Getting Started With TensorFlow

Ying-Ting Wang

1. **Tensors**

**Tensor** is the central unit of data in TensorFlow. A tensor consist of a set of primitive values, shaped into an array of any number of dimensions. A tensor’s **rank** is its number of dimensions. Here are some examples of tensors:

3 # a rank 0 tensor; this is a scalar with shape [ ]

[1., 2., 3.] # a rank 1 tensor; this is a vector with shape [3]

[[1., 2., 3.], [4., 5., 6.]] # a rank 2 tensor; a matrix with shape [2, 3]

1. **TensorFlow Core tutorial**

Importing TensorFlow

The canonical import statement for TensorFlow is ~~as follows(精簡句子)~~:

>>> import tensorflow as tf

This gives Python access to all of TensorFlow’s classes, methods and symbols. ~~Most of the documentation assumes you have already done this.(這句話不需要在這個文件中提；且如果還有其他系列教學，還是要保留 “import tensorflow as tf” 這句)~~

The Computational Graph

~~You might think of(用肯定的敘述句直接表達意思)~~ TensorFlow Core programs as consisting of two discrete sections:

1. Building the computational graph.
2. Running the computational graph.

A **computational graph** is a series of TensorFlow operations, arranged into a graph of nodes. ~~Let’s build a simple~~ A computational graph, each node takes zero or more tensors as inputs and produces a tensor as an output. =>斷行 (我認為下一段應該要是全新的開始，在講另一件事情了 => constant )

One type of node is a constant. Like all TensorFlow constants, it takes no inputs, and it outputs a value it stores internally. We can create two floating point Tensors *node1* and *node2* as follows:

>>> node1 = tf.constant(3.0, tf.float32)

>>> node2 = tf.constant(4.0) # also tf.float32 implicitly

>>> print(node1, node2)

The print statement produces

Tensor(“Const:0”, shape=( ), dtype=float32)

Tensor(“Const\_1:0”, shape=( ), dtype=float32)

~~Notice that~~ Printing the nodes does not output values 3.0 and 4.0 as ~~you might~~ expect. Instead, when evaluated, the nodes would produce 3.0 and 4.0 respectively. To actually evaluate the nodes, we must run the computational graph within a **session**. A session encapsulates the control and state of the TensorFlow runtime.

The following code creates a *Session* object and then invokes its **run** method to run enough of the computational graph to evaluate *node1* and *node2*. By running the computational graph in a session as follows:

>>> sess = tf.Session( )

>>> print(sess.run([node1, node2]))

We see the expected values of 3.0 and 4.0:

[3.0, 4.0]

We can build more ~~complicated(不必要的形容詞)~~ computations by combining Tensor nodes with opeartions (Operations are also nodes.). For example, we can add ~~our~~ two constant nodes and produce a new graph as follows:

>>> node3 = tf.add(node1, node2)

>>> print(“node3: “, node3)

>>> print(“sess.run(node3): “, sess.run(node3))

The ~~last two(不必要的形容詞)~~ print statements produce

node3: Tensor(“Add\_2:0”, shape=( ), dtype=float32)

sess.run(node3): 7.0

TensorFlow provides a utility called TensorBoard, that can display a picture of the computational graph. Here is a screenshot showing how TensorBoard visualizes the graph:



Fig 1. Two constant nodes combined to a new node.

Fig 1. shows producing a constant result.

A graph can be parameterized to accept external inputs, known as placeholders. A placeholder is a promise to provide a value later.

>>> a = tf.placeholder(tf.float32)

>>> b = tf.placeholder(tf.float32)

>>> adder\_node = a + b # + provides a shortcut for tf.add(a, b)

~~The preceding three lines are a bit like a function or a lambda in which we define two input paramenters (a and b) and then an operation on them.(看不懂他用什麼比喻，會讓人混淆)~~ We can evaluate this graph with multiple inputs by using the feed\_dict parameter to specify Tensors that provide concrete values to these placeholders: (這一段目前看不太懂 : feed\_dict & concrete values)

>>> print(sess.run(adder\_node, {a: 3, b: 4.5}))

>>> print(sess.run(adder\_node, {a: [1,3], b: [2,4]}))

Resulting in the output

7.5

[ 3. 7. ]

In TensorBoard, the graph looks like this:

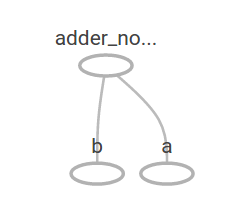


Fig 2. Two nodes are combined to a new node using “adder\_node”

We can make the computational graph more complex by adding another operation. For example,

>>> adder\_node = a + b (這裡官方文件省略沒有寫，我覺得要補上，因為要回頭找，或是以為這裡跟前面沒有關聯)

>>> add\_and\_triple = adder\_node \* 3

>>> print(sess.run(add\_and\_triple, {a: 3, b: 4.5}))

Produces the output

22.5

The preceding computational graph would look as follows in TensorBoard:

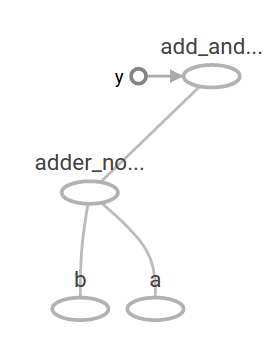


Fig 3. Two nodes are combined to a new node, then opearte to another new node.

哪裡冒出來的 y...?

~~In machine learning we will typically want a model that can take arbitrary inputs(把重點”arbitrary inputs”提前)~~, In machine learning we want to take arbitrary inputs in a model, ~~such as the one above (不懂指得是什麼)~~. To make the model trainable, we need to be able to modify the graph to get new outputs with the same input (我對這裡有疑問，因為這一句來回看了好幾遍，不知是否可以修飾得更好?). Variables allow us to add trainable parameters to a graph. They are constructed with a type and initial value:

>>> W = tf.Variable([.3], tf.float32)

>>> b = tf.Variable([-.3], tf.float32)

>>> x = tf.placeholder(tf.float32)

>>> linear\_model = W \* x + b

Constants are initialized when you call tf.constant, and their value can never change. By contrast, variables are not initialized when you call tf.Variable. To initialize all the variables in a TensorFlow program, you must explicitly call a special operation as follows:

>>> init = tf.global\_variables\_initializer( )

>>> sess.run(init)

It is important to realize init is a handle to the TensorFlow sub-graph that initializes all the gloabl variables. Until we call sess.run, the variables are uninitialized.

Since x is a placeholder, we can evaluate linear\_model for several values of x simultaneously as follows:

>>> print(sess.run(linear\_model, {x: [1, 2, 3, 4]}))

To produce the output

[ 0. 0.30000001 0.60000002 0.90000004 ]

To evaluate the model on training data, a placeholder “d” is needed to provide the desired values. A loss function measures how far apart the current model is from the provided data. A loss model sums the squares of the difference between the current model and the provided data. linear\_model – d creates a vector where each element is the corresponding example’s error delta. We call tf.square to square that error. Then, all the squared errors are summed to create a single scalar that abstracts the error of all examples using tf.reduce\_sum :

>>> d = tf.placeholder(tf.float32)

>>> squared\_deltas = tf.square(linear\_model – d)

>>> loss = tf.reduce\_sum(squared\_deltas)

>>> print(sess.run(loss, {x:[1, 2, 3, 4], d:[0, -1, -2, -3]}))

Producing the loss value:

23.66

A variable is initialized to the value provided to the value provided to tf.Variable can be changed using operation tf.assign. For example, W=-1 and b=1 are the optimal parameters for this model. We can change W and b accordingly:

>>> fixW = tf.assign(W, [-1.])

>>> fixb = tf.assign(b, [1.])

>>> sess.run([fixW, fixb])

>>> print(sess.run(loss, {x:[1, 2, 3, 4], y:[0, -1, -2, -3]}))

The final print shows the loss now is zero.

0.0

1. **tf.train API**