SAIF

Shanghai Advanced Institute of Finance
上海高级金融学院

量化俱乐部-深度学习-RunningWithTensorFlow

2019-11-03

Menu

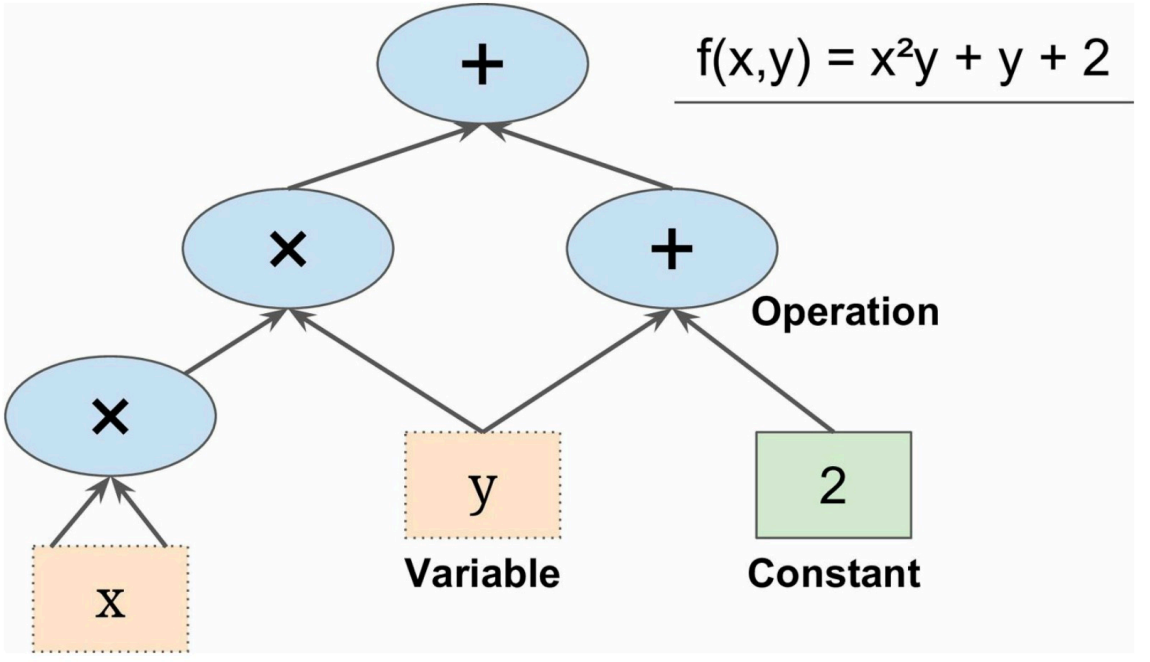
0. Installation
1. Creating first graph and running it in a session
2. Managing Graphs
3. Lifecycle of a Node Value
4. Linear Regression with TensorFlow
5. Implementing Gradient Descent
6. Feeding data to the Training Algorithm
7. Saving and Restoring Models
8. Visualizing with TensorBoard
9. Name Scopes
10. Modularity
11. Sharing Variables

自我介绍

张骁喆，高金FMBA 2017。10年计算机工作经验，熟悉开发，测试，运维，项目管理，产品设计各环节。
曾就职eBay，唯品会，2016加入SAP，现担任SAP Cloud部门开发团队主管。
2017高金开始接触量化投资和python，目前毕业论文研究方向使用机器学习在A股进行量化策略研究。
量化投资策略咨询/机器学习咨询培训/项目管理产品管理。

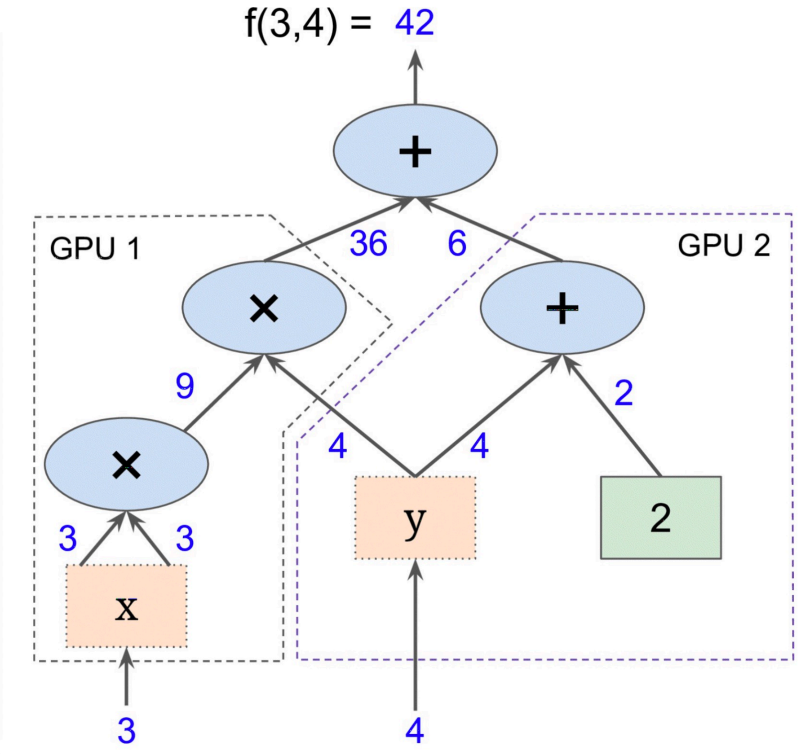
Intro

TensorFlow is a powerful open source software library for numerical computation, particularly well suited and fine-tuned for large-scale Machine Learning. Its basic principle is simple: you first define in Python a graph of computations to perform, and then TensorFlow takes that graph and runs it efficiently using optimized C++ Code.



Intro

Most importantly, it is possible to break up the graph into several chunks and run them in parallel across multiple CPUs or GPUs. TensorFlow also supports distributed computing, so you can train colossal neural networks on humongous training sets in a reasonable amount of time by splitting the computations across hundreds of servers. TensorFlow can train a network with millions of parameters on a training set composed of billions of instances with millions of features each.



Intro

When TensorFlow was open-sourced in November 2015, there were already many popular open source libraries for Deep Learning, and to be fair most of TensorFlow’ s features already existed in one library or another. Nevertheless, TensorFlow’ s clean design, scalability, flexibility, and great documentation (not to mention Google’ s name) quickly boosted it to the top of the list. Here are some of TensorFlow’ s highlights:

1. not only on Windows, Linux, and macOS, but also on mobile devices(iOS and Android
2. It provides a very simple Python API called TF.Learn(tensorflow.contrib.learn), compatible with Scikit-Learn
3. Several other high-level APIs have been built independently on top of TensorFlow, such as Keras or Pretty Tensor
4. It also comes with a great visualization tool called TensorBoard that allows you to browse through the computation graph, view learning curves, and more
5. It provides several advanced optimization nodes to search for the parameters that minimize a cost function. These are very easy to use since TensorFlow automatically takes care of computing the gradients of the functions you define. This is called automatic diferentiating (or autodif).

Intro

In this chapter, we will go through the basics of TensorFlow, from installation to creating, running, saving, and visualizing simple computational graphs. Mastering these basics is important before you build your first neural network (which we will do in the next chapters)

Installations

Let’s get started! Assuming you installed Jupyter and Scikit-Learn by following the installation instructions, you can simply use pip to install TensorFlow :

$ pip3 install --upgrade tensorflow

```
[i328815@MacPro ~]$ pip3 install --upgrade tensorflow
Collecting tensorflow
  Downloading https://files.pythonhosted.org/packages/c8
6m-macosx_10_11_x86_64.whl (102.7MB)
```

Creating First Graph and Running It in a Session

The following code creates the graph represented in slide 4.
That’ s all there is to it! The most important thing to understand is that this code does not actually perform any computation, even though it looks like it does (especially the last line). It just creates a computation graph. In fact, even the variables are not initialized yet. To evaluate this graph, you need to open a TensorFlow session and use it to initialize the variables and evaluate f

In [2]:
```
import tensorflow as tf

reset_graph()

x = tf.Variable(3, name="x")
y = tf.Variable(4, name="y")
f = x*x*y + y + 2

f
```

Out[2]:
```
<tf.Tensor 'add_1:0' shape=() dtype=int32>
```

Creating First Graph and Running It in a Session

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.
The following code creates a session, initializes the variables, and evaluates, and f then closes the session (which frees up resources):

```
In [3]: sess = tf.Session()
        sess.run(x.initializer)
        sess.run(y.initializer)
        result = sess.run(f)

        result
Out[3]: 42

In [4]: sess.close()
```

Creating First Graph and Running It in a Session

Having to repeat sess.run() all the time is a bit cumbersome, but fortunately there is a better way. 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) This makes the code easier to read. Moreover, the session is automatically closed at the end of the block.

```
In [5]: with tf.Session() as sess:
            x.initializer.run()
            y.initializer.run()
            result = f.eval()

In [6]: result
Out[6]: 42
```

Creating First Graph and Running It in a Session

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 [7]: init = tf.global_variables_initializer()

        with tf.Session() as sess:
            init.run()
            result = f.eval()

In [8]: result

Out[8]: 42
```

Creating First Graph and Running It in a Session

A TensorFlow program is typically split into two parts: the first part builds a computation graph (this is called the construction phase), and the second part runs it (this is the execution phase). The construction phase typically builds a computation graph representing the ML model and the computations required to train it. The execution phase generally runs a loop that evaluates a training step repeatedly (for example, one step per mini-batch), gradually improving the model parameters. We will go through an example shortly.

Managing Graphs

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

```
In [12]: reset_graph()

         x1 = tf.Variable(1)
         x1.graph is tf.get_default_graph()
```

```
Out[12]: True
```

In most cases this is fine, but sometimes you may want to manage multiple independent graphs. You can do this by creating a new Graph and temporarily making it the default graph inside a with block.

```
In [13]: graph = tf.Graph()
         with graph.as_default():
             x2 = tf.Variable(2)

         x2.graph is graph
```

```
Out[13]: True
```

```
In [14]: x2.graph is tf.get_default_graph()
```

```
Out[14]: False
```

Lifecycle of a Node Value

When you evaluate a node, TensorFlow automatically determines the set of nodes that it depends on and it evaluates these nodes first. For example, consider the following code:

```
In [15]: 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
```

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.

Linear Regression with TensorFlow

在TensorFlow程序中，所有的数据都通过tensor形式来表示。从功能的角度上看，张量可以简单理解为多维数组。其中零阶张量表示标量(scalar)也就是一个数，第一阶张量为向量(vector),也就是一个一维数组，第n阶张量可以理解为一个n维数组。但张量在TensorFlow中的实现并不是直接采用数组的形式，他只是对运算结果的引用，保存的是如何得到这些数字的计算过程。一个张量主要保存三个属性，name，shape和type.

1. 比如下面代码打出来的“ add ” 就说明了 result 张量是计算节点“add 输出的第0结果
2. 张量的第 2个属性是张量的维度。这个属性描述了 个张量的维度信息。比 如上面样例中 =2 说明了张量 result 维数组 数组的长度为 2。
3. 张量的第3 个属性是类型（ type ），每 个张量会有 个唯 的类型.

```
import tensorflow as tf
# tf.constant 是一个计算，这个计算的结果为一个张量，保存在变量 a 中。
a = tf.constant([1.0, 2.0], name="a")
b = tf.constant([2.0, 3.0], name="b")
result = tf.add(a, b, name="add")
print result
'''
输出:
Tensor("add:0", shape=(2,), dtype=float32)
```

Linear Regression with TensorFlow

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

In [17]:
```
import numpy as np
from sklearn.datasets import fetch_california_housing

reset_graph()

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()
```

In [18]:
```
theta_value
```

Out[18]:
```
array([[-3.68962631e+01],
       [ 4.36777472e-01],
       [ 9.44449380e-03],
       [-1.07348785e-01],
       [ 6.44962370e-01],
       [-3.94082872e-06],
       [-3.78797273e-03],
       [-4.20847952e-01],
       [-4.34020907e-01]], dtype=float32)
```

In [19]:
```
X
```

Out[19]:
```
<tf.Tensor 'X:0' shape=(20640, 9) dtype=float32>
```

Linear Regression with TensorFlow

As usual they do not perform any computations immediately; instead, they create nodes in the graph that will perform them when the graph is run. You may recognize that the definition of theta corresponds to the Normal Equation.
Finally, the code creates a session and uses it to evaluate theta.

$$\hat{\theta} = (\mathbf{X}^T \cdot \mathbf{X})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

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 if you have one.

Implementing Gradient Descent

Let’ s try using Batch Gradient Descent 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.

Implementing Gradient Descent - Manual

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

1. 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 $\theta^{(\text{next step})} = \theta - \eta \nabla_\theta MSE(\theta)$.

2. 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.

In [24]:
```
reset_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="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()
```

```
Epoch 0 MSE = 9.161542
Epoch 100 MSE = 0.71450055
Epoch 200 MSE = 0.56670487
Epoch 300 MSE = 0.55557173
Epoch 400 MSE = 0.5488112
Epoch 500 MSE = 0.5436363
Epoch 600 MSE = 0.53962904
Epoch 700 MSE = 0.5365092
Epoch 800 MSE = 0.53406775
Epoch 900 MSE = 0.5321473
```

Implementing Gradient Descent - 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.

Fortunately, TensorFlow's autodiff feature comes to the rescue: it can automatically and efficiently compute the gradients for you. It computes all the partial derivatives of the outputs with regards to all the inputs in just n outputs + 1 graph traversals.

In [26]:
```
# Same as above except for the gradients = ... line:

reset_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="theta")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
```

In [27]:
```
gradients = tf.gradients(mse, [theta])[0]
```

In [28]:
```
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()

print("Best theta:")
print(best_theta)
```

Implementing Gradient Descent - 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.

In [33]:
```
reset_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="theta")
y_pred = tf.matmul(X, theta, name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
```

In [34]:
```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
training_op = optimizer.minimize(mse)
```

In [35]:
```
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()

print("Best theta:")
print(best_theta)
```

Feeding Data to the Training Algorithm

Placeholder 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.

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

In [40]:
```
reset_graph()

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.]]
```

Feeding Data to the Training Algorithm

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")
```

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()
```

Saving and Restoring Models

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.

In [49]:
```
reset_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="theta")
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")
```

Saving and Restoring Models

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:

In [51]:
```
with tf.Session() as sess:
    saver.restore(sess, "/tmp/my_model_final.ckpt")
    best_theta_restored = theta.eval()
```

```
WARNING:tensorflow:From /Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages/tensorflow/pyth
on/training/saver.py:1276: checkpoint_exists (from tensorflow.python.training.checkpoint_management) is deprecated an
d will be removed in a future version.
Instructions for updating:
Use standard file APIs to check for files with this prefix.
INFO:tensorflow:Restoring parameters from /tmp/my_model_final.ckpt
```

In [54]:
```
# 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):

reset_graph()
# notice that we start with an empty graph.

saver = tf.train.import_meta_graph("/tmp/my_model_final.ckpt.meta")  # this loads the graph structure
theta = tf.get_default_graph().get_tensor_by_name("theta:0")
```

Visualizing the Graph with TensorBoard

So now we have a computation graph that trains a Linear Regression model using Mini-batch Gradient Descent, and we are saving checkpoints at regular intervals. However, we are still relying on the print() function to visualize progress during training. There is a better way: enter TensorBoard. If you feed it some training stats, it will display nice interactive visualizations of these stats in your web browser (e.g., learning curves). You can also provide it the graph's definition and it will give you a great interface to browse through it. This is very useful to identify errors in the graph, to find bottlenecks, and so on.

Visualizing the Graph with TensorBoard

The first step is to tweak your program a bit so it writes the graph definition and some training stats — for example, the training error (MSE) — to a log directory that TensorBoard will read from

Next, add the following code at the very end of the construction phase:
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 log files in the log directory

```
In [59]: mse_summary = tf.summary.scalar('MSE', mse)
         file_writer = tf.summary.FileWriter(logdir, tf.get_default_graph())
```

Visualizing the Graph with TensorBoard

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 [61]:
```
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()
```

Visualizing the Graph with TensorBoard

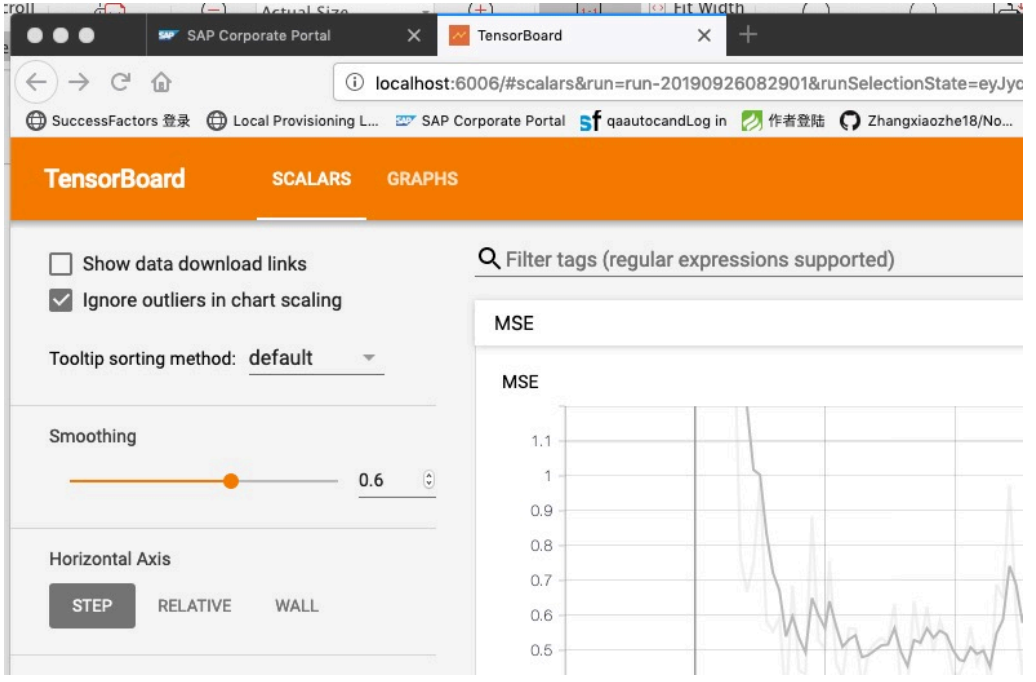
Great! Now it’ s time to fire up the TensorBoard server. You need to start the server by running the tensorboard command, pointing it to the root log directory. This starts the TensorBoard web server, listening on port 6006 (which is “goog” written upside down)

$ tensorboard --logdir tf_logs/

```
[i328815@MacPro ~/Documents/Notes_HandsOn_ML/tf_logs]$ cd ..
[i328815@MacPro ~/Documents/Notes_HandsOn_ML]$ tensorboard --logdir tf_logs/
W1017 15:27:58.765697 123145531944960 plugin_event_accumulator.py:294] Found mor
graph containing a graph_def, as well as one or more graph events.  Overwriting
W1017 15:27:58.765883 123145531944960 plugin_event_accumulator.py:302] Found mor
 metagraph with the newest event.
TensorBoard 1.14.0 at http://MacPro:6006/ (Press CTRL+C to quit)
```

Visualizing the Graph with TensorBoard

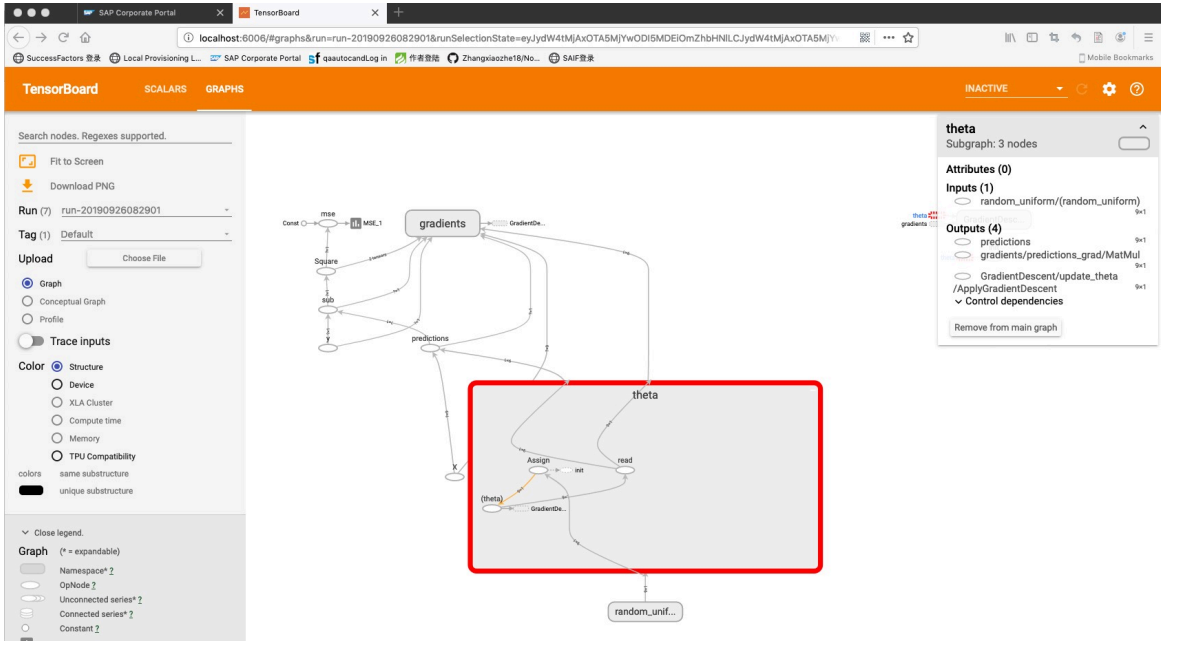
Next open a browser and go to http://0.0.0.0:6006/ (or http://localhost:6006/). Welcome to TensorBoard! In the Events tab you should see MSE on the right. If you click on it, you will see a plot of the MSE during training, for both runs. You can check or uncheck the runs you want to see, zoom in or out, hover over the curve to get details, and so on.



Visualizing the Graph with TensorBoard

Shanghai Advanced Institute of Finance 上海高级金融学院

Now click on the Graphs tab. You should see the graph shown. To reduce clutter, the nodes that have many edges (i.e., connections to other nodes) are separated out to an auxiliary area on the right (you can move a node back and forth between the main graph and the auxiliary area by right-clicking on it). Some parts of the graph are also collapsed by default. For example, try hovering over the gradients node, then click on the ⊕ icon to expand this subgraph. Next, in this subgraph, try expanding the mse_grad subgraph.



Name Scopes

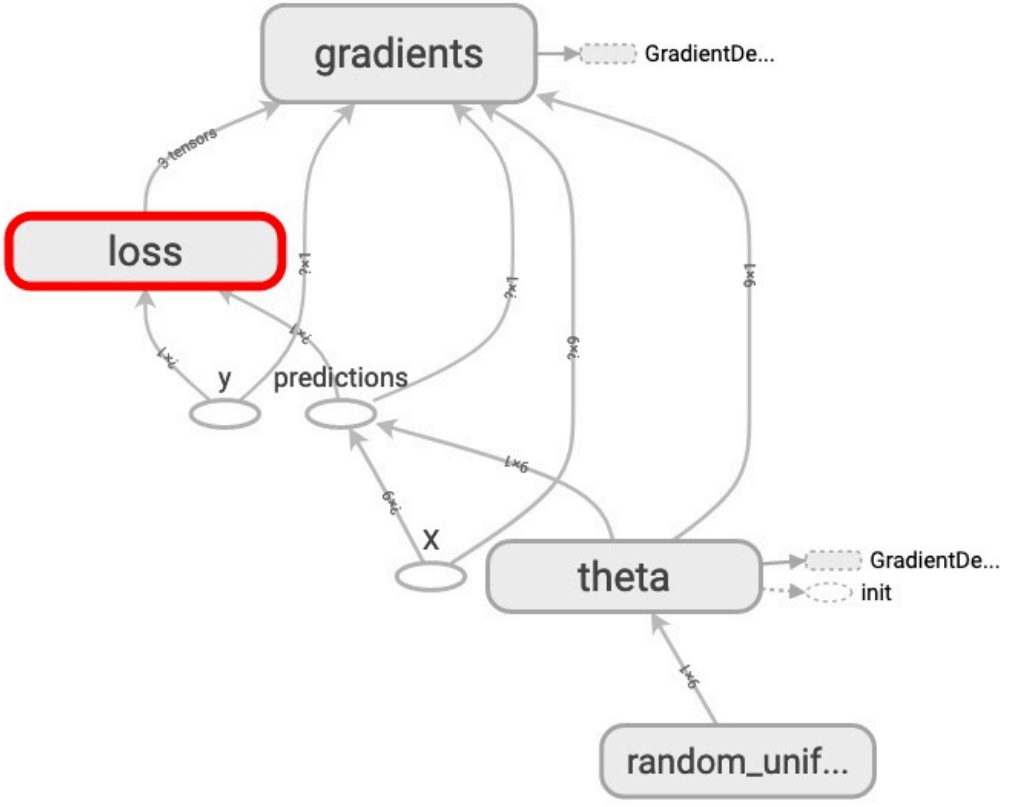
When dealing with more complex models such as neural networks, the graph can easily become cluttered with thousands of nodes. To avoid this, you can create name scopes to group related nodes. For example, let’ s modify the previous code to define the error and mse ops within a name scope called "loss":

In [84]:
```
reset_graph()

now = datetime.utcnow().strftime("%Y%m%d%H%M%S")
root_logdir = "tf_logs"
logdir = "{}/run-{}/".format(root_logdir, now)

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="theta")
y_pred = tf.matmul(X, theta, name="predictions")
```

In [85]:
```
with tf.name_scope("loss") as scope:
    error = y_pred - y
    mse = tf.reduce_mean(tf.square(error), name="mse")
```

Name Scopes

In TensorBoard, the mse and error nodes now appear inside the loss namespace, which appears collapsed by default



Modularity

Suppose you want to create a graph that adds the output of two rectified linear units (ReLU). A ReLU computes a linear function of the inputs, and outputs the result if it is positive, and 0 otherwise

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

The following code does the job, but it's quite repetitive :

```
n_features = 3
X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")

w1 = tf.Variable(tf.random_normal((n_features, 1)), name="weights1")
w2 = tf.Variable(tf.random_normal((n_features, 1)), name="weights2")
b1 = tf.Variable(0.0, name="bias1")
b2 = tf.Variable(0.0, name="bias2")

z1 = tf.add(tf.matmul(X, w1), b1, name="z1")
z2 = tf.add(tf.matmul(X, w2), b2, name="z2")

relu1 = tf.maximum(z1, 0., name="relu1")
relu2 = tf.maximum(z1, 0., name="relu2")

output = tf.add(relu1, relu2, name="output")
```

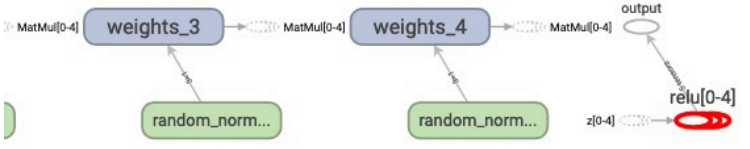
Modularity

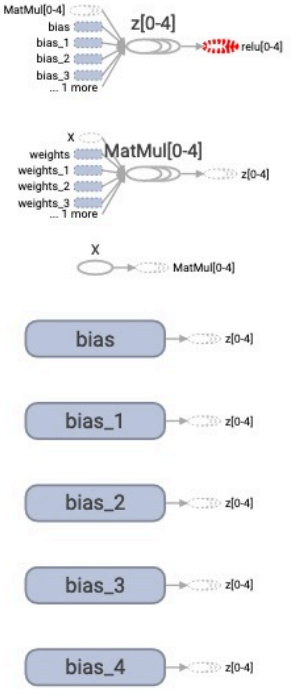
Fortunately, TensorFlow lets you stay DRY (Don’ t Repeat Yourself): simply create a function to build a ReLU

In [71]:
```
# Much better, using a function to build the ReLUs:
reset_graph()

def relu(X):
    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="relu")

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")
```

Modularity

Note that when you create a node, TensorFlow checks whether its name already exists, and if it does it appends an underscore followed by an index to make the name unique.

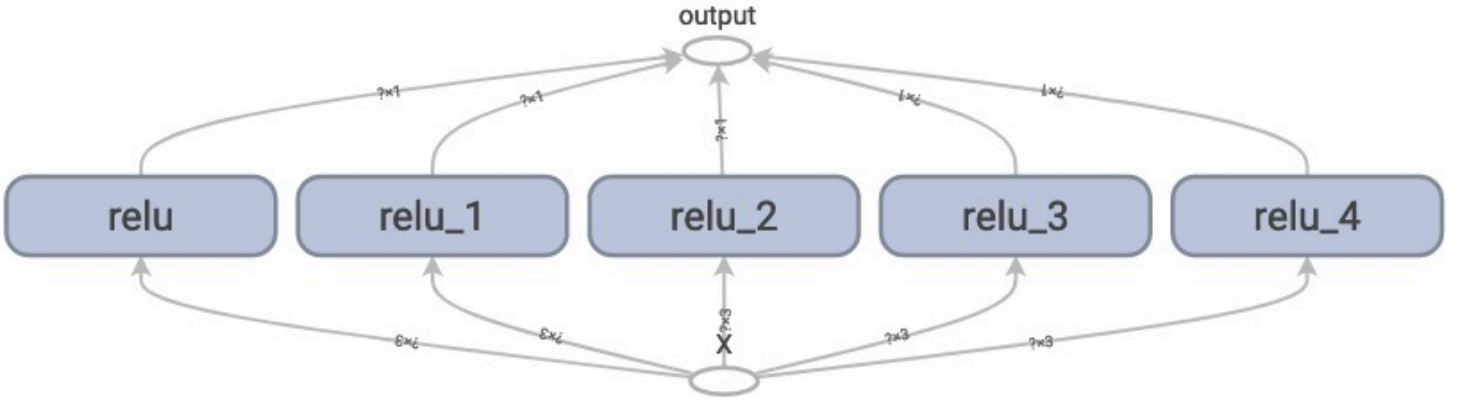




Modularity

Using name scopes, you can make the graph much clearer. Simply move all the content of the relu() function inside a name scope.

```
In [94]: # Even better using name scopes:
         reset_graph()

         def relu(X):
             with tf.name_scope("relu"):
                 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")
```



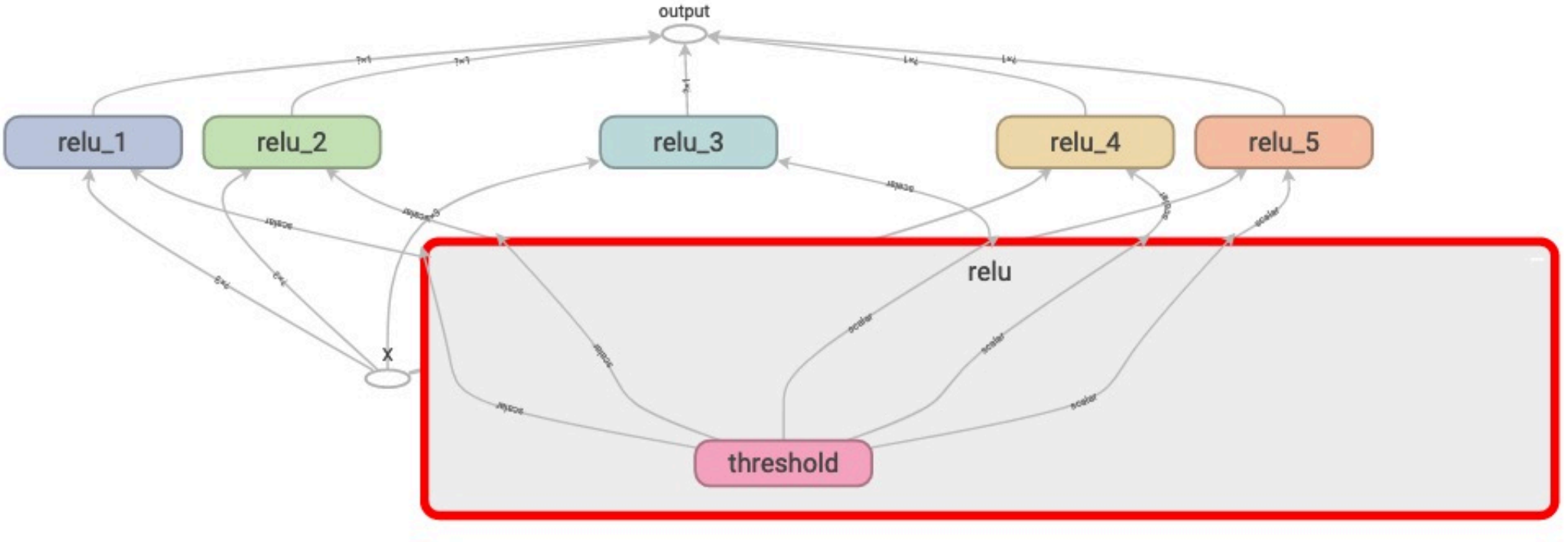
Sharing Variables

If you want to share a variable between various components of your graph, one simple option is to create it first, then pass it as a parameter to the functions that need it. For example, suppose you want to control the ReLU threshold (currently hardcoded to 0) using a shared threshold variable for all ReLUs. You could just create that variable first, and then pass it to the relu() function:

In [81]:
```
reset_graph()

def relu(X):
    with tf.variable_scope("relu", reuse=True):
        threshold = tf.get_variable("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, threshold, name="max")

X = tf.placeholder(tf.float32, shape=(None, n_features), name="X")
with tf.variable_scope("relu"):
    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()
```

Sharing Variables

Above code first defines the relu() function, then creates the relu/threshold variable (as a scalar that will later be initialized to 0.0) and builds five ReLUs by calling the relu() function. The relu() function reuses the relu/threshold variable, and creates the other ReLU nodes. (Notes_HandsOn_ML/logs/relu6)



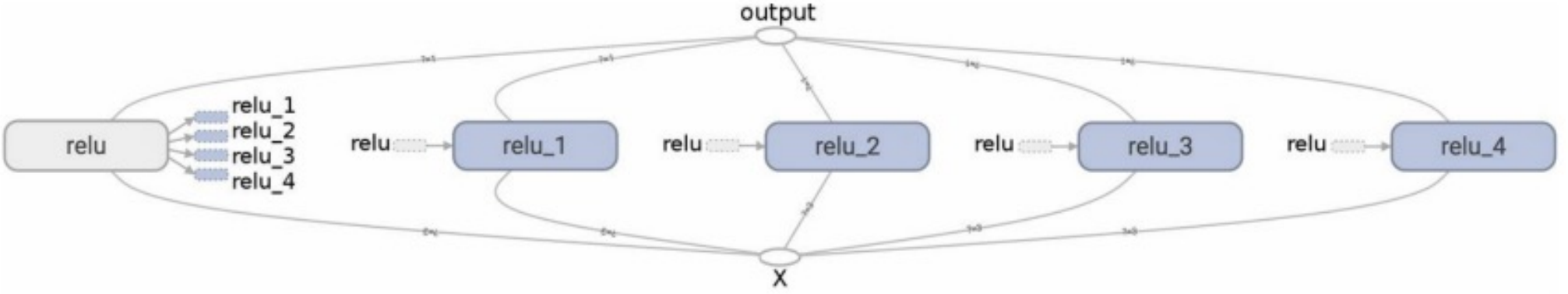
Sharing Variables

It is somewhat unfortunate that the threshold variable must be defined outside the relu() function, where all the rest of the ReLU code resides. To fix this, the following code creates the threshold variable within the relu() function upon the first call, then reuses it in subsequent calls

In [96]:
```
reset_graph()

def relu(X):
    threshold = tf.get_variable("threshold", shape=(),
                                initializer=tf.constant_initializer(0.0))
    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")
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()
```

Sharing Variables

The resulting graph is slightly different than before, since the shared variable lives within the first ReLU.



SAIF

Shanghai Advanced
Institute of Finance
上海高级金融学院