Project #1

EE488C Special Topics in EE <Deep Learning & AlphaGo>
2016 Fall, School of EE, KAIST

- Contents
  - Installing & using TensorFlow
  - Learning XOR using TensorFlow
  - MNIST handwritten digit recognition using TensorFlow
  - Appendix: Useful tips
    - Showing images in MNIST dataset
    - MNIST handwritten digit recognition using TFLearn
    - TensorBoard: A tool for visualization learning

- Installing TensorFlow
  - You can use one of 24 computers to run TensorFlow codes for this project.
  - Detailed remote login instructions are given in “EE488C Login Howto.pdf”.
  - If you want to install TensorFlow on your computer, follow instructions in https://www.tensorflow.org
  - If you want to use VirtualBox to run TensorFlow on your Windows PC, follow instructions given in “EE488C VirtualBox Howto.pdf”
  - For installing python, read page 32 in Lecture note #3.

- Using TensorFlow
  - Read “Get Started” and “Tutorials” on https://www.tensorflow.org before trying codes in this project.
  - Refer to https://www.tensorflow.org/versions/r0.11/api_docs/python/index.html for usage of TensorFlow Python API’s.

- Source codes for Project #1
  - Download projet1.zip from KLMS that contains all the source codes for Project #1.

- Learning XOR using TensorFlow
  - In this example, we show how to train a simple neural network to learn XOR. We use the same neural network that we learned in Lecture 7, which has one hidden layer with two neurons.
  - The following example code (project1_xor.py) trains the neural network to learn to perform XOR. To demonstrate convergence to the global minimum shown in page 16 in Lecture note #7, we initialize the parameters near the global minimum. Try to see how the learning performance is affected by initialization by choosing different initial values for weights and biases.

```
import tensorflow as tf
import math
import numpy as np

#--------------------
# Parameter Specification
#--------------------
NUM_ITER = 200      # Number of iterations
nh = 2              # Number of hidden neurons
lr_init = 0.2       # Initial learning rate for gradient descent algorithm
lr_final = 0.01     # Final learning rate
var_init = 0.1      # Standard deviation of initializer

# Symbolic variables
x_ = tf.placeholder(tf.float32, shape=[None,2])
y_ = tf.placeholder(tf.float32, shape=[None,1])

# Training data
x_data = [[0,0], [0,1], [1,0], [1,1]]
y_data = [[0], [1], [1], [0]]

# Weight initialization
# Chosen to be an optimal point to demonstrate convergence to global optimum
W_init = [[1.0,-1.0],[-1.0,1.0]]
w_init = [[1.0],[1.0]]
c_init = [[0.0,0.0]]
b_init = 0.

#--------------------
# Layer setting
#--------------------
# Weights and biases
W = tf.Variable(W_init+tf.truncated_normal([2, nh], stddev=var_init))
w = tf.Variable(w_init+tf.truncated_normal([nh, 1], stddev=var_init))
c = tf.Variable(c_init+tf.truncated_normal([nh], stddev=var_init))
b = tf.Variable(b_init+tf.truncated_normal([1], stddev=var_init))

#-- Activation setting --
h = tf.nn.relu(tf.matmul(x_, W)+c)
yhat = tf.matmul(h, w)+b

#-- MSE cost function --
cost = tf.reduce_mean((y_-yhat)**2)

lr = tf.placeholder(tf.float32, shape=[])
train_step = tf.train.GradientDescentOptimizer(lr).minimize(cost)

#--------------------
# Run optimization
#--------------------
sess = tf.Session()

sess.run(tf.initialize_all_variables())
for i in range(NUM_ITER):
    j=np.random.randint(4)    # random index 0~3
    lr_current=lr_init + (lr_final - lr_init) * i / NUM_ITER
    a=sess.run(train_step, feed_dict={x_:[x_data[j]], y_:[y_data[j]], lr: lr_current})
    deploy_cost = sess.run(cost, feed_dict={x_:x_data, y_:y_data})
    deploy_yhat = sess.run(yhat, feed_dict={x_:x_data})
    print('{:2d}: XOR(0,0)={:7.4f}   XOR(0,1)={:7.4f}   XOR(1,0)={:7.4f}   XOR(1,1)={:7.4f}
cost={:.5g}'.\
        format(i+1, float(deploy_yhat[0]), float(deploy_yhat[1]), float(deploy_yhat[2]),
float(deploy_yhat[3]),float(deploy_cost)))

print "W: ", sess.run(W)
print "w: ", sess.run(w)
print "c: ", sess.run(c)
print "b: ", sess.run(b)
```

Listing 1. project1_xor.py

- In the following, we explain the code.
- Parameter specification
  - This part defines the hyperparameters for training. This includes the number of training iterations, the initial & final learning rates, and the standard deviation for random initialization. It also specifies the number of neurons in the hidden layer.
- Symbolic variables
  - x_ = tf.placeholder(tf.float32, shape=[None,2])
    y_ = tf.placeholder(tf.float32, shape=[None,1])
    In TensorFlow, symbolic manipulations are done to calculate gradient automatically. x_ and y_ are symbolic variables whose values are not specified yet. They are place holders for 32-bit floating matrices of sizes None*2 and None*1, respectively. ‘None’ means unspecified. It will be specified later when actual data is fed to x_ and y_ when you run “a=sess.run(train_step, feed_dict={x_:[x_data[j]], y_:[y_data[j]], lr: lr_init})” and “deploy_cost = sess.run(cost, feed_dict={x_:x_data, y_:y_data})”. Note that for the former, only the j-th training example is used for training and thus x_ and y_ will become 1x2 and 1x1, respectively. For the latter, the whole training examples are used for evaluating the cost, therefore x_ and y_ will become 4x2 and 4x1, respectively.
- Training data
  - x_data = [[0,0], [0,1], [1,0], [1,1]]
  - y_data = [[0], [1], [1], [0]]
    This is the training data (input & label).
- Weight initialization
  - W_init = [[1.0,-1.0],[-1.0,1.0]]
  - w_init = [[1.0],[1.0]]
  - c_init = [[0.0,0.0]]
  - b_init = 0.
    These will be used to specify initial weight and bias parameters of the neural network (same notation as in Lecture note #7). Actual initial values for weights and biases will include random noise, which will be added later. The above parameters are chosen such that the initial point is near the global optimum point introduced in page 16 of Lecture note #7. If the initial point is sufficiently near the global optimum, gradient descent will be able to find a global optimum and this program will demonstrate it. You can try to change the initial point to see whether gradient descent will be stuck at a local minimum, will exhibit slowing down behavior near a saddle point, will oscillate, or will diverge.

- Neural network specification
  - W = tf.Variable(W_init+tf.truncated_normal([2, nh], stddev=var_init))
  - w = tf.Variable(w_init+tf.truncated_normal([nh, 1], stddev=var_init))
  - c = tf.Variable(c_init+tf.truncated_normal([nh], stddev=var_init))
  - b = tf.Variable(b_init+tf.truncated_normal([1], stddev=var_init))
  - In TensorFlow, a Variable is a modifiable tensor and neural network parameters (weights and biases) are defined as Variables. The above four lines define weights and biases that will be initialized as specified. Note that some randomness is added to make the initial point slightly different from W_init, w_init, c_init, and b_init. A truncated-normal noise with mean 0 and standard deviation var_init is used. Truncation is done so that the absolute value does not exceed 2 times the standard deviation. If we do not specify the mean and standard deviation of the truncated normal function, default values are 0 and 1, respectively.
  - h = tf.nn.relu(tf.matmul(x_, W)+c)
  - y_hat = tf.matmul(h, w)+b
    In the first layer, we multiply the input to the neural network 'x_' by the weight matrix 'W' and add the bias term 'c'. Then, we apply the ReLU activation function to the resulting vector. In the second layer, we multiply the output of the first layer by the weight matrix 'w' and add the bias term 'b'. There is no non-linear activation at the second layer.
- Training and evaluation
  - cost = tf.reduce_mean((y_-y_hat)**2)
    This specifies the cost function. In this case, we use the mean square error between the training data and the neural network output. Reduce_mean calculates the average over all examples in a batch.
  - lr = tf.placeholder(tf.float32, shape=[])
    This is the learning rate for the gradient descent optimizer.
  - train_step = tf.train.GradientDescentOptimizer(lr).minimize(cost)
    This specifies the training step, i.e., use the gradient descent optimizer with learning rate 'lr' to minimize 'cost' function defined above. TensorFlow will automatically perform backpropagation using symbolic differentiation when calculating the gradient. Therefore, you don't need to worry about how it is done.
  - sess = tf.Session()
    A session object encapsulates the environment in which operation objects are executed and tensor objects are evaluated. We define a new session 'sess' ahead of the training procedure.
  - Sess.run(tf.initialize_all_variables())
    This initializes all variables defined so far.
  - train_step = tf.train.GradientDescentOptimizer(lr).minimize(cost)
    This specifies the training step, i.e., use the gradient descent optimizer with learning rate 'lr' to minimize 'cost' function defined above. TensorFlow will automatically perform backpropagation using symbolic differentiation when calculating the gradient.
  - for i in range(NUM_ITER):

    ```
    j=np.random.randint(4)    # random index 0~3
    lr_current=lr_init + (lr_final - lr_init) * i / NUM_ITER
    a=sess.run(train_step, feed_dict={x_:[x_data[j]], y_:[y_data[j]], lr: lr_current})
    deploy_cost = sess.run(cost, feed_dict={x_:x_data, y_:y_data})
    deploy_yhat = sess.run(yhat, feed_dict={x_:x_data})
    print('{:2d}: XOR(0,0)={:7.4f}   XOR(0,1)={:7.4f}   XOR(1,0)={:7.4f}  ...
    ```

    We train the neural network for 200 iterations and print the cost and yhat values at each iteration. 'lr_current' is the current learning rate, which changes from 'lr_init' to 'lr_final'.
  - The last four lines print out the values of 'W', 'w', 'c', and 'b' after training is finished.

- Run the code by typing "python project1_xor.py". You will get something like the following, which shows 'cost' converges to a small value after 200 iterations and y_hat values become close to desired values. It also shows the values of 'W', 'w', 'c', and 'b' after the training is done.

```
 1: XOR(0,0)=-0.2281   XOR(0,1)= 0.6871   XOR(1,0)=-0.1545   XOR(1,1)=-0.2281   cost=0.38369
 2: XOR(0,0)= 0.2315   XOR(0,1)= 1.1467   XOR(1,0)= 0.9131   XOR(1,1)= 0.3805   cost=0.05686
 3: XOR(0,0)= 0.2659   XOR(0,1)= 1.1811   XOR(1,0)= 1.0199   XOR(1,1)= 0.4693   cost=0.081041
 4: XOR(0,0)= 0.1611   XOR(0,1)= 1.0762   XOR(1,0)= 0.9151   XOR(1,1)= 0.3645   cost=0.042952
 5: XOR(0,0)= 0.1944   XOR(0,1)= 1.1096   XOR(1,0)= 1.0250   XOR(1,1)= 0.4557   cost=0.064521
 6: XOR(0,0)= 0.1516   XOR(0,1)= 0.9575   XOR(1,0)= 0.9822   XOR(1,1)= 0.4129   cost=0.048903
 7: XOR(0,0)= 0.1618   XOR(0,1)= 0.9644   XOR(1,0)= 1.0062   XOR(1,1)= 0.4327   cost=0.053688
 8: XOR(0,0)= 0.1756   XOR(0,1)= 1.0105   XOR(1,0)= 1.0199   XOR(1,1)= 0.4465   cost=0.057672
 9: XOR(0,0)= 0.1047   XOR(0,1)= 0.9430   XOR(1,0)= 0.8979   XOR(1,1)= 0.3246   cost=0.032507
10: XOR(0,0)= 0.1438   XOR(0,1)= 0.9821   XOR(1,0)= 1.0328   XOR(1,1)= 0.4374   cost=0.053351
...
191: XOR(0,0)= 0.0000   XOR(0,1)= 1.0000   XOR(1,0)= 1.0000   XOR(1,1)= 0.0000   cost=2.7494e-10
192: XOR(0,0)= 0.0000   XOR(0,1)= 1.0000   XOR(1,0)= 1.0000   XOR(1,1)= 0.0000   cost=2.6154e-10
193: XOR(0,0)= 0.0000   XOR(0,1)= 1.0000   XOR(1,0)= 1.0000   XOR(1,1)= 0.0000   cost=2.6779e-10
194: XOR(0,0)= 0.0000   XOR(0,1)= 1.0000   XOR(1,0)= 1.0000   XOR(1,1)= 0.0000   cost=2.6769e-10
195: XOR(0,0)= 0.0000   XOR(0,1)= 1.0000   XOR(1,0)= 1.0000   XOR(1,1)= 0.0000   cost=2.6744e-10
196: XOR(0,0)= 0.0000   XOR(0,1)= 1.0000   XOR(1,0)= 1.0000   XOR(1,1)= 0.0000   cost=2.6555e-10
197: XOR(0,0)= 0.0000   XOR(0,1)= 1.0000   XOR(1,0)= 1.0000   XOR(1,1)= 0.0000   cost=2.6501e-10
198: XOR(0,0)= 0.0000   XOR(0,1)= 1.0000   XOR(1,0)= 1.0000   XOR(1,1)= 0.0000   cost=2.6376e-10
199: XOR(0,0)= 0.0000   XOR(0,1)= 1.0000   XOR(1,0)= 1.0000   XOR(1,1)= 0.0000   cost=2.635e-10
200: XOR(0,0)= 0.0000   XOR(0,1)= 1.0000   XOR(1,0)= 1.0000   XOR(1,1)= 0.0000   cost=2.6277e-10
W:  [[ 0.96100384 -1.892079  ]
 [-0.92295176  1.2204988 ]]
w:  [[ 1.08343947]
 [ 0.91188282]]
c:  [[-0.03805346 -0.12388773]]
b:  [  1.55943162e-05]
```

- Task #1
  - Modify the initial point (W,w,c,b) in "project1_xor.py" such that the gradient descent algorithm converges to a bad local minimum.
  - Submit your code and the output of your code that includes the outputs of the neural network and the cost at each iteration and the values of W,w,c,b at the end of iterations.
  - To submit your code and the output, simply place your code in your home directory. Its name should be "project1_task1.py". To save the output of your program as a text file in your home directory, do "python project1_task1.py > project1_task1.out". Then, the output file "project1_task1.out" will be saved.
  - For example, if your machine is 143.248.141.84 and your id is s25, then your home directory should contain "project1_task1.py" and "project1_task1.out" as shown below.
  - Even if you use your own computer, you should copy the files to your home directory of the machine assigned to you. Refer to "EE488C Login Howto.pdf" for the IP address of the machine you are supposed to use and your login id.
  - Immediately after the deadline, all the files will be collected automatically. Double check the file name since if the file name is different, it will not be collected.

```
s25@EE2354-4:~$ ls -l
total 20
-rw-r--r-- 1 s25 s25 8980  9월 29 19:30 examples.desktop
-rw-rw-r-- 1 s25 s25   14 10월 26 16:04 project1_task1.out
-rw-rw-r-- 1 s25 s25   25 10월 26 16:04 project1_task1.py
s25@EE2354-4:~$
```

- MNIST handwritten digit recognition using TensorFlow (project1_mnist_*.py)
    - MNIST is database of handwritten digits, 0~9. This dataset has a training set with 55,000 images, a validation set with 5,000 images, and a test set with 10,000 images. Each image is gray scale and its dimension is 28 pixels by 28 pixels.
    - In this section, we take a look at several implementations to classify MNIST from a simple softmax regression to a convolutional neural network. You will notice that the accuracy of classification gets better as the neural network becomes more complex.
    - The following implementation uses a simple softmax regression.

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets('./MNIST_data', one_hot=True)
sess = tf.InteractiveSession()
x = tf.placeholder(tf.float32, shape=[None, 784])
y_ = tf.placeholder(tf.float32, shape=[None, 10])

# Implement regression
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
y = tf.nn.softmax(tf.matmul(x, W) + b)

# Train and Evaluate the Model
cross_entropy = - tf.reduce_sum(y_*tf.log(y))
train_step = tf.train.GradientDescentOptimizer(5e-3).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))

sess.run(tf.initialize_all_variables())
print("=================================")
print("|Epoch\tBatch\t|Train\t|Val\t|")
print("|===============================|")
for j in range(1):
    for i in range(550):
        batch = mnist.train.next_batch(100)
        train_step.run(feed_dict={x: batch[0], y_: batch[1]})
        if i%50 == 49:
            train_accuracy = accuracy.eval(feed_dict={x:batch[0], y_: batch[1]})
            val_accuracy = accuracy.eval(feed_dict=\
                {x: mnist.validation.images, y_:mnist.validation.labels})
            print("|%d\t|%d\t|%.4f\t|%.4f\t|"%(j+1, i+1, train_accuracy, val_accuracy))
print("|===============================|")
test_accuracy = accuracy.eval(feed_dict=\
    {x: mnist.test.images, y_:mnist.test.labels})
print("test accuracy=%.4f"%(test_accuracy))
```

Listing 2. project1_mnist_softmax.py

- In the following, we explain the above code.
    - Loading MNIST dataset & session definition
        - mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
          This loads the MNIST dataset. “one_hot=True” means the labels are one-hot vectors, e.g., [1,0,0,...,0], [0,1,0,...,0], … , or [0,0,0,…,1].
        - sess = tf.InteractiveSession()
          Here we define a session object.

- Softmax regression
  - x = tf.placeholder(tf.float32, shape=[None, 784])
    y_ = tf.placeholder(tf.float32, shape=[None, 10])
    TensorFlow placeholders with sizes equal to the MNIST image size 784=28x28, and digit category 0~9.
  - W = tf.Variable(tf.zeros([784, 10]))
    defines 784 by 10 weight matrix from 784 input dimensions to 10 output dimensions. We initialize this matrix with zeros for simplicity, but you can try other initializations such as truncated normal.
  - b = tf.Variable(tf.zeros([10]))
    defines the bias vector of size 10x1. We also initialize this vector with zeros.
  - y = tf.nn.softmax(tf.matmul(x, W) + b)
    Finally, we compute the softmax output values using ‘x’, ‘W’, and ‘b’. The i-th output of the softmax function is defined as $\frac{\exp x_i}{\sum_{k=1}^{n} \exp x_k}$, where x_i is the i-th input of the softmax layer. This means softmax performs component-wise exponential function with normalization. Therefore the output of the softmax layer behaves as a probability mass function (i.e., each output is between 0 and 1 and the sum of all outputs is 1) and indicates the likelihood of each output.
- Training and evaluation
  - cross_entropy = -tf.reduce_sum(y_*tf.log(y))
    This defines the cross entropy cost function between the target variable and the estimation. The function ‘tf.reduce_sum’ computes the sum of elements across dimensions of a tensor.
  - train_step = tf.train.GradientDescentOptimizer(5e-3).minimize(cross_entropy)
    We train the neural network to minimize the cross entropy. Here, we use gradiend descent optimizer with learning rate 0.01.
  - correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
    This checks whether our estimation is correct or not. The function ‘tf.argmax’ finds the index of the maximum value in the vector, and the function tf.equal returns true if and only if the two arguments are the same.
  - accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
    This returns the accuracy of our prediction. Since the return type of tf.equal is bool, we must convert its type to floating-point to make it real (0 or 1). Then the tf.reduce_mean function computes the average.
  - sess.run(tf.initialize_all_variables())
    To use the variables within a session, we must initialize them.
  -
    ```
    for j in range(1):
      for i in range(550):
        batch = mnist.train.next_batch(100)
        train_step.run(feed_dict={x: batch[0], y_: batch[1]})
        if i%10 == 9:
          train_accuracy = accuracy.eval(feed_dict={x:batch[0], y_: batch[1]})
          val_accuracy = accuracy.eval(feed_dict=\
            {x: mnist.validation.images, y_:mnist.validation.labels})
          print("|%d\t|%d\t|%4.4f\t|%4.4f\t|"%(j+1, i+1, train_accuracy, val_accuracy))
    ```

    We run only one epoch since there’s already enough training data and this model is so simple that its performance does not improve much even if we increase the number of epochs. For each minibatch, we take 100 images by ‘batch = mnist.train.next_batch(100)’. ‘batch’ contains 100 images and the labels. We don’t do random shuffling of training examples here since the ordering of MNIST data is pretty random already. ‘batch[0]’ contains 100 images and ‘batch[1]’ contains 100 labels. To cover all 55,000 images in the train set, we perform 550 iterations. We print out the training and validation accuracy at every 10th iteration. After all iterations are done, we print out the test accuracy.

- The following code uses one convolutional layer and two fully connected layers (project1_mnist_conv1.py)

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets('./MNIST_data', one_hot=True)
sess = tf.InteractiveSession()
x = tf.placeholder(tf.float32, shape=[None, 784])
y_ = tf.placeholder(tf.float32, shape=[None, 10])

# Convolutional layer
x_image = tf.reshape(x, [-1,28,28,1])
W_conv = tf.Variable(tf.truncated_normal([5, 5, 1, 30], stddev=0.1))
b_conv = tf.Variable(tf.constant(0.1, shape=[30]))
h_conv = tf.nn.conv2d(x_image, W_conv, strides=[1, 1, 1, 1], padding='VALID')
h_relu = tf.nn.relu(h_conv + b_conv)
h_pool = tf.nn.max_pool(h_relu, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

# Fully-connected layer
W_fc1 = tf.Variable(tf.truncated_normal([12 * 12 * 30, 500], stddev=0.1))
b_fc1 = tf.Variable(tf.constant(0.1, shape=[500]))
h_pool_flat = tf.reshape(h_pool, [-1, 12*12*30])
h_fc1 = tf.nn.relu(tf.matmul(h_pool_flat, W_fc1) + b_fc1)

# Output layer
W_fc2 = tf.Variable(tf.truncated_normal([500, 10], stddev=0.1))
b_fc2 = tf.Variable(tf.constant(0.1, shape=[10]))
y_hat=tf.nn.softmax(tf.matmul(h_fc1, W_fc2) + b_fc2)

# Train and Evaluate the Model
cross_entropy = - tf.reduce_sum(y_*tf.log(y_hat))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_hat,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

sess.run(tf.initialize_all_variables())
print("=================================")
print("|Epoch\tBatch\t|Train\t|Val\t|")
print("|===============================|")
for j in range(5):
    for i in range(550):
        batch = mnist.train.next_batch(100)
        train_step.run(feed_dict={x: batch[0], y_: batch[1]})
        if i%50 == 49:
            train_accuracy = accuracy.eval(feed_dict={x:batch[0], y_: batch[1]})
            val_accuracy = accuracy.eval(feed_dict=\
                {x: mnist.validation.images, y_:mnist.validation.labels})
            print("|%d\t|%d\t|%.4f\t|%.4f\t|"%(j+1, i+1, train_accuracy, val_accuracy))
print("|===============================|")
test_accuracy = accuracy.eval(feed_dict=\
    {x: mnist.test.images, y_:mnist.test.labels})
print("test accuracy=%.4f"%(test_accuracy))
```

Listing 3. project1_mnist_conv1.py

- In the following, we explain the code project1_mnist_conv1.py.
  - Convolutional layer
    - ‘x_image = tf.reshape(x, [-1,28,28,1])
      First we should reshape the input image x to (batch_size)x28x28x1. -1 means indefinite, which will be automatically calculated to match the total size, e.g., it becomes equal to the number of examples in a minibatch during training and it becomes equal to the number of examples in the test set during evaluation using the test set. The last dimension indicates the number of channels of an image, i.e., a monochrome image has 1 channel and a full color RGB image has 3 channels (R, G, B).
    - W_conv = tf.Variable(tf.truncated_normal([5, 5, 1, 30], stddev=0.1))
      This layer consists of 30 convolutional filters and each convolutional filter has size of 5x5x1. The weights of filters are initialized as truncated normal random variables whose mean is 0 and standard deviation is 0.1.
    - b_conv = tf.Variable(tf.constant(0.1, shape=[30]))
      This is a 30x1 bias vector of the convolutional layer. We initialize each component as 0.1.
    - h_conv = tf.nn.conv2d(x_image, W_conv, strides=[1, 1, 1, 1], padding='VALID')
      We apply the 2-dimensional convolutional filter (W_conv) to image (x_image). ‘strides’ specify the decimation factors for each dimension of the input, e.g., stride of two means skipping every other sample. We must fix strides[0]=strides[3]=1, and strides[2] and strides[3] indicate the horizontal and vertical strides, respectively. ‘padding’ means the type of padding algorithm to use. ‘VALID’ means each output of convolution is from a valid region in the input and the output size becomes (28 – 5 + 1) x (28 – 5 + 1) x 30 = 24 x 24 x 30. If padding=‘SAME’ is used, then the output has the same size as the input by applying zero padding at the input, i.e., 28 x 28.
    - h_relu = tf.nn.relu(h_conv + b_conv)
      After applying the convolutional filter, we add the bias vector, and then we apply the ReLU activation function, f(x)=max(0,x).
    - h_pool = tf.nn.max_pool(h_relu, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding=‘SAME')
      We apply the max pooling to the output of ReLU function. We take the maximum over the 2x2 block of the output, and the strides is 2x2. So, there is no overlap among pooling windows. The output of the pooling layer is 12x12x30, i.e., the sizes become half for horizontal and vertical dimensions. Padding=‘SAME’ or ‘VALID’ does not matter here since 24 (input size) is divisible by 2 (stride size). If not, then ‘VALID’ will produce a smaller size than ‘SAME’ since it will discard the last leftover sample. Note that ‘SAME’ does not mean the output size is the same as the input size. The output size will be $\left\lfloor \frac{n}{2} \right\rfloor$ if ‘VALID’ and will be $\left\lceil \frac{n}{2} \right\rceil$ if ‘SAME’.
  - Fully-connected layer
    - W_fc1 = tf.Variable(tf.truncated_normal([12 * 12 * 30, 500], stddev=0.1))
      This layer maps the 12x12x30 feature vector into a 500x1 vector. This layer is called a fully-connected layer since every unit in this layer is connected to every input of this layer. Parameters of the layer are initialized as truncated normal random variables whose mean is 0 and standard deviation is 0.1.
    - b_fc1 = tf.Variable(tf.constant(0.1, shape=[500]))
      This is a 500x1 bias vector of the fully-connected layer. We initialize it with a constant 0.1.
    - h_pool_flat = tf.reshape(h_pool, [-1, 12*12*30])
      We reshape the output of the previous pooling layer to (batch_size)x12x12x30. -1 means indefinite.
    - h_fc1 = tf.nn.relu(tf.matmul(h_pool_flat, W_fc1) + b_fc1)
      After multiplying by W_fc1 and adding the bias, we apply the ReLU activation function.

    - Output layer (second fully connected layer)
        - W_fc2 = tf.Variable(tf.truncated_normal([500, 10], stddev=0.1))
          This layer maps the 500x1 feature vector into 10x1 vector that corresponds to output classes 0 ~ 9. Parameters of the layer are initialized as truncated normal random variables whose mean is 0 and standard deviation is 0.1.
        - b_fc2 = tf.Variable(tf.constant(0.1, shape=[10]))
          This is a 10x1 bias vector of the output layer. We initialize it with a constant 0.1.
        - y_conv=tf.nn.softmax(tf.matmul(h_fc1, W_fc2) + b_fc2)
          Finally, we compute the softmax output values after multiplying h_fc1 by W_fc2 and adding the bias.
- Run project1_mnist_conv1.py to see how much improvement you get in accuracy over project1_mnist_softmax.py.
- More about parameter initialization: We can initialize the parameters of the neural network in various ways. Random initialization is preferred. So far, we used truncated-normal initialization, but other types of initialization are possible as shown below.
    - Zero initialization
        - W_conv = tf.Variable(tf.zeros([5, 5, 1, 30]))
        - b_conv = tf.Variable(tf.zeros([30]))
        - W_fc1 = tf.Variable(tf.zeros([12 * 12 * 30, 500]))
        - b_fc1 = tf.Variable(tf.zeros([500]))
        - W_fc2 = tf.Variable(tf.zeros([500, 10]))
        - b_fc2 = tf.Variable(tf.zeros([10]))
    - Uniform initialization: The function ‘tf.random_uniform’ returns a random value uniform in [0,1).
        - W_conv = tf.Variable(tf.random_uniform([5, 5, 1, 30]))
        - b_conv = tf.Variable(tf.random_uniform([30]))
        - W_fc1 = tf.Variable(tf.random_uniform([12 * 12 * 30, 500]))
        - b_fc1 = tf.Variable(tf.random_uniform([500]))
        - W_fc2 = tf.Variable(tf.random_uniform([500, 10]))
        - b_fc2 = tf.Variable(tf.random_uniform([10]))
- More about activation function: In the above code, we assumed ReLU and softmax activation functions. You can use other activation functions, e.g., tanh as shown below.
    - tanh function
        - h_relu = tf.tanh(h_conv + b_conv)
        - h_fc1 = tf.tanh(tf.matmul(h_pool_flat, W_fc1) + b_fc1)

- Adding one more convolutional layer (project1_mnist_conv2.py): The above code (project_mnist_conv1.py) has only one convolutional layer. In project1_mnist_conv2.py, we show how to add another convolutional layer. Specifically, the following changes are made.
  - After the first convolutional layer, we add the following for the second convolutional layer. The size of the convolutional filter is 3x3 for the second convolutional layer (it was 5x5 for the first convolutional layer). The size of 'h_conv2' is (batch) x (12-3+1) x (12-3+1) x 50 = (batch) x 10 x 10 x 50. The size of 'h_pool2' is (batch) x 5 x 5 x 50.
    - W_conv2 = tf.Variable(tf.truncated_normal([3, 3, 30, 50], stddev=0.1))
    - b_conv2 = tf.Variable(tf.constant(0.1, shape=[50]))
    - h_conv2 = tf.nn.conv2d(h_pool, W_conv2, strides=[1, 1, 1, 1], padding='VALID')
    - h_relu2 = tf.nn.relu(h_conv2 + b_conv2)
    - h_pool2 = tf.nn.max_pool(h_relu2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')
  - Then, we should change the dimension of next fully-connected layer and connect it with the second convolutional layer.
    - W_fc1 = tf.Variable(tf.truncated_normal([5 * 5 * 50, 500], stddev=0.1))
    - b_fc1 = tf.Variable(tf.constant(0.1, shape=[500]))
    - h_pool_flat = tf.reshape(h_pool2, [-1, 5*5*50])
    - h_fc1 = tf.nn.relu(tf.matmul(h_pool_flat, W_fc1) + b_fc1)
- Run project1_mnist_conv2.py. Although it has one more convolutional layer than project1_mnist_conv1.py, its accuracy will be similar. This is because project1_mnist_conv1.py already achieves good enough performance.
- Dropout (project1_mnist_dropout.py): Finally, we show how to add a dropout layer. Dropout is known to be effective to prevent the neural network from overfitting. This operation randomly drops units from the neural network during training. In the code project1_mnist_dropout.py, we will apply dropout after the first fully-connected layer.
  - After the first fully-connected layer, we add the following definitions, which applies dropout to the output h_fc1 with keep probability equal to 'keep_prob', whose value will be specified later.
    - keep_prob = tf.placeholder(tf.float32)
    - h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
  - When computing y_hat, we use 'h_fc_drop' instead of 'h_fc1' as shown below.
    - y_hat=tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)
  - When training and evaluating, 'keep_prob' needs to be specified as follows.
    - train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})
    - train_accuracy = accuracy.eval(feed_dict={x:batch[0], y_: batch[1], keep_prob: 1.})
    - val_accuracy = accuracy.eval(feed_dict={x: mnist.validation.images, y_:mnist.validation.labels, keep_prob: 1.})
    - test_accuracy = accuracy.eval(feed_dict={x: mnist.test.images, y_:mnist.test.labels, keep_prob: 1.})
  - Note that keep_prob is set to 0.5 during training, but it is set to 1.0 for evaluation since we do not want to drop any units during evaluation.
- Run project1_mnist_dropout.py. The number of epochs is set to 20 in project1_mnist_dropout.py for better performance and the test accuracy will be about 99%.

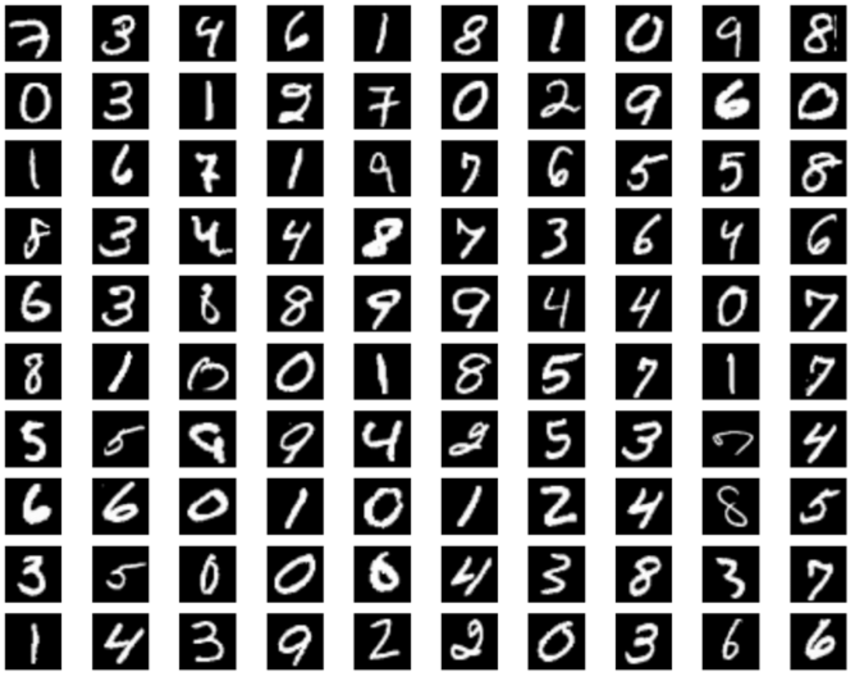
- Task #2
  - Design your own neural network, choose an optimizer, and select values of hyperparameters to maximize the test accuracy. You can start from project1_mnist_dropout.py. But, you are allowed to change the red part only as shown in the next page. Do NOT change any other parts for fairness.
  - You are allowed to have one more layer than the one in project1_mnist_dropout.py, but not more.
  - Convolution kernels should not be bigger than 5x5. The number of channels for convolutional layers should not exceed 50. For example, the number of channels for convolutional layers are 30 and 50 in project1_mnist_dropout.py. The number of neurons in a fully connected layer should not exceed 1024.
  - To submit your code, simply save the python file as “project1_task2.py” and place the file in your home directory in the server computer assigned to you before the deadline. Even if you use your own computer to develop your code, copy your file to your home directory of the server assigned to you. To see which server computer you are supposed to use and your user id, please refer to “EE488C Login Howto.pdf”.
  - Grading will be based on test accuracy. Test accuracy will be based on executing your code by T/A. Therefore, the test accuracy obtained by T/A is expected to be slightly different from that obtained by you due to random initialization.
  - All source codes will be checked for plagiarism and whether all the rules are followed.
  - ***Warning: Only use the training set for training. Never use the test set for training since it is considered cheating.***
  - If you use one of 24 servers, then you may encounter problems when running your program because many students share the servers. There can be up to 5 students using a single machine. You may not be able to run your program if other programs are occupying all the GPU memory. In that case, you can kill other students’ processes if those processes are running for more than 10 minutes. Use “killprocesses” to kill such processes to free up GPU resources.

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets('./MNIST_data', one_hot=True)
sess = tf.InteractiveSession()
x = tf.placeholder(tf.float32, shape=[None, 784])
y_ = tf.placeholder(tf.float32, shape=[None, 10])

# First convolutional layer
x_image = tf.reshape(x, [-1,28,28,1])
W_conv = tf.Variable(tf.truncated_normal([5, 5, 1, 30], stddev=0.1))
b_conv = tf.Variable(tf.constant(0.1, shape=[30]))
h_conv = tf.nn.conv2d(x_image, W_conv, strides=[1, 1, 1, 1], padding='VALID')
h_relu = tf.nn.relu(h_conv + b_conv)
h_pool = tf.nn.max_pool(h_relu, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

# Second convolutional layer
W_conv2 = tf.Variable(tf.truncated_normal([3, 3, 30, 50], stddev=0.1))
b_conv2 = tf.Variable(tf.constant(0.1, shape=[50]))
h_conv2 = tf.nn.conv2d(h_pool, W_conv2, strides=[1, 1, 1, 1], padding='VALID')
h_relu2 = tf.nn.relu(h_conv2 + b_conv2)
h_pool2 = tf.nn.max_pool(h_relu2, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

# Fully-connected Layer
W_fc1 = tf.Variable(tf.truncated_normal([5 * 5 * 50, 500], stddev=0.1))
b_fc1 = tf.Variable(tf.constant(0.1, shape=[500]))
h_pool_flat = tf.reshape(h_pool2, [-1, 5*5*50])
h_fc1 = tf.nn.relu(tf.matmul(h_pool_flat, W_fc1) + b_fc1)

# Dropout
keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

# Output layer
W_fc2 = tf.Variable(tf.truncated_normal([500,10], stddev=0.1))
b_fc2 = tf.Variable(tf.constant(0.1, shape=[10]))
y_hat=tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)

# Train and Evaluate the Model
cross_entropy = - tf.reduce_sum(y_*tf.log(y_hat))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_hat,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

sess.run(tf.initialize_all_variables())
print("=================================")
print("|Epoch\tBatch\t|Train\t|Val\t|")
print("|===============================|")
for j in range(20):
    for i in range(550):
        batch = mnist.train.next_batch(100)
        train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})
        if i%50 == 49:
            train_accuracy = accuracy.eval(feed_dict={x:batch[0], y_: batch[1], keep_prob: 1.})
            val_accuracy = accuracy.eval(feed_dict=\
                {x: mnist.validation.images, y_:mnist.validation.labels, keep_prob: 1.})
            print("|%d\t|%d\t|%.4f\t|%.4f\t|"%(j+1, i+1, train_accuracy, val_accuracy))
print("=================================")
test_accuracy = accuracy.eval(feed_dict=\
    {x: mnist.test.images, y_:mnist.test.labels, keep_prob: 1.})
print("test accuracy=%.4f"%(test_accuracy))
```

Appendix

Useful Tips

- Showing images in the MNIST dataset
  - If you want to plot images in the mnist dataset, run “python project1_mnist_show.py”, which will show the first 100 images in the training set as shown below.



- Running a python program
  - You can run a python script by using “python project1_softmax.py”.
  - You can also run a python script interactively, i.e.,
    - Run python first, i.e., type “python” and ENTER at linux command prompt.
    - At the python prompt, type “execfile(‘project1_softmax.py’)“ and press ENTER. The script will run and print out results.
    - After running the program, you can check the values of variables, e.g., “test_accuracy” as shown below. You can also change the value of a variable or copy-paste some part of your code at the python prompt to run the code line by line. It can be useful for debugging.

```
user@i3-6320:~/ee488/project1$ python
Python 2.7.12 (default, Jul  1 2016, 15:12:24)
...
>>> execfile('project1_mnist_softmax.py')
I tensorflow/stream_executor/dso_loader.cc:116] successfully opened CUDA library libcublas.so
locally
...
|1      |550    |0.9300 |0.9074 |
|===============================|
test accuracy=0.9041
>>> test_accuracy
0.90410006
>>> dir(mnist)   # this will show all the attributes & methods of object 'mnist'
['__add__', '__class__', '__contains__', '__delattr__', '__dict__', ...
>>> dir(W)       # this will show all the attributes & methods of object 'W'
['SaveSliceInfo', '_AsTensor', '_OverloadAllOperators', '_OverloadOperator', ...
>>> sess.run(b)  # to print out the values of 'b'
array([-0.25389311,  0.33093959,  0.02442881, -0.18822268,  0.06158808,
        0.92338496, -0.06923198,  0.4266983 , -1.10328841, -0.15240334], dtype=float32)
```

- MNIST handwritten digit recognition using TFLearn
  - *TFLearn* (http://tflearn.org) is a deep learning library built on top of TensorFlow. It is simpler to use than TensorFlow. Therefore, you may want to consider using it.
  - The following code (project1_mnist_tflearn.py) is adapted from https://github.com/tflearn/tflearn/blob/master/examples/images/convnet_mnist.py to perform essentially the same task as project1_mnist_dropout.py, i.e., the neural network defined below consists of two convolutional layers and two fully-connected layers with dropout. As you can see, the code is simpler with TFLearn than with TensorFlow.
  - You can learn more details by visiting http://tflearn.org.

```
from __future__ import division, print_function, absolute_import

import tflearn
from tflearn.layers.core import input_data, dropout, fully_connected
from tflearn.layers.conv import conv_2d, max_pool_2d
from tflearn.layers.normalization import local_response_normalization
from tflearn.layers.estimator import regression

# Data loading and preprocessing
import tflearn.datasets.mnist as mnist
X, Y, testX, testY = mnist.load_data(one_hot=True)
X = X.reshape([-1, 28, 28, 1])
testX = testX.reshape([-1, 28, 28, 1])

# First convolutional layer
network = input_data(shape=[None, 28, 28, 1], name='input')
network = conv_2d(network, 32, 5, activation='relu')
network = max_pool_2d(network, 2)

# Second convolutional layer
network = conv_2d(network, 64, 5, activation='relu')
network = max_pool_2d(network, 2)

# Fully-connected layer
network = fully_connected(network, 1024, activation='relu')
network = dropout(network, 0.5)

# Output layer
network = fully_connected(network, 10, activation='softmax')
network = regression(network, optimizer='adam', learning_rate=0.001,
                     loss='categorical_crossentropy', name='target')

# Train and Evaluate the Model
model = tflearn.DNN(network, tensorboard_verbose=0)
model.fit({'input': X}, {'target': Y}, n_epoch=20,
           validation_set=({'input': testX}, {'target': testY}),
           snapshot_step=100, show_metric=True, run_id='convnet_mnist')
```

Listing 4. project1_mnist_tflearn.py

- TensorBoard: A visualization tool for TensorFlow
    - TensorFlow provides a tool called TensorBoard for visualization. Specifically, TensorBoard can show computation graphs and plot parameters such as accuracy and cross entropy so that you can easily monitor the whole learning process. Of course, you can simply print out values as done in many examples explained in this project, a visualization tool can be useful. The following example shows an example that uses Tensorboard.

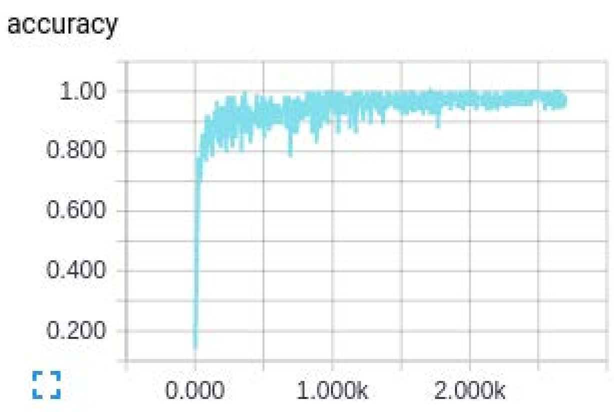
```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data

tf.reset_default_graph()

mnist = input_data.read_data_sets('./MNIST_data', one_hot=True)
sess = tf.InteractiveSession()

with tf.name_scope('Input'):
    x = tf.placeholder(tf.float32, shape=[None, 784])
    y_ = tf.placeholder(tf.float32, shape=[None, 10])

# Convolutional layer
with tf.name_scope('Conv1'):
    x_image = tf.reshape(x, [-1,28,28,1])
    W_conv = tf.Variable(tf.truncated_normal([5, 5, 1, 30], stddev=0.1))
    b_conv = tf.Variable(tf.constant(0.1, shape=[30]))
    h_conv = tf.nn.conv2d(x_image, W_conv, strides=[1, 1, 1, 1], padding='VALID')
    h_relu = tf.nn.relu(h_conv + b_conv)
    h_pool = tf.nn.max_pool(h_relu, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME')

# Fully-connected layer
with tf.name_scope('FC'):
    W_fc1 = tf.Variable(tf.truncated_normal([12 * 12 * 30, 500], stddev=0.1))
    b_fc1 = tf.Variable(tf.constant(0.1, shape=[500]))
    h_pool_flat = tf.reshape(h_pool, [-1, 12*12*30])
    h_fc1 = tf.nn.relu(tf.matmul(h_pool_flat, W_fc1) + b_fc1)

# Output layer
with tf.name_scope('Output'):
    W_fc2 = tf.Variable(tf.truncated_normal([500, 10], stddev=0.1))
    b_fc2 = tf.Variable(tf.constant(0.1, shape=[10]))
    y_hat=tf.nn.softmax(tf.matmul(h_fc1, W_fc2) + b_fc2)

# Train and Evaluate the Model
with tf.name_scope('CrossEntropy'):
    cross_entropy = - tf.reduce_sum(y_*tf.log(y_hat))
    tf.scalar_summary("cost", cross_entropy)

with tf.name_scope('Training'):
    train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)

with tf.name_scope('Accuracy'):
    correct_prediction = tf.equal(tf.argmax(y_hat,1), tf.argmax(y_,1))
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    tf.scalar_summary("accuracy", accuracy)

summary_op = tf.merge_all_summaries()
```

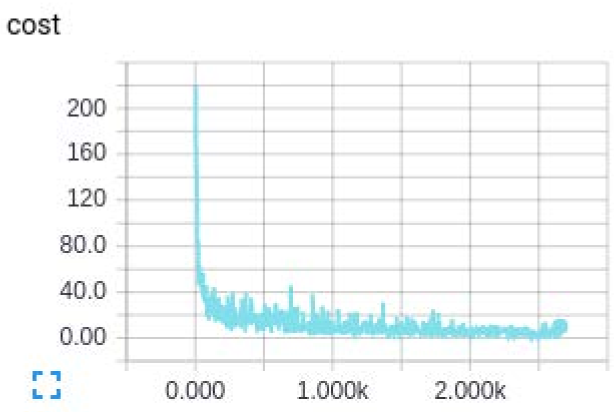
Listing 8. project1_mnist_tboard.py

```
with tf.Session() as sess:
    sess.run(tf.initialize_all_variables())
    writer = tf.train.SummaryWriter("proj1", graph=tf.get_default_graph())
    print("================================")
    print("|Epoch\tBatch\t|Train\t|Val\t|")
    print("|==============================|")
    avg_cost=0
    batch_count = int(mnist.train.num_examples/100)
    for j in range(5):
        for i in range(batch_count):
            batch = mnist.train.next_batch(100)
            _, summary = sess.run([train_step, summary_op], feed_dict={x: batch[0], y_:
batch[1]})
            writer.add_summary(summary, j * batch_count + i)
            if i%50 == 49:
                train_accuracy = accuracy.eval(feed_dict={x:batch[0], y_: batch[1]})
                val_accuracy = accuracy.eval(feed_dict=\
                    {x: mnist.validation.images, y_:mnist.validation.labels})
                print("|%d\t|%d\t|%.4f\t|%.4f\t|"%(j+1, i+1, train_accuracy, val_accuracy))
    print("|==============================|")
    test_accuracy = accuracy.eval(feed_dict=\
        {x: mnist.test.images, y_:mnist.test.labels})
    print("test accuracy=%.4f"%(test_accuracy))
```

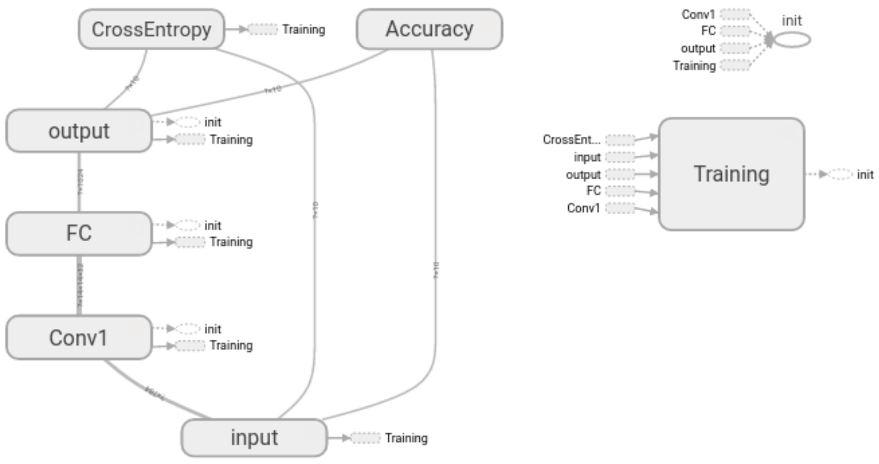
Listing 8. (Cont.) project1_mnist_tboard.py

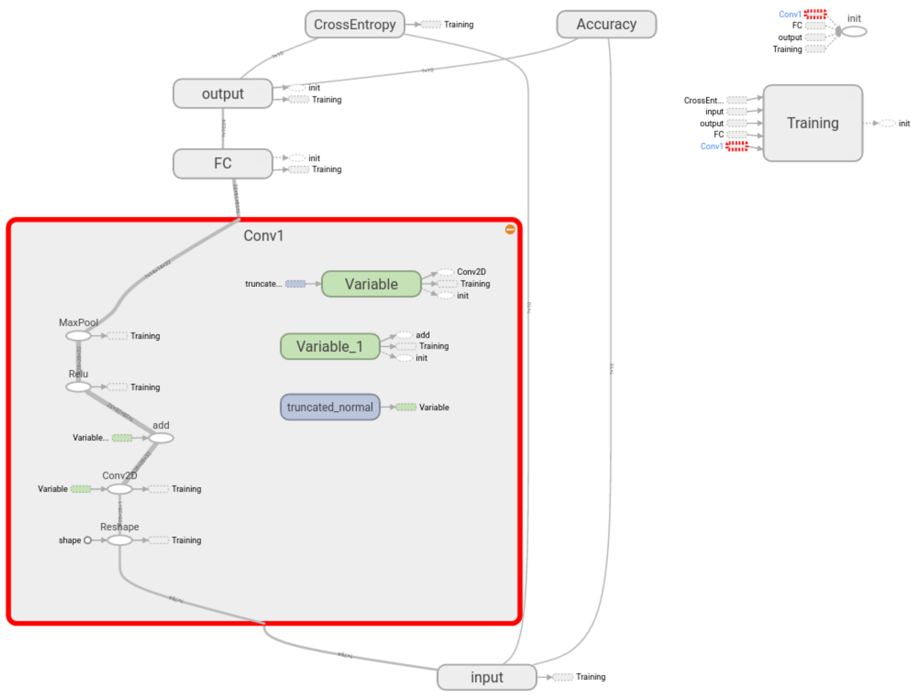
- For more information, please refer to TensorFlow API page at https://www.tensorflow.org/versions/r0.11/api_docs/index.html
- When you run the above python program, it will generate a log file in "./proj1" directory. To launch tensorboard, run "python /usr/local/lib/python2.7/dist-packages/tensorflow/tensorboard/tensorboard.py --logdir=proj1" and then launch a web browser with address http://127.0.1.1:6006
- If you use one of 24 servers, you can launch a web browser in command line by typing "chromium-browser".
- Then, it will show graphs as shown in the next page.


accuracy
1.00
0.800
0.600
0.400
0.200
0.000
1.000k
2.000k


cost
200
160
120
80.0
40.0
0.00
0.000
1.000k
2.000k

Main Graph


CrossEntropy
Training
Accuracy
output
init
Training
FC
init
Training
Conv1
init
Training
input
Training
Conv1
FC
output
Training
init
CrossEnt...
input
output
FC
Conv1
Training
init


CrossEntropy
Training
Accuracy
output
init
Training
FC
init
Training
Conv1
truncate...
Variable
Conv2D
Training
init
Variable_1
add
Training
init
truncated_normal
Variable
MaxPool
Training
Relu
Training
Variable...
add
Variable
Conv2D
Training
shape
Reshape
Training
input
Training
Conv1
FC
output
Training
init
CrossEnt...
input
output
FC
Conv1
Training
init