Chapter 9. Up and Running with TensorFlow

Basic principle: 1. define a graph of computations to perform; 2. TensorFlow takes that graph and run it efficiently using optimized C++ code.

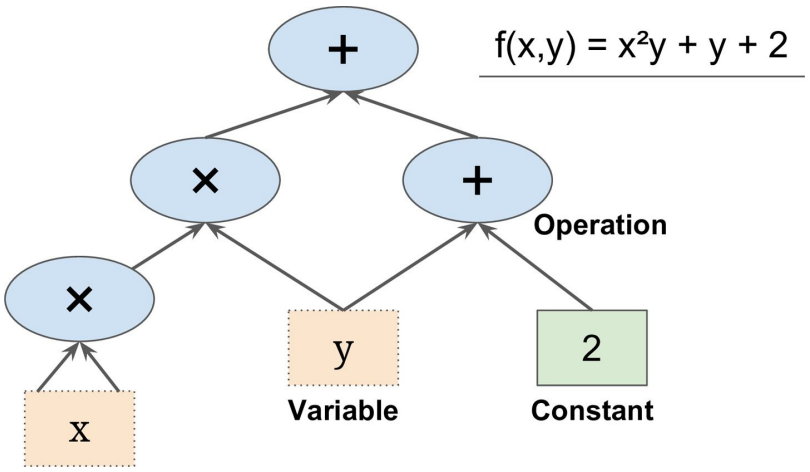

Figure 9-1. A simple computation graph

Importantly, parallel with multiple CPUs or GPUs, distributed computing.

Characteristic: clean design, scalability, flexibility, and great documentation. (flexible, scalable, and production-ready)

Highlights:

- All types of platforms
- API *TF.Learn* (*tensorflow. contrib. learn*)
- *TF-slim* (*tensorflow. contrib. slim*)
- APIs Keras or Pretty Tensor
- Its main Python API offers much more flexibility (at the cost of higher complexity) to create all sorts of computations
- Highly efficient C++ implementations
- Advanced optimization nodes to search for the parameters (gradient) that minimize a cost function (*automatic differentiating, autodiff*)
- Visualization tool *TensorBoard*
- Cloud computing
- Great community: https://github.com/jtoy/awesome-tensorflow (https://github.com/jtoy/awesome-tensorflow)

Installation

Install in a virtual environment

```
$ pip3 install --upgrade tensorflow
```

Check version

```
$ python3 -c 'import tensorflow; print(tensorflow.version)'
```

Creating Your First Graph and Running It in a Session

1. Creates a computation graph

In [95]:
```
import tensorflow as tf
x = tf.Variable(3, name="x")
y = tf.Variable(4, name="y")
f = x*x*y + y + 2
```

The code does not perform any computation, even any variable initialization. To evaluate this graph, open a TensorFlow *session* and use it to initialize the variables and evaluate f. A TensorFlow session takes care of placing the operations onto devices such as CPUs and GPUs and running them, and it holds all the variable values.

In [96]:
```
sess = tf.Session()
sess.run(x.initializer)
sess.run(y.initializer)
result = sess.run(f)
print(result)
sess.close()
```

```
42
```

Repeat $sess.run()$ all the time is a bit cumbersome

In [97]:
```
with tf.Session() as sess:
    x.initializer.run()
    y.initializer.run()
    result = f.eval()
```

Inside the with block, the session is set as the default session. Calling x.initializer.run() is equivalent to calling tf.get_default_session().run(x.initializer), and similarly f.eval() is equivalent to calling tf.get_default_session().run(f). Moreover, the session is automatically closed at the end of the block.

Instead of manually running the initializer for every single variable, you can use the global_variables_initializer() function. Note that it does not actually perform the initialization immediately, but rather creates a node in the graph that will initialize all variables when it is run:

In [98]:
```
init = tf.global_variables_initializer() # prepare an init node

with tf.Session() as sess:
    init.run() # actually initialize all the variables
    result = f.eval()
```

Inside Jupyter or within a Python shell you may prefer to create an *InteractiveSession*. The only difference from a regular *Session* is that when an *InteractiveSession* is created it automatically sets itself as the default session, so you don't need a with block (but you do need to close the session manually when you are done with it):

```
sess = tf.InteractiveSession()}
init.run()
result = f.eval()
print(result)
42
sess.close()
```

A TensorFlow program is typically split into two parts:

1. Construction phase: builds a computation graph representing the ML model and the computations required to train it.
2. Execution phase: runs a loop that evaluates a training step repeatedly, gradually improving the model parameters.

Managing Graphs

Any node you create is automatically added to the default graph:

In [99]:
```
x1 = tf.Variable(1)
x1.graph is tf.get_default_graph()
```

Out[99]:
```
True
```

Managing multiple independent graphs, create a new Graph and temporarily making it the default graph inside a *with* block

In [100]:
```
>>> graph = tf.Graph()
>>> with graph.as_default():
...     x2 = tf.Variable(2)
...
>>> x2.graph is graph
```

Out[100]:
```
True
```

In [101]:
```
>>> x2.graph is tf.get_default_graph()
```

Out[101]:
```
False
```

TIP

> In Jupyter (or in a Python shell), it is common to run the same commands more than once while you are experimenting. As a result, you may end up with a default graph containing many duplicate nodes. One solution is to restart the Jupyter kernel (or the Python shell), but a more convenient solution is to just reset the default graph by running tf.reset_default_graph().

Lifecycle of a Node Value

When evaluating a node, TensorFlow automatically determines the set of nodes that it depends on and it evaluates these nodes first.

In [102]:
```
w = tf.constant(3)
x = w + 2
y = x + 5
z = x * 3

with tf.Session() as sess:
    print(y.eval()) # 10
    print(z.eval()) # 15
```

```
10
15
```

IMPORTANT: it will NOT reuse the result of the previous evaluation of w and x. In short, the preceding code evaluates w and x twice.

All node values are dropped between graph runs, except variable values, which are maintained by the session across graph runs. A variable starts its life when its initializer is run, and it ends when the session is closed.

If you want to evaluate y and z efficiently, without evaluating w and x twice as in the previous code, you must ask TensorFlow to evaluate both y and z in just one graph run

In [103]:
```
with tf.Session() as sess:
    y_val, z_val = sess.run([y, z])
```

WARNING

> In single-process TensorFlow, multiple sessions do not share any state, even if they reuse the same graph (each session would have its own copy of every variable). In distributed TensorFlow, variable state is stored on the servers, not in the sessions, so multiple sessions can share the same variables.

Linear Regression with TensorFlow

TensorFlow operators (*ops*), constant and variables (*source ops*), inputs and outputs are multidimensional arrays (*tensors*).

The following code manipulates 2D arrays to perform Linear Regression on the California housing dataset.

In [104]:
```
import numpy as np
from sklearn.datasets import fetch_california_housing

housing = fetch_california_housing()
m, n = housing.data.shape
housing_data_plus_bias = np.c_[np.ones((m, 1)), housing.data]

X = tf.constant(housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1), dtype=tf.float32, name="y")
XT = tf.transpose(X)
theta = tf.matmul(tf.matmul(tf.matrix_inverse(tf.matmul(XT, X)), XT), y)

with tf.Session() as sess:
    theta_value = theta.eval()
```

The main benefit of this code versus computing the Normal Equation directly using NumPy is that TensorFlow will automatically run this on your GPU card.

Implementing Gradient Descent

WARNING

> When using Gradient Descent, remember that it is important to first normalize the input feature vectors, or else training may be much slower.

Gradient Descent requires scaling the feature vectors first. We could do this using TF, but let's just use Scikit-Learn for now.

In [105]:
```
from  sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaled_housing_data = scaler.fit_transform(housing.data)
scaled_housing_data_plus_bias = np.c_[np.ones((m, 1)), scaled_housing_data]
```

In [106]:
```
print(scaled_housing_data_plus_bias.mean(axis=0))
print(scaled_housing_data_plus_bias.mean(axis=1))
print(scaled_housing_data_plus_bias.mean())
print(scaled_housing_data_plus_bias.shape)
```

```
[ 1.00000000e+00  6.60969987e-17  5.50808322e-18  6.60969987e-17
 -1.06030602e-16 -1.10161664e-17  3.44255201e-18 -1.07958431e-15
 -8.52651283e-15]
[ 0.38915536  0.36424355  0.5116157  ... -0.06612179 -0.06360587
  0.01359031]
0.11111111111111005
(20640, 9)
```

Processing math: 100%

Manually Computing the Gradients

The assign() function creates a node that will assign a new value to a variable.

In [107]:
```
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)

# # TF autodiff
# gradients = tf.gradients(mse, [theta])[0]

# # Gradient Descent optimizer
# optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
# optimizer = tf.train.MomentumOptimizer(learning_rate=learning_rate, momentum=0
# training_op = optimizer.minimize(mse)

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()
```

```
Epoch 0 MSE = 7.0824747
Epoch 100 MSE = 0.7553069
Epoch 200 MSE = 0.6274912
Epoch 300 MSE = 0.59767675
Epoch 400 MSE = 0.5772778
Epoch 500 MSE = 0.5625756
Epoch 600 MSE = 0.5519619
Epoch 700 MSE = 0.5442983
Epoch 800 MSE = 0.53876376
Epoch 900 MSE = 0.53476614
```

Using autodiff

Use *symbolic differentiation* to automatically compute the partial derivatives, but the resulting code would not necessarily be very efficient.

TensorFlow's autodiff feature can automatically and efficiently compute the gradients.

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

The gradients() function takes an op (in this case mse), 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 .

Using an Optimizer

Replace the preceding gradients = ... and training_op = ... lines with the following code

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

Change different type of optimizer

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

Feeding Data to the Training Algorithm

The placeholder nodes are typically used to pass the training data to TensorFlow during training.

NOTE

> 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.

Mini-batch Gradient Descent

In [108]:
```
n_epochs = 10
learning_rate = 0.01

tf.reset_default_graph()

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

theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=42), name="the
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(mse)

init = tf.global_variables_initializer()

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

def fetch_batch(epoch, batch_index, batch_size):
    np.random.seed(epoch * n_batches + batch_index)  # not shown in the book
    indices = np.random.randint(m, size=batch_size)  # not shown
    X_batch = scaled_housing_data_plus_bias[indices] # not shown
    y_batch = housing.target.reshape(-1, 1)[indices] # not shown
    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()

best_theta
```

Out[108]:
```
array([[ 2.070016  ],
       [ 0.82045615],
       [ 0.11731729],
       [-0.22739057],
       [ 0.31134027],
       [ 0.00353193],
       [-0.01126994],
       [-0.91643935],
       [-0.8795008 ]], dtype=float32)
```

Saving and Restoring Models

Save models to use it in another program, compare it to other models. Moreover, 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.

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:

In [109]:
```
tf.reset_default_graph()

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, seed=42), name="the
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(mse)

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:
            print("Epoch", epoch, "MSE =", mse.eval())
            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")
```

```
Epoch 0 MSE = 2.7544262
Epoch 100 MSE = 0.63222194
Epoch 200 MSE = 0.5727803
Epoch 300 MSE = 0.5585008
Epoch 400 MSE = 0.54907
Epoch 500 MSE = 0.54228795
Epoch 600 MSE = 0.5373791
Epoch 700 MSE = 0.53382194
Epoch 800 MSE = 0.5312425
Epoch 900 MSE = 0.5293705
```

In [110]:
```
best_theta
```

Out[110]:
```
array([[ 2.06855226e+00],
       [ 7.74078071e-01],
       [ 1.31192386e-01],
       [-1.17845066e-01],
       [ 1.64778143e-01],
       [ 7.44078017e-04],
       [-3.91945131e-02],
       [-8.61356676e-01],
       [-8.23479772e-01]], dtype=float32)
```

Restoring a model: 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

In [111]:
```
with tf.Session() as sess:
    saver.restore(sess, "/tmp/my_model_final.ckpt")
    best_theta_restored = theta.eval() # not shown in the book
```

```
INFO:tensorflow:Restoring parameters from /tmp/my_model_final.ckpt
```

In [112]:
```
np.allclose(best_theta, best_theta_restored)
```

Out[112]:
```
True
```

If you want to have a saver that loads and restores theta with a different name, such as "weights":

In [113]:
```
saver = tf.train.Saver({"weights": theta})
```

By default the saver also saves the graph structure itself in a second file with the extension `.meta`. You can use the function `tf.train.import_meta_graph()` to restore the graph structure. This function loads the graph into the default graph and returns a `Saver` that can then be used to restore the graph state (i.e., the variable values):

In [114]:
```
tf.reset_default_graph()
# notice that we start with an empty graph.

saver = tf.train.import_meta_graph("/tmp/my_model_final.ckpt.meta")  # this loads
theta = tf.get_default_graph().get_tensor_by_name("theta:0") # not shown in the l

with tf.Session() as sess:
    saver.restore(sess, "/tmp/my_model_final.ckpt")  # this restores the graph's
    best_theta_restored = theta.eval() # not shown in the book
```

```
INFO:tensorflow:Restoring parameters from /tmp/my_model_final.ckpt
```

In [115]:
```
np.allclose(best_theta, best_theta_restored)
```

Out[115]:
```
True
```

This means that you can import a pretrained model without having to have the corresponding Python code to build the graph. This is very handy when you keep tweaking and saving your model: you can load a previously saved model without having to search for the version of the code that built it.

Visualizing the Graph and Training Curves Using TensorBoard

Writes the graph definition and some training stats (MSE) to a log directory that TensorBoard will read from. You need to use a different log directory every time you run your program.

In [116]:
```
tf.reset_default_graph()
```

In [117]:
```
from datetime import datetime
now = datetime.utcnow().strftime("%Y%m%d%H%M%S")
root_logdir = "tf_logs"
logdir = "{}/run-{}/".format(root_logdir, now)
```

In [118]:
```
n_epochs = 1000
learning_rate = 0.01

X = tf.placeholder(tf.float32, shape=(None, n + 1), name="X")
y = tf.placeholder(tf.float32, shape=(None, 1), name="y")
theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0, seed=42), name="the
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(mse)

init = tf.global_variables_initializer()
```

Next, add the following code at the very end of the construction phase:

In [119]:
```
mse_summary = tf.summary.scalar('MSE', mse)
file_writer = tf.summary.FileWriter(logdir, tf.get_default_graph())
```

The first line creates a node in the graph that will evaluate the MSE value and write it to a TensorBoard-compatible binary log string called a *summary*. The second line creates a `FileWriter` that you will use to write summaries to logfiles in the log directory. The first parameter indicates the path of the log directory (relative to the current directory). The second

(optional) parameter is the graph you want to visualize. Upon creation, the FileWriter creates the log directory if it does not already exist (and its parent directories if needed), and writes the graph definition in a binary logfile called an *events* file.

Next you need to update the execution phase to evaluate the `mse_summary` node regularly during training (e.g., every 10 mini-batches). This will output a summary that you can then write to the events file using the `file_writer`.

In [120]:
```
n_epochs = 10
batch_size = 100
n_batches = int(np.ceil(m / batch_size))
```

In [121]:
```
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)
            if batch_index % 10 == 0:
                summary_str = mse_summary.eval(feed_dict={X: X_batch, y: y_batch}
                step = epoch * n_batches + batch_index
                file_writer.add_summary(summary_str, step)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})

    best_theta = theta.eval()
```

In [122]:
```
file_writer.flush()
file_writer.close()
print("Best theta:")
print(best_theta)
```

```
Best theta:
[[ 2.070016  ]
 [ 0.82045615]
 [ 0.11731729]
 [-0.22739057]
 [ 0.31134027]
 [ 0.00353193]
 [-0.01126994]
 [-0.91643935]
 [-0.8795008 ]]
```

WARNING

> Avoid logging training stats at every single training step, as this would significantly slow down training.

Start TensorBoard server

```
$ tensorboard --logdir tf_logs/
Starting TensorBoard on port 6006
(You can navigate to http://0.0.0.0:6006 (http://0.0.0.0:6006) or
http://localhost:6006/ (http://localhost:6006/))
```

Name Scopes

Create *name scopes* to group related nodes.

Modify the previous code to define the error and mse ops within a name scope called "loss":

In [123]:
```
with tf.name_scope("loss") as scope:    # gives the name scopes names by appendi
    error = y_pred - y
    mse = tf.reduce_mean(tf.square(error), name="mse")

print(error.op.name)
print(mse.op.name)
```

```
loss/sub
loss/mse
```

Modularity

Rectified linear unit (ReLU)

$$h_{\mathbf{w},b}(\mathbf{X}) = max(\mathbf{X} \cdot \mathbf{w} + b, 0)$$

In [124]:
```
tf.reset_default_graph()

def relu(X):
    with tf.name_scope("relu"):    # gives the name scopes unique names by appen
        w_shape = (int(X.get_shape()[1]), 1)
        w = tf.Variable(tf.random_normal(w_shape), name="weights")
        b = tf.Variable(0.0, name="bias")
        z = tf.add(tf.matmul(X, w), b, name="z")
        return tf.maximum(z, 0., name="max")

n_features = 3
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
relus = [relu(X) for i in range(5)]
output = tf.add_n(relus, name="output")

file_writer = tf.summary.FileWriter("logs/relu2", tf.get_default_graph())
file_writer.close()
```

Sharing Variables

Option 1. Set the shared variable as an attribute of the `relu()` function upon the first call

In [125]:
```
tf.reset_default_graph()

def relu(X):
    with tf.name_scope("relu"):
        if not hasattr(relu, "threshold"):
            relu.threshold = tf.Variable(0.0, name="threshold")
        w_shape = int(X.get_shape()[1]), 1
        w = tf.Variable(tf.random_normal(w_shape), name="weights")
        b = tf.Variable(0.0, name="bias")
        z = tf.add(tf.matmul(X, w), b, name="z")
        return tf.maximum(z, relu.threshold, name="max")
```

Option 2. Use the `get_variable()` function to create the shared variable if it does not exist yet, or reuse it if it already exists. The desired behavior (creating or reusing) is controlled by an attribute of the current `variable_scope()` .

In [126]:
```
with tf.variable_scope("relu"):
    threshold = tf.get_variable("threshold", shape=(), initializer=tf.constant_i
```

Note that if the variable has already been created by an earlier call to `get_variable()` , this code will raise an exception. This behavior prevents reusing variables by mistake. If you want to reuse a variable, you need to explicitly say so by setting the variable scope's `reuse` attribute to `True`

In [127]:
```
with tf.variable_scope("relu", reuse=True):
    threshold = tf.get_variable("threshold")
```

Alternatively, you can set the `reuse` attribute to `True` inside the block by calling the scope's `reuse_variables()` method:

In [128]:
```
with tf.variable_scope("relu") as scope:
    scope.reuse_variables()
    threshold = tf.get_variable("threshold")
```

WARNING

> Once `reuse` is set to `True` , it cannot be set back to `False` within the block. Moreover, if you define other variable scopes inside this one, they will automatically inherit `reuse=True` . Lastly, only variables created by `get_variable()` can be reused this way.

`relu()` function access the `threshold`

In [129]:
```
tf.reset_default_graph()

def relu(X):
    with tf.variable_scope("relu", reuse=True):
        threshold = tf.get_variable("threshold")  # reuse existing variable
        w_shape = int(X.get_shape()[1]), 1
        w = tf.Variable(tf.random_normal(w_shape), name="weights")
        b = tf.Variable(0.0, name="bias")
        z = tf.add(tf.matmul(X, w), b, name="z")
        return tf.maximum(z, threshold, name="max")

X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
with tf.variable_scope("relu"):  # create the variable
    threshold = tf.get_variable("threshold", shape=(),
                                initializer=tf.constant_initializer(0.0))
relus = [relu(X) for relu_index in range(5)]
output = tf.add_n(relus, name="output")
file_writer = tf.summary.FileWriter("logs/relu6", tf.get_default_graph())
file_writer.close()
```

Create the `threshold` variable within the `relu()` function upon the first call, then reuses it in subsequent calls. Now the `relu()` function does not have to worry about name scopes or variable sharing: it just calls `get_variable()`, which will create or reuse the threshold variable (it does not need to know which is the case).

In [130]:
```
tf.reset_default_graph()

def relu(X):
    threshold = tf.get_variable("threshold", shape=(),
                                initializer=tf.constant_initializer(0.0))
    w_shape = (int(X.get_shape()[1]), 1)                        # not shown in t
    w = tf.Variable(tf.random_normal(w_shape), name="weights")  # not shown
    b = tf.Variable(0.0, name="bias")                           # not shown
    z = tf.add(tf.matmul(X, w), b, name="z")                    # not shown
    return tf.maximum(z, threshold, name="max")

X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
relus = []
for relu_index in range(5):
    with tf.variable_scope("relu", reuse=(relu_index >= 1)) as scope:
        relus.append(relu(X))
output = tf.add_n(relus, name="output")
file_writer = tf.summary.FileWriter("logs/relu9", tf.get_default_graph())
file_writer.close()
```