Laboratory #1 Tensorflow

Table of Contents

Based on the definition that tensorflow website has provided: "TensorFlow is an open source software library for high performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. Originally developed by researchers and engineers from the Google Brain team within Google's AI organization, it comes with strong support for machine learning and deep learning and the flexible numerical computation core is used across many other scientific domains." Its first version was released in February 2017. One of the interesting properties of this application is that it is an interdeciplinary software whereas it uses several kernels, softwares, GPU,…. Figure 1 is from Google and is a good overview of its structure.

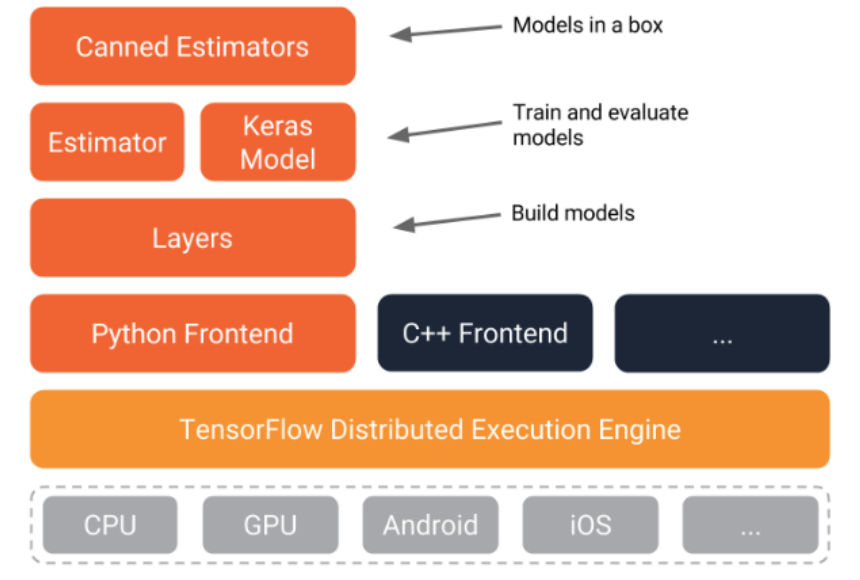


Figure 1- Tensorflow flowchart (from Google)

In this lab, we are going to see some examples on Tensorflow and its applications.

What is a Tensors?

A tensor is a multidimensional array. You can see some figures of tensors in figure 2.

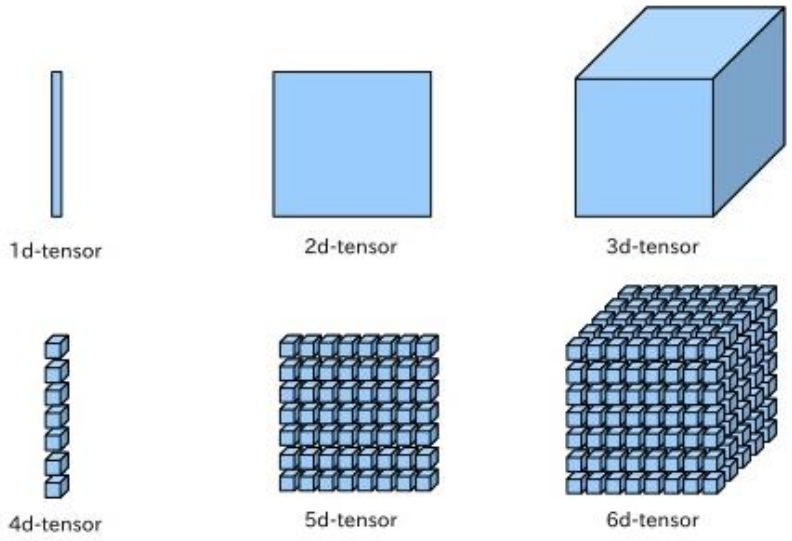


Figure 2- Different tensors

Please run through all the steps of the instruction and answer to the questions carefully. Finally write a report that contains your analysis plus the answer to questions that are indicated by Q.

Step1. Warm-up

Let us start with basic commands in Tensorflow. Run following code in Python and analyze the output:

```
import tensorflow as tf
hello = tf.constant('Hello, TensorFlow!')
sess = tf.Session()
print(sess.run(hello))
```

Analyze the response and explain what *tf.Session()* and *tf.constant()* commands do.

Step2. Implement OCR code in Tensorflow

We have already seen how to implement OCR (Optical Character Recognition) in python and Keras. The goal of this section is to implement same code in Python. We start with loading the data. We can use following code to load MNIST dataset in Python:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

Spend some time to understand this portion of the code.

Q1- Explain this portion of the code.

Q2- what is the usage of *one_hot* in *read_data_sets()* command.

We can write a function to split data into train and test. This function can help us to define the number of training and testing sets that are required.

```
# Functions that can define the size of train and test sets

def TRAIN_SIZE(num):
    print ('Total Training Images in Dataset = ' +
str(mnist.train.images.shape))
    print ('-----------------------------------------------------')
    x_train = mnist.train.images[:num,:]
    print ('x_train Examples Loaded = ' + str(x_train.shape))
    y_train = mnist.train.labels[:num,:]
    print ('y_train Examples Loaded = ' + str(y_train.shape))
    print('')
    return x_train, y_train

def TEST_SIZE(num):
    print ('Total Test Examples in Dataset = ' +
str(mnist.test.images.shape))
    print ('-----------------------------------------------------')
    x_test = mnist.test.images[:num,:]
    print ('x_test Examples Loaded = ' + str(x_test.shape))
    y_test = mnist.test.labels[:num,:]
    print ('y_test Examples Loaded = ' + str(y_test.shape))
    return x_test, y_test
```

We also define some functions that will help us in future for plotting some graphs.

```
import matplotlib.pyplot as plt
import random as ran
import numpy as np

def display_digit(num):
    print(y_train[num])
```

```
    label = y_train[num].argmax(axis=0)
    image = x_train[num].reshape([28,28])
    plt.title('Example: %d  Label: %d' % (num, label))
    plt.imshow(image, cmap=plt.get_cmap('gray_r'))
    plt.show()

def display_mult_flat(start, stop):
    images = x_train[start].reshape([1,784])
    for i in range(start+1,stop):
        images = np.concatenate((images, x_train[i].reshape([1,784])))
    plt.imshow(images, cmap=plt.get_cmap('gray_r'))
    plt.show()
```

We can now start our actual coding. We can use 55000 of data for training. We can easily change this number to see how result is changing.

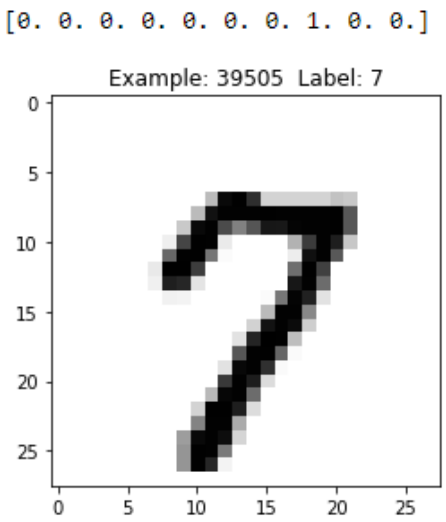
```
x_train, y_train = TRAIN_SIZE(55000)
```

Similarly:

```
x_test, y_test = TEST_SIZE(10000)
```

If you want to see the input values , following code can randomly show you one of the input values.

```
display_digit(ran.randint(0, x_train.shape[0]))
```



We need to define variables that we need throughout the code. We define these variables using *tf.placeholder()* and *tf.Variable()* commands but before doing so, we need to open a session in tensorflow.

```
sess = tf.Session()

# Input and output
x = tf.placeholder(tf.float32, shape=[None, 784]) # input image 28*28 = 784
y_ = tf.placeholder(tf.float32, shape=[None, 10])  #0-9 digits (10 classes)


# Model parameters
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
```

As we discussed earlier, the best output activation node for multiclass will be softmax which can be implemented as:

```
y = tf.nn.softmax(tf.matmul(x,W) + b)
```

Q3- It appears that *tf.placeholder()* , *tf.Variable()* and *tf.constant()* all are for assigning variables so what is the difference between them?

We can initialize variables using following code:

```
init = tf.global_variables_initializer()
sess.run(init)
```

We are now ready to train the model. Follow following code to start training the model.

```
LEARNING_RATE = 0.1
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y),
reduction_indices=[1]))
training =
tf.train.GradientDescentOptimizer(LEARNING_RATE).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

Q4- Write a few sentences about `tf.train.GradientDescentOptimizer` and `tf.equal` commands. What is the effect of `tf.argmax()`.

We need to repeat the training steps based on the number of epochs that are defined.

```
TRAIN_STEPS = 250
for i in range(TRAIN_STEPS+1):
    sess.run(training, feed_dict={x: x_train, y_: y_train})
    if i%100 == 0:
```

```
        print('Training Step:' + str(i) + '  Accuracy =  ' +
str(sess.run(accuracy, feed_dict={x: x_test, y_: y_test})) + '  Loss = ' +
str(sess.run(cross_entropy, {x: x_train, y_: y_train})))
```

This code will show the accuracy every 100 steps.

```
Training Step:0  Accuracy =  0.8916  Loss = 0.45068264
Training Step:100  Accuracy =  0.8982  Loss = 0.41931435
Training Step:200  Accuracy =  0.8999  Loss = 0.398809
Training Step:300  Accuracy =  0.9037  Loss = 0.38404155
Training Step:400  Accuracy =  0.9063  Loss = 0.37273577
Training Step:500  Accuracy =  0.9075  Loss = 0.36370683
Training Step:600  Accuracy =  0.9089  Loss = 0.35627177
Training Step:700  Accuracy =  0.9099  Loss = 0.3500044
Training Step:800  Accuracy =  0.9115  Loss = 0.3446235
Training Step:900  Accuracy =  0.9126  Loss = 0.33993515
Training Step:1000  Accuracy =  0.9129  Loss = 0.33579978
```

Q5- Explain this part of the code.

Add new layer

Now we are good to add a few hidden layers and see the result. We are going to add two hidden layers. Input layer has 784 nodes. First hidden layer has 100 nodes. Second hidden layer has 30 nodes. The output layer has 10 nodes.

```
# Input and output
X = tf.placeholder(tf.float32, shape=[None, 784]) # input image 28*28 = 784
Y_ = tf.placeholder(tf.float32, shape=[None, 10])  #0-9 digits (10 classes)

# Two hidden layers, output layer and their number of neurons (the last layer
has 10 softmax neurons)
L = 100
M = 30

# tf.truncated_normal is a TensorFlow function that produces random values
following the normal (Gaussian) distribution between -2*stddev and +2*stddev
W1 = tf.Variable(tf.truncated_normal([784, L], stddev=0.1))  # 784 = 28 * 28
B1 = tf.Variable(tf.zeros([L]))
W2 = tf.Variable(tf.truncated_normal([L, M], stddev=0.1))
B2 = tf.Variable(tf.zeros([M]))
W3 = tf.Variable(tf.truncated_normal([M, 10], stddev=0.1))
B3 = tf.Variable(tf.zeros([10]))

# The model
XX = tf.reshape(X, [-1, 784])
Y1 = tf.nn.sigmoid(tf.matmul(XX, W1) + B1)
Y2 = tf.nn.sigmoid(tf.matmul(Y1, W2) + B2)
Ylogits = tf.matmul(Y2, W3) + B3
Y = tf.nn.softmax(Ylogits)
```

Similar to previous step, we can define a cross entropy algorithm

```
# cross-entropy loss function (= -sum(Y_i * log(Yi)) ), normalised for
batches of 100  images
# TensorFlow provides the softmax_cross_entropy_with_logits function to avoid
numerical stability
# problems with log(0) which is NaN
cross_entropy = tf.nn.softmax_cross_entropy_with_logits(logits=Ylogits,
labels=Y_)
cross_entropy = tf.reduce_mean(cross_entropy)*100

# accuracy of the trained model, between 0 (worst) and 1 (best)
correct_prediction = tf.equal(tf.argmax(Y, 1), tf.argmax(Y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

The training step and the initialization can be defined as:

```
# init
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)
```

Now is the time to run the training.

```
LEARNING_RATE = 0.01
#cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y),
reduction_indices=[1]))
training =
tf.train.GradientDescentOptimizer(LEARNING_RATE).minimize(cross_entropy)
#correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
#accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
TRAIN_STEPS = 1000
for i in range(TRAIN_STEPS+1):
    sess.run(training, feed_dict={X: x_train, Y_: y_train})
    if i%100 == 0:
        print('Training Step:' + str(i) + '  Accuracy =  ' +
str(sess.run(accuracy, feed_dict={X: x_test, Y_: y_test})) + '  Loss = ' +
str(sess.run(cross_entropy, {X: x_train, Y_: y_train})))
```

```
Training Step:0  Accuracy =  0.1135  Loss = 229.63834
Training Step:100  Accuracy =  0.7618  Loss = 91.94073
Training Step:200  Accuracy =  0.8766  Loss = 47.96678
Training Step:300  Accuracy =  0.9035  Loss = 36.222427
Training Step:400  Accuracy =  0.9149  Loss = 30.9825
Training Step:500  Accuracy =  0.9241  Loss = 27.526648
Training Step:600  Accuracy =  0.9308  Loss = 24.82506
Training Step:700  Accuracy =  0.9361  Loss = 22.577211
Training Step:800  Accuracy =  0.9401  Loss = 20.67293
Training Step:900  Accuracy =  0.9428  Loss = 19.04895
Training Step:1000  Accuracy =  0.947  Loss = 17.651802
```

You can change the activation function to ReLU using: *tf.nn.relu*

Drop out

Tensorflow can make dropout algorithm. By default it can keep 75% of the nodes during the training. Let's do this and consider ReLU as the activation function:

```
# Input and output
X = tf.placeholder(tf.float32, shape=[None, 784]) # input image 28*28 = 784
Y_ = tf.placeholder(tf.float32, shape=[None, 10])  #0-9 digits (10 classes)

pkeep = tf.placeholder(tf.float32)

L = 100
M = 30
N = 60
O = 30

# tf.truncated_normal is a TensorFlow function that produces random values
following the normal (Gaussian) distribution between -2*stddev and +2*stddev
W1 = tf.Variable(tf.truncated_normal([784, L], stddev=0.1))  # 784 = 28 * 28
B1 = tf.Variable(tf.ones([L])/10)
W2 = tf.Variable(tf.truncated_normal([L, M], stddev=0.1))
B2 = tf.Variable(tf.ones([M])/10)
W3 = tf.Variable(tf.truncated_normal([M, 10], stddev=0.1))
B3 = tf.Variable(tf.zeros([10]))

# The model
XX = tf.reshape(X, [-1, 784])
Y1 = tf.nn.relu(tf.matmul(XX, W1) + B1)
Y1d = tf.nn.dropout(Y1, pkeep)
Y2 = tf.nn.relu(tf.matmul(Y1, W2) + B2)
Y2d = tf.nn.dropout(Y2, pkeep)
Ylogits = tf.matmul(Y2d, W3) + B3
Y = tf.nn.softmax(Ylogits)
```

Look at the way that bias is defined this time. When you use Tensorflow, it is better to initialize bias to be a very small value rather than zero.

Now let us run the code as we did before. We should tell the software, how many of the neurons will be kept in training phase and how many in test phase. We keep 75% in training but will keep all of them in testing.

```
# cross-entropy loss function (= -sum(Y_i * log(Yi)) ), normalised for
batches of 100  images
# TensorFlow provides the softmax_cross_entropy_with_logits function to avoid
numerical stability
# problems with log(0) which is NaN
cross_entropy = tf.nn.softmax_cross_entropy_with_logits_v2(logits=Ylogits,
labels=Y_)
cross_entropy = tf.reduce_mean(cross_entropy)*100
```

```
# accuracy of the trained model, between 0 (worst) and 1 (best)
correct_prediction = tf.equal(tf.argmax(Y, 1), tf.argmax(Y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

# init
init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

LEARNING_RATE = 0.01
#cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y),
reduction_indices=[1]))
training =
tf.train.GradientDescentOptimizer(LEARNING_RATE).minimize(cross_entropy)
#correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
#accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
TRAIN_STEPS = 1000
for i in range(TRAIN_STEPS+1):
    sess.run(training, feed_dict={X: x_train, Y_: y_train, pkeep: 0.75 })
    if i%100 == 0:
        print('Training Step:' + str(i) + '  Accuracy =  ' +
str(sess.run(accuracy, feed_dict={X: x_test, Y_: y_test, pkeep: 1.0})) + '
Loss = ' + str(sess.run(cross_entropy, {X: x_train, Y_: y_train , pkeep:
0.75})))
```

```
Training Step:0  Accuracy =  0.2804  Loss = 220.09549
Training Step:100  Accuracy =  0.888  Loss = 45.822235
Training Step:200  Accuracy =  0.9396  Loss = 25.82566
Training Step:300  Accuracy =  0.9238  Loss = 28.617971
Training Step:400  Accuracy =  0.9562  Loss = 16.885233
Training Step:500  Accuracy =  0.9619  Loss = 14.017251
Training Step:600  Accuracy =  0.9576  Loss = 15.326747
Training Step:700  Accuracy =  0.9265  Loss = 36.81757
Training Step:800  Accuracy =  0.9475  Loss = 25.138563
Training Step:900  Accuracy =  0.9511  Loss = 22.313171
Training Step:1000  Accuracy =  0.9579  Loss = 18.196114
```

Step 3. Stochastic Gradient Descent

AS we discussed in class, the best way of addressing the deep neural network is through making smaller batches of data. Please make batches of 200 data and design a deep neural network with four hidden layer, ReLU as hidden layer activation function and softmax as the output activation layer. Please consider dropout for all the layers.