# CS 590 Machine Learning
# Homework 08 CNN
# (Ashwini Kulkarni)

**Code Walkthrough: for trainMode is true**

```
import tensorflow as tf
from random import random
import loadData #used for its batch function

trainMode = True #if true, then we are training the network on data and saving to file
                 #if false, then we are loading the network from file and running on someth

#trainMode settings:
numIterations = 75000

#non-trainMode settings
checkpointFile = "savedNetwork-62000" #the file to load from (in non-trainMode)
```

The TensorFlow API is composed of a set of Python modules that enable constructing and executing TensorFlow graphs

Importing random package for use of random() function

trainMode conditional flag

numIterations : iterations

**If its TrainMode =true =>**

```
if trainMode:
    #LOAD DATA
    print "Creating data..."
    X_train, y_train, X_test, y_test = [[],[],[],[]]
    #X_train is a list [X1, ..., Xn] where each Xi = [xi.1, ..., xi.m], and each xi.j is between 0 and 1.
    #y_train is a list [y1, ..., yn] where each yi = [yi.1, ..., yi.p], and each yi.j is between 0 and 1.
    #X_test is a list [X1, ..., Xq] where each Xi = [xi.1, ..., xi.m], and each xi.j is between 0 and 1.
    #y_test is a list [y1, ..., yq] where each yi = [yi.1, ..., yi.p], and each yi.j is between 0 and 1.

    #populate data
    for i in range(10000000):
        thisX = []
        thisY = 0.0
        for i in range(10):
            r = random()
            thisX.append(r)
            thisY += r
        X_train.append(thisX)
        y_train.append([thisY/10])
    for i in range(100):
        thisX = []
        thisY = 0.0
        for i in range(10):
            r = random()
            thisX.append(r)
            thisY += r
        X_test.append(thisX)
        y_test.append([thisY/10])
else: #not trainMode
    #LOAD DATA
    #fill the data to test
    X_test = []
    for j in range(10):
        X = []
        for i in range(10):
            r = random()
            X.append(r)
        X_test.append(X)

sess = tf.InteractiveSession()

#inputs and target outputs
```

If trainMode is True, Then we are creating dataset for train and test such that,
X_train instance will have 10 attribute which will hold float value between 0 to and its generated randomly using random () from random package which returns values between 0 to 1.
As per the code outer loop (line56), we are going to create/load train data with 10 million records/instances. And each instance have 10 attributes (line59). Target will be mean of all values for all attribute for that particular instance.
Similarly, creating/loading test data with 100 instances.
So the result of above block of code will be

| | |
|---|---|
| TrainDataset | 10million instances |
| Attributes | 10(x1,x2,x3..,x10) |
| Target | Mean(y) |
| Test DataSet | 100 instances |

```
85  sess = tf.InteractiveSession()
86
87  #inputs and target outputs
88  x = tf.placeholder("float", shape=[None, 10]) #input
89  y_ = tf.placeholder("float", shape=[None, 1]) #output
90
91  #layers
92  fc1 = fullyConnectedLayer(x, 10, 20)
93  y_pred = fullyConnectedLayer(fc1, 20, 1)
94
95  cross_entropy = tf.square(y_pred-y_) #squared distance
96  train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
97  accuracy = tf.reduce_sum(tf.square(y_pred-y_))
```

Note:- Here we are creating interactive session, which allows us to run variables without needing to constantly refer to the session object.
Using Placeholder : creating nodes for the input and target data
tf.placeholder(dtype, shape=None, name=None)
as generated random values are of floattype so datatype will be float.
Shape specifies dimension of data which we are going to fed to graph. (None: 1st dimension is batch size)
After setting properties for input and outpute(line 88,89)
Call to function fullyConnectedLayer
Parameters: x, input size and output size

```
#takes a FLAT input layer and fully connects it to another, using ReLU.
def fullyConnectedLayer(inputTensor, inputSize, outputSize):
    W = weight_variable([inputSize,outputSize])
    B = bias_variable([outputSize])
    #return tf.nn.relu(tf.matmul(inputTensor, W) + B)
    return tf.nn.sigmoid(tf.matmul(inputTensor,W) + B)
```

Calculates:
It Call weight_varable() and bias_variable() and and computes sigmoid of activation function.
Returns: sigmoid value for activation function.
After getting output of first convolution layer it fed to second layer and fuction will return value which is nothing but predicted target value.

weight_varable() and bias_variable()

```
#creates a variable that has random values. Remember that every time eval() is called,
#the values will be re-generated!
def weight_variable(shape):
  initial = tf.truncated_normal(shape, stddev=0.1)
  return tf.Variable(initial)

#creates a variable that has constant values of 0.1.
def bias_variable(shape):
  initial = tf.constant(0.1, shape=shape)
  return tf.Variable(initial)

```

Here we are initializing weights and positive bias.

After getting y_predicted, calculating cross entropy cost functions:

```

cross_entropy = tf.square(y_pred-y_) #squared distance
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
accuracy = tf.reduce_sum(tf.square(y_pred-y_))

```

Using ADAM algorithm(for Gradient descent optimization algorithms
) Passing Learning rate and By using minimize for loss as train_step. And then calculating accuracy using reduce_sum method.

The returned operation train_step, when run, will apply the gradient descent updates to the parameters. Training the model can therefore be accomplished by repeatedly running train_step.

```

if trainMode:
    sess.run(tf.initialize_all_variables())
    saver = tf.train.Saver()
    batchPointer = 0 #points to the index in X_train that is next up
    for i in range(numIterations):
      batch = loadData.next_batch(X_train, y_train, batchPointer, 50)
      batchPointer = (batchPointer+50)%len(y_train)
      if i%100 == 0:
        train_accuracy = accuracy.eval(feed_dict={x:batch[0], y_: batch[1]})
        print "step %d, squared error %g"%(i, train_accuracy)
      if i%1000 == 0:
        print "saving checkpoint..."
        saver.save(sess, "savedNetwork", global_step=i)
      try:
        train_step.run(feed_dict={x: batch[0], y_: batch[1]})
      except:
#       for L in batch:
#           for LL in L:
#               for LLL in LL:
#                   if
        print "\n\nbatch[0]:"
        print batch[0]
        print "\n\n\n\nbatch[1]:"
        print batch[1]
        exit()

    print "FINAL SQUARED ERROR %g"%accuracy.eval(feed_dict={
        x: X_test, y_: y_test})
```

After defining model.

If trainMode is true:

We will run session with initializing all variables.

Tf.train.Saver() used for Saving and restoring variables.

X train | Y_train
$x_1$ $x_2$ ... $x_{10}$ | Y

Location 0

$1^{st}$ interation of for loop defined at line 103

batchpoint = $0^{th}$ Location

loading data in batch 0 to 50

batchpoint = new pointer location for next iteration.

that will be $50^{th}$ location

after every 100 iterations it will get train accuracy & display it

after every 1000 iteration.

session variable will be saved.

run trainstep with assigning ~~val~~ values to ~~tensor~~ for current batch

location 50

$2^{nd}$ iteration.

**After training all data to model, we can evaluate model by feeding test data and calculate accuracy using accuracy.eval(feed_dict={x: X_test, y_: y_test}).**