# hwalsuklee\_tensorflow-mnist-cnnmnist\_cnn\_train

# Some code was borrowed from https://github.com/petewarden/tensorflow\_makefile/blob/master/tensorflow/models/image/mnist/convolutional.py  
  
from \_\_future\_\_ import absolute\_import  
from \_\_future\_\_ import division  
from \_\_future\_\_ import print\_function  
  
import numpy  
  
import tensorflow as tf  
import tensorflow.contrib.slim as slim  
  
import mnist\_data  
import cnn\_model  
  
MODEL\_DIRECTORY = "model/model.ckpt"  
LOGS\_DIRECTORY = "logs/train"  
  
# Params for Train  
training\_epochs = 10 # 10 for augmented training data, 20 for training data  
TRAIN\_BATCH\_SIZE = 50  
display\_step = 100  
validation\_step = 500  
  
# Params for test  
TEST\_BATCH\_SIZE = 5000  
  
  
def train():  
 # Some parameters  
 batch\_size = TRAIN\_BATCH\_SIZE  
 num\_labels = mnist\_data.NUM\_LABELS  
  
 # Prepare mnist data  
 train\_total\_data, train\_size, validation\_data, validation\_labels, test\_data, test\_labels = mnist\_data.\  
 prepare\_MNIST\_data(True)  
  
 # Boolean for MODE of train or test  
 is\_training = tf.placeholder(tf.bool, name='MODE')  
  
 # tf Graph input  
 x = tf.placeholder(tf.float32, [None, 784])  
 y\_ = tf.placeholder(tf.float32, [None, 10]) # answer  
  
 # Predict  
 y = cnn\_model.CNN(x)  
  
 # Get loss of model  
 with tf.name\_scope("LOSS"):  
 loss = slim.losses.softmax\_cross\_entropy(y, y\_)  
  
 # Create a summary to monitor loss tensor  
 tf.scalar\_summary('loss', loss)  
  
 # Define optimizer  
 with tf.name\_scope("ADAM"):  
 # Optimizer: set up a variable that's incremented once per batch and  
 # controls the learning rate decay.  
 batch = tf.Variable(0)  
  
 learning\_rate = tf.train.exponential\_decay(  
 1e-4, # Base learning rate.  
 batch\*batch\_size, # Current index into the dataset.  
 train\_size, # Decay step.  
 0.95, # Decay rate.  
 staircase=True)  
 # Use simple momentum for the optimization.  
 train\_step = tf.train.AdamOptimizer(learning\_rate).minimize(loss, global\_step=batch)  
  
 # Create a summary to monitor learning\_rate tensor  
 tf.scalar\_summary('learning\_rate', learning\_rate)  
  
 # Get accuracy of model  
 with tf.name\_scope("ACC"):  
 correct\_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y\_, 1))  
 accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))  
  
 # Create a summary to monitor accuracy tensor  
 tf.scalar\_summary('acc', accuracy)  
  
 # Merge all summaries into a single op  
 merged\_summary\_op = tf.merge\_all\_summaries()  
  
 # Add ops to save and restore all the variables  
 saver = tf.train.Saver()  
 sess = tf.InteractiveSession()  
 sess.run(tf.global\_variables\_initializer(), feed\_dict={is\_training: True})  
  
 # Training cycle  
 total\_batch = int(train\_size/batch\_size)  
  
 # op to write logs to Tensorboard  
 summary\_writer = tf.train.SummaryWriter(LOGS\_DIRECTORY, graph=tf.get\_default\_graph())  
  
 # Save the maximum accuracy value for validation data  
 max\_acc = 0.  
  
 # Loop for epoch  
 for epoch in range(training\_epochs):  
  
 # Random shuffling  
 numpy.random.shuffle(train\_total\_data)  
 train\_data\_ = train\_total\_data[:, :-num\_labels]  
 train\_labels\_ = train\_total\_data[:, -num\_labels:]  
  
 # Loop over all batches  
 for i in range(total\_batch):  
  
 # Compute the offset of the current minibatch in the data.  
 offset = (i\*batch\_size)%(train\_size)  
 batch\_xs = train\_data\_[offset:(offset + batch\_size), :]  
 batch\_ys = train\_labels\_[offset:(offset + batch\_size), :]  
  
 # Run optimization op (backprop), loss op (to get loss value)  
 # and summary nodes  
 \_, train\_accuracy, summary = sess.run([train\_step, accuracy, merged\_summary\_op],  
 feed\_dict={x: batch\_xs, y\_: batch\_ys, is\_training: True})  
  
 # Write logs at every iteration  
 summary\_writer.add\_summary(summary, epoch\*total\_batch + i)  
  
 # Display logs  
 if i%display\_step == 0:  
 print("Epoch:", '%04d,'%(epoch + 1),  
 "batch\_index %4d/%4d, training accuracy %.5f"%(i, total\_batch, train\_accuracy))  
  
 # Get accuracy for validation data  
 if i%validation\_step == 0:  
 # Calculate accuracy  
 validation\_accuracy = sess.run(accuracy,  
 feed\_dict={x: validation\_data, y\_: validation\_labels,  
 is\_training: False})  
  
 print("Epoch:", '%04d,'%(epoch + 1),  
 "batch\_index %4d/%4d, validation accuracy %.5f"%(i, total\_batch, validation\_accuracy))  
  
 # Save the current model if the maximum accuracy is updated  
 if validation\_accuracy > max\_acc:  
 max\_acc = validation\_accuracy  
 save\_path = saver.save(sess, MODEL\_DIRECTORY)  
 print("Model updated and saved in file: %s"%save\_path)  
  
 print("Optimization Finished!")  
  
 # Restore variables from disk  
 saver.restore(sess, MODEL\_DIRECTORY)  
  
 # Calculate accuracy for all mnist test images  
 test\_size = test\_labels.shape[0]  
 batch\_size = TEST\_BATCH\_SIZE  
 total\_batch = int(test\_size/batch\_size)  
  
 acc\_buffer = []  
  
 # Loop over all batches  
 for i in range(total\_batch):  
 # Compute the offset of the current minibatch in the data.  
 offset = (i\*batch\_size)%(test\_size)  
 batch\_xs = test\_data[offset:(offset + batch\_size), :]  
 batch\_ys = test\_labels[offset:(offset + batch\_size), :]  
  
 y\_final = sess.run(y, feed\_dict={x: batch\_xs, y\_: batch\_ys, is\_training: False})  
 correct\_prediction = numpy.equal(numpy.argmax(y\_final, 1), numpy.argmax(batch\_ys, 1))  
 acc\_buffer.append(numpy.sum(correct\_prediction)/batch\_size)  
  
 print("test accuracy for the stored model: %g"%numpy.mean(acc\_buffer))  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 train()