed it.

To implement Mini-batch Gradient Descent, we only need to tweak the existing code slightly. First change the definition of `X` and `y` in the construction phase to make them placeholder nodes:

```
X = tf.placeholder(tf.float32, shape=(None, n + 1), name="X")
y = tf.placeholder(tf.float32, shape=(None, 1), name="y")
```

Then define the batch size and compute the total number of batches:

```
batch_size = 100
n_batches = int(np.ceil(m / batch_size))
```

Finally, in the execution phase, fetch the mini-batches one by one, then provide the value of `X` and `y` via the `feed_dict` parameter when evaluating a node that depends on either of them.

```
def fetch_batch(epoch, batch_index, batch_size):
    [...] # load the data from disk
    return X_batch, y_batch

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

    for epoch in range(n_epochs):
        for batch_index in range(n_batches):
            X_batch, y_batch = fetch_batch(epoch, batch_index, batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
```

```
best_theta = theta.eval()
```

We don't need to pass the value of `X` and `y` when evaluating `theta` since it does not depend on either of them.

## Saving and Restoring Models

Once you have trained your model, you should save its parameters to disk so you can come back to it whenever you want, use it in another program, compare it to other models, and so on. Moreover, you probably want to save checkpoints at regular intervals during training so that if your computer crashes during training you can continue from the last checkpoint rather than start over from scratch.

TensorFlow makes saving and restoring a model very easy. Just create a `Saver` node at the end of the construction phase (after all variable nodes are created); then, in the execution phase, just call its `save()` method whenever you want to save the model, passing it the session and path of the checkpoint file:

```
[...]
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0), name="theta")
[...]
init = tf.global_variables_initializer()
saver = tf.train.Saver()

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

    for epoch in range(n_epochs):
        if epoch % 100 == 0:  # checkpoint every 100 epochs
            save_path = saver.save(sess, "/tmp/my_model.ckpt")

        sess.run(training_op)

    best_theta = theta.eval()
    save_path = saver.save(sess, "/tmp/my_model_final.ckpt")
```

Restoring a model is just as easy: you create a `Saver` at the end of the construction phase just like before, but then at the beginning of the execution phase, instead of initializing the variables using the `init` node, you call the `restore()` method of the `Saver` object:

```
with tf.Session() as sess:
    saver.restore(sess, "/tmp/my_model_final.ckpt")
    [...]
```