Chapter9 Up and Running with TensorFlow

1. basic principle: 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
2. Most importantly, it is possible to break up the graph into several chunks and run them in parallel across multiple CPUs or GPUs. It also supports distributed computing.
3. A TensorFlow program is typically split into two parts: the first part builds a computation graph, and the second part runs it. Construction phase + 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, gradually improving the model parameters.
4. Reset the default graph:

tf.reset\_default\_graph()

1. Inside the with block, the session is set as the default session.

x.initializer.run() equivalent tf.get\_default\_session().run(x.initializer)

init = tf.global\_variables\_initializer() # prepare an init node

it doesn’t actually perform the initialization immediately, but rather create a node in the graph that will initialize all variables when it is run.

X = tf.Variables(housing.data, dtype = tf.float32, name = ‘X)

A variable starts its life when its initializer is run, and it ends when the session is closed.

X = tf.constant(housing.target, dtype = tf.float32, name = ‘y’) # no initialization

Theta\_val = theta.eval()

1. The inputs and outputs are multidimensional arrays, called tensors.
2. Matrix functions:

tf.transpose(), tf.matmul(), tf.matrix\_inverse()

1. Implementing Gradient Descent

tf.random\_uniform([n, 1], -1, 1)

generate a tensor containing random values, given shape, value range.

Eg: theta = tf.Variable(tf.random\_uniform([n + 1, 1], -1.0, 1.0, seed = 42,

name = 'theta'))

tf.assign()

assign a new value to a variable

eg: training\_op = tf.assign(theta, theta - learning\_rate \* gradients)

tf.reduce\_mean(tf.square(error), name = ‘mse’)

autodiff

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

given ops and variables

optimizer = tf.train.GradientDescentOptimizer( learning\_rate = learning\_rate)

training\_op = optimizer.minimize(mse)

1. Feeding data to the training algorithm

For mini-batch Gradient Descent, you need to replace x and y at every iteration with the next mini-batch.

tf.placeholder()

given outputs tensor’s data type, shape, name

eg: A = tf.placeholder(tf.float32, shape = (None, 1), name = ‘A’)

None means ‘any size’.

Feet data when eval

B\_val = B.eval(feed\_dict = {A:[[1, 2, 3]]})

1. Saving and Restoring models

Save checkpoints as 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.

saver = tf.train.Saver()

at the end of construction phase, after all variable nodes are created.

Int execution phase, just call save() method to save model. Given the session and path of the checkpoint file.

Save\_path = saver.save(sess, ‘/tmp/my\_model\_final.ckpt’)

Restoring a model

Create a Saver at the end of construction phase, and initialize the variables using restore() method at the beginning of execution phase.

saver.restore(sess, ‘/tmp/my\_model\_final.ckpt’)

if you only want to restore some variable.

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

Eg: saver = tf.train.import\_meta\_graph("/tmp/my\_model\_final.ckpt.meta")

1. Name scopes
2. Modularity
3. Sharing Variables

Create it first, then pass it as a parameter to the functions that need it.

Using get\_variable() function