Python TensorFlow Tutorial – Build a Neural Network

April 8, 2017 Andy Deep learning, Neural networks, TensorFlow 19

Google's TensorFlow has been a hot topic in deep learning recently. The open source software, designed to allow efficient computation of data flow graphs, is especially

POPULAR TUTORIALS

Neural Networks Tutorial Pathway to Deep Learnin
Python TensorFlow Tutor – Build a Neural Network
Convolutional Neural Networks Tutorial in TensorFlow
Keras tutorial – build a convolutional neural network in 11 lines
Word2Vec word embeddi tutorial in Python and TensorFlow

CATEGORIES

Amazon AWS
CNTK
Convolutional Neural Networks

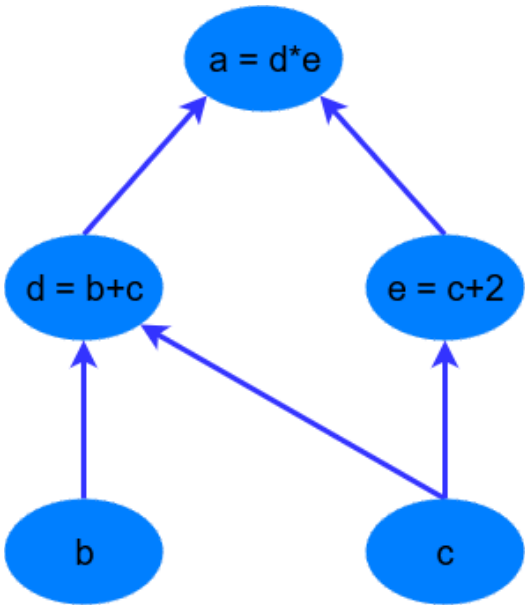
incarnation – version 1.0 – it can even be run on certain mobile operating systems. This introductory tutorial to TensorFlow will give an overview of some of the basic concepts of TensorFlow in Python. These will be a good stepping stone to building more complex deep learning networks, such as Convolution Neural Networks, natural language models and Recurrent Neural Networks in the package. We'll be creating a simple three-layer neural network to classify the MNIST dataset. This tutorial assumes that you are familiar with the basics of neural networks, which you can get up to scratch with in the neural networks tutorial if required. To install TensorFlow, follow the instructions here. The code for this tutorial can be found in this site's GitHub repository. Once you're done, you also might want to check out a higher level deep learning library that sits on top of TensorFlow called Keras – see my Keras tutorial.

First, let's have a look at the main ideas of TensorFlow.

TensorFlow is based on graph based computation – "what on earth is that?", you might say. It's an alternative way of conceptualising mathematical calculations. Consider the following expression $a = (b + c) * (c + 2)$. We can break this function down into the following components:

$$d = b + c$$
$$e = c + 2$$
$$a = d * e$$

Now we can represent these operations graphically as:

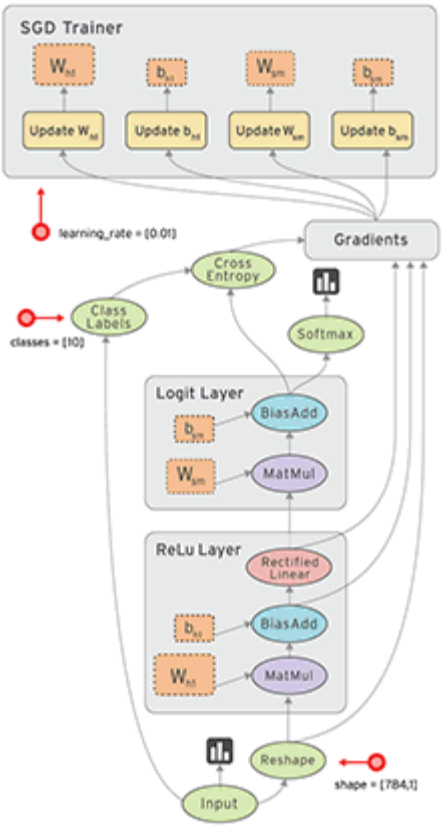


Simple computational graph

This may seem like a silly example – but notice a powerful idea in expressing the equation this way: two of the computations ($d = b + c$ and $e = c + 2$) can be performed in parallel. By splitting up these calculations across CPUs or GPUs, this can give us significant gains in computational times. These gains are a *must* for big data applications and deep learning – especially for complicated neural network architectures such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). The idea

performance improvements via parallel operations and other efficiency gains.

We can look at a similar graph in TensorFlow below, which shows the computational graph of a three-layer neural network.



TensorFlow data flow graph

The animated data flows between different nodes in the graph are *tensors* which are multi-dimensional data arrays. For instance, the input data tensor may be 5000 x 64 x 1, which represents a 64 node input layer with 5000 training samples. After the input layer there is a hidden layer with rectified linear units as the activation function. There is a final output layer (called a "logit layer" in the above graph)

block which finally flow to the Stochastic Gradient Descent optimiser which performs the back-propagation and gradient descent.

Here we can see how computational graphs can be used to represent the calculations in neural networks, and this, of course, is what TensorFlow excels at. Let's see how to perform some basic mathematical operations in TensorFlow to get a feel for how it all works.

2.0 A Simple TensorFlow example

Let's first make TensorFlow perform our little example calculation above – $a = (b + c) * (c + 2)$. First we need to introduce ourselves to TensorFlow *variables* and *constants*. Let's declare some then I'll explain the syntax:

```
1 import tensorflow as tf
2
3 # first, create a TensorFlow constant
4 const = tf.constant(2.0, name="const")
5
6 # create TensorFlow variables
7 b = tf.Variable(2.0, name='b')
8 c = tf.Variable(1.0, name='c')
```

As can be observed above, TensorFlow constants can be declared using the *tf.constant* function, and variables with the *tf.Variable* function. The first element in both is the value to be assigned the constant / variable when it is initialised. The second is an optional name string which can

/ variable from the initialised value, but it can also be set explicitly using the optional *dtype* argument. TensorFlow has many of its own types like *tf.float32*, *tf.int32* etc. – see them all here.

It's important to note that, as the Python code runs through these commands, the variables haven't actually been declared as they would have been if you just had a standard Python declaration (i.e. b = 2.0). Instead, all the constants, variables, operations and the computational graph are only created when the initialisation commands are run.

Next, we create the TensorFlow *operations*:

```
# now create some operations
d = tf.add(b, c, name='d')
e = tf.add(c, const, name='e')
a = tf.multiply(d, e, name='a')
```

TensorFlow has a wealth of operations available to perform all sorts of interactions between variables, some of which we'll get to later in the tutorial. The operations above are pretty obvious, and they instantiate the operations $b + c$, $c + 2.0$ and $d * e$.

The next step is to setup an object to initialise the variables and the graph structure:

```
# setup the variable initialisation
init_op = tf.global_variables_initialize
```

session – *tf.Session.* The TensorFlow session is an object where all operations are run. TensorFlow was initially created in a *static* graph paradigm – in other words, first all the operations and variables are defined (the graph structure) and then these are compiled within the *tf.Session* object. There is now the option to build graphs on the fly using the TensorFlow Eager framework, to check this out see my **TensorFlow Eager tutorial**.

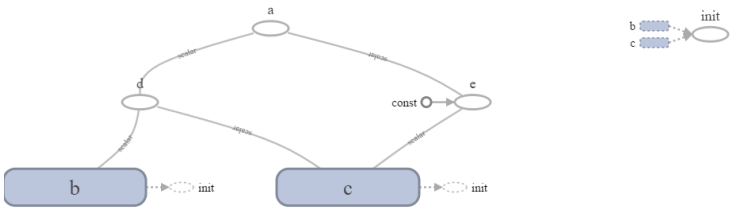
However, there are still advantages in building static graphs using the *tf.Session* object. You can do this by using the *with* Python syntax, to run the graph like so:

```
# start the session
with tf.Session() as sess:
    # initialise the variables
    sess.run(init_op)
    # compute the output of the graph
    a_out = sess.run(a)
    print("Variable a is {}".format(a_ou
```

The first command within the *with* block is the initialisation, which is run with the, well, *run* command. Next we want to figure out what the variable *a* should be. All we have to do is run the operation which calculates *a* i.e. *a = tf.multiply(d, e, name='a').* Note that *a* is an *operation*, not a variable and therefore it can be *run*. We do just that with the *sess.run(a)* command and assign the output to *a_out,* the value of which we then print out.

Note something cool – we defined operations *d* and *e* which need to be calculated before we can figure out what *a* is. However, we don't have to explicitly run *those* operations, as TensorFlow knows what other operations

through its data flow graph which shows it all the required dependencies. Using the TensorBoard functionality, we can see the graph that TensorFlow created in this little program:



Simple TensorFlow graph

Now that's obviously a trivial example – what if we had an array of *b* values that we wanted to calculate the value of *a* over?

2.1 The TensorFlow placeholder

Let's also say that we didn't know what the value of the array *b* would be during the declaration phase of the TensorFlow problem (i.e. before the *with tf.Session() as sess*) stage. In this case, TensorFlow requires us to declare the basic structure of the data by using the *tf.placeholder* variable declaration. Let's use it for *b*:

```
1 # create TensorFlow variables
2 b = tf.placeholder(tf.float32, [None, 1]
```

Because we aren't providing an initialisation in this declaration, we need to tell TensorFlow what data type each element within the *tensor* is going to be. In this case, we want to use *tf.float32*. The second argument is the shape of the data that will be "injected" into this variable. In this

variable (hence the "?"), the placeholder is willing to accept a *None* argument in the size declaration. Now we can inject as much 1-dimensional data that we want into the *b* variable.

The only other change we need to make to our program is in the *sess.run(a,...)* command:

```
a_out = sess.run(a, feed_dict={b: np.ara
```


Note that we have added the *feed_dict* argument to the *sess.run(a,...)* command. Here we remove the mystery and specify exactly what the variable *b* is to be – a one-dimensional range from 0 to 10. As suggested by the argument name, *feed_dict,* the input to be supplied is a Python dictionary, with each key being the name of the *placeholder* that we are filling.

When we run the program again this time we get:

```
Variable a is [[  3.]
 [  6.]
 [  9.]
 [ 12.]
 [ 15.]
 [ 18.]
 [ 21.]
 [ 24.]
 [ 27.]
 [ 30.]]
```

Notice how TensorFlow adapts naturally from a scalar

the data will flow through the graph.

Now we are ready to build a basic MNIST predicting neural network.

3.0 A Neural Network Example

Now we'll go through an example in TensorFlow of creating a simple three layer neural network. In future articles, we'll show how to build more complicated neural network structures such as convolution neural networks and recurrent neural networks. For this example though, we'll keep it simple. If you need to scrub up on your neural network basics, check out my **popular tutorial on the subject**. In this example, we'll be using the MNIST dataset (and its associated loader) that the TensorFlow package provides. This MNIST dataset is a set of 28×28 pixel grayscale images which represent hand-written digits. It has 55,000 training rows, 10,000 testing rows and 5,000 validation rows.

We can load the data by running:

```
1 from tensorflow.examples.tutorials.mnist
2 mnist = input_data.read_data_sets("MNIST_
```

The *one_hot=True* argument specifies that instead of the labels associated with each image being the digit itself i.e. "4", it is a vector with "one hot" node and all the other nodes being zero i.e. [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]. This lets us easily feed it into the output layer of our neural network.

Next, we can set-up the placeholder variables for the training data (and some training parameters):

```
# Python optimisation variables
learning_rate = 0.5
epochs = 10
batch_size = 100

# declare the training data placeholders
# input x - for 28 x 28 pixels = 784
x = tf.placeholder(tf.float32, [None, 78
# now declare the output data placeholde
y = tf.placeholder(tf.float32, [None, 10
```


Notice the *x* input layer is 784 nodes corresponding to the 28 x 28 (=784) pixels, and the *y* output layer is 10 nodes corresponding to the 10 possible digits. Again, the size of *x* is (? x 784), where the ? stands for an as yet unspecified number of samples to be input – this is the function of the *placeholder* variable.

Now we need to setup the weight and bias variables for the three layer neural network. There are always *L-1* number of weights/bias tensors, where *L* is the number of layers. So in this case, we need to setup two tensors for each:

```
# now declare the weights connecting the
W1 = tf.Variable(tf.random_normal([784,
b1 = tf.Variable(tf.random_normal([300])
# and the weights connecting the hidden
W2 = tf.Variable(tf.random_normal([300,
b2 = tf.Variable(tf.random_normal([10]),
```

connections between the input and hidden layer. This neural network will have 300 nodes in the hidden layer, so the size of the weight tensor W1 is [784, 300]. We initialise the values of the weights using a random normal distribution with a mean of zero and a standard deviation of 0.03. TensorFlow has a replicated version of the numpy random normal function, which allows you to create a matrix of a given size populated with random samples drawn from a given distribution. Likewise, we create W2 and b2 variables to connect the hidden layer to the output layer of the neural network.

Next, we have to setup node inputs and activation functions of the hidden layer nodes:

```
# calculate the output of the hidden lay
hidden_out = tf.add(tf.matmul(x, W1), b1
hidden_out = tf.nn.relu(hidden_out)
```

In the first line, we execute the standard matrix multiplication of the weights (*W1*) by the input vector *x* and we add the bias *b1*. The matrix multiplication is executed using the *tf.matmul* operation. Next, we finalise the *hidden_out* operation by applying a rectified linear unit activation function to the matrix multiplication plus bias. Note that TensorFlow has a rectified linear unit activation already setup for us, *tf.nn.relu*.

This is to execute the following equations, as detailed in the neural networks tutorial:

$$z^{(l+1)} = W^{(l)}x + b^{(l)}$$
$$h^{(l+1)} = f(z^{(l+1)})$$

```
# now calculate the hidden layer output
# output layer
y_ = tf.nn.softmax(tf.add(tf.matmul(hidd
```

Again we perform the weight multiplication with the output from the hidden layer (*hidden_out*) and add the bias, *b2*. In this case, we are going to use a **softmax activation** for the output layer – we can use the included TensorFlow softmax function *tf.nn.softmax*.

We also have to include a cost or loss function for the optimisation / backpropagation to work on. Here we'll use the cross entropy cost function, represented by:

$$J = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{n} y_j^{(i)} log(y_{j\_}^{(i)}) + (1-y_j^{(i)}) log(1-y_{j\_}^{(i)})$$

Where $y_j^{(i)}$ is the ith training label for output node *j*, $y_{j\_}^{(i)}$ is the ith predicted label for output node *j*, *m* is the number of training / batch samples and *n* is the number . There are two operations occurring in the above equation. The first is the summation of the logarithmic products and additions *across all the output nodes*. The second is taking a mean of this summation *across all the training samples*. We can implement this cross entropy cost function in TensorFlow with the following code:

```
y_clipped = tf.clip_by_value(y_, 1e-10,
cross_entropy = -tf.reduce_mean(tf.reduc
                        + (1 - y) * tf.
```

between 1e-10 to 0.999999. This is to make sure that we never get a case were we have a *log(0)* operation occurring during training – this would return NaN and break the training process. The second line is the cross entropy calculation.

To perform this calculation, first we use TensorFlow's *tf.reduce_sum* function – this function basically takes the sum of a given axis of the tensor you supply. In this case, the tensor that is supplied is the element-wise cross-entropy calculation for a single node and training sample i.e.: $y_j^{(i)} log(y_{j\_}^{(i)}) + (1 - y_j^{(i)}) log(1 - y_{j\_}^{(i)})$. Remember that *y* and *y_clipped* in the above calculation are (*m* x *10*) tensors – therefore we need to perform the first sum over the second axis. This is specified using the axis=1 argument, where "1" actually refers to the second axis when we have a zero-based indices system like Python.

After this operation, we have an (*m* x *1*) tensor. To take the mean of this tensor and complete our cross entropy cost calculation (i.e. execute this part $\frac{1}{m} \sum_{i=1}^{m}$), we use TensorFlow's *tf.reduce_mean* function. This function simply takes the mean of whatever tensor you provide it. So now we have a cost function that we can use in the training process.

Let's setup the optimiser in TensorFlow:

```
1 # add an optimiser
2 optimiser = tf.train.GradientDescentOpti
```

Here we are just using the gradient descent optimiser provided by TensorFlow. We initialize it with a learning rate,

descent see here and here) and the backpropagation for you. How easy is that? TensorFlow has a library of popular neural network training optimisers, see here.

Finally, before we move on to the main show, were we actually run the operations, let's setup the variable initialisation operation and an operation to measure the accuracy of our predictions:


```
# finally setup the initialisation opera
init_op = tf.global_variables_initialize

# define an accuracy assessment operatio
correct_prediction = tf.equal(tf.argmax(
accuracy = tf.reduce_mean(tf.cast(correc
```

The correct prediction operation *correct_prediction* makes use of the TensorFlow *tf.equal* function which returns *True* or *False* depending on whether to arguments supplied to it are equal. The *tf.argmax* function is the same as the numpy argmax function, which returns the index of the maximum value in a vector / tensor. Therefore, the *correct_prediction* operation returns a tensor of size (*m* x *1*) of *True* and *False* values designating whether the neural network has correctly predicted the digit. We then want to calculate the mean accuracy from this tensor – first we have to cast the type of the *correct_prediction* operation from a Boolean to a TensorFlow float in order to perform the *reduce_mean* operation. Once we've done that, we now have an *accuracy* operation ready to assess the performance of our neural network.

3.2 Setting up the training

the full code below, then talk through it:

```
# start the session
with tf.Session() as sess:
   # initialise the variables
   sess.run(init_op)
   total_batch = int(len(mnist.train.lab
   for epoch in range(epochs):
        avg_cost = 0
        for i in range(total_batch):
            batch_x, batch_y = mnist.tra
             _, c = sess.run([optimiser,
                          feed_dict={x: b
            avg_cost += c / total_batch
        print("Epoch:", (epoch + 1), "co
   print(sess.run(accuracy, feed_dict={x
```


Stepping through the lines above, the first couple relate to setting up the *with* statement and running the initialisation operation. The third line relates to our mini-batch training scheme that we are going to run for this neural network. If you want to know about mini-batch gradient descent, check out this post. In the third line, we are calculating the number of batches to run through in each training epoch. After that, we loop through each training epoch and initialise an *avg_cost* variable to keep track of the average cross entropy cost for each epoch. The next line is where we extract a randomised batch of samples, *batch_x* and *batch_y*, from the MNIST training dataset. The TensorFlow provided MNIST dataset has a handy utility function, *next_batch,* that makes it easy to extract batches of data for training.

The following line is where we run two operations. Notice

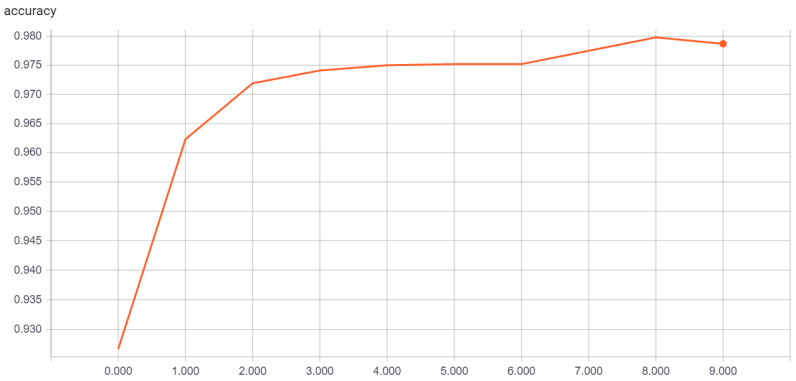
will be performed. As such, we get two outputs, which we have assigned to the variables _ and *c*. We don't really care too much about the output from the *optimiser* operation but we want to know the output from the *cross_entropy* operation – which we have assigned to the variable *c*. Note, we run the *optimiser* (and *cross_entropy*) operation on the batch samples. In the following line, we use *c* to calculate the average cost for the epoch.

Finally, we print out our progress in the average cost, and after the training is complete, we run the *accuracy* operation to print out the accuracy of our trained network on the test set. Running this program produces the following output:


```
Epoch: 1 cost = 0.586
Epoch: 2 cost = 0.213
Epoch: 3 cost = 0.150
Epoch: 4 cost = 0.113
Epoch: 5 cost = 0.094
Epoch: 6 cost = 0.073
Epoch: 7 cost = 0.058
Epoch: 8 cost = 0.045
Epoch: 9 cost = 0.036
Epoch: 10 cost = 0.027

Training complete!
0.9787
```

There we go – approximately 98% accuracy on the test set, not bad. We could do a number of things to improve the model, such as regularisation (see this tips and tricks post), but here we are just interested in exploring TensorFlow. You can also use TensorBoard visualisation to look at things



TensorBoard plot of the increase in accuracy over 10 epochs

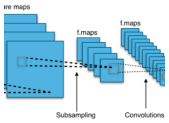
In a future article, I'll introduce you to TensorBoard visualisation, which is a really nice feature of TensorFlow. For now, I hope this tutorial was instructive and helps get you going on the TensorFlow journey. Just a reminder, you can check out the code for this post here. I've also written an article that shows you how to build more complex neural networks such as convolution neural networks, recurrent neural networks and Word2Vec natural language models in TensorFlow. You also might want to check out a higher level deep learning library that sits on top of TensorFlow called Keras – see my Keras tutorial.

Have fun!

[ebook_store ebook_id="653"]

« **PREVIOUS**
Stochastic Gradient Descent – Mini-batch and more

NEXT »
Convolutional Neural Networks Tutorial in TensorFlow



Note: some posts contain Udemy affiliate links