Project #1

EE488B Special Topics in EE <Deep Learning & AlphaGo>
2017 Fall, School of EE, KAIST

- Contents

- Using TensorFlow
  - Read Appendices A & B for using computing resources provided for this class.
  - Read Appendix C if you want to install TensorFlow on your personal machine – Windows, Mac OS X or Ubuntu.
  - Read “Get Started” and “Tutorials” at https://www.tensorflow.org before working on this project.

- Source codes
  - Download project1.zip from KLMS that contains all the source codes.

- Exercise 1: Learning XOR using TensorFlow
  - This is an exercise. You don’t need to submit anything.
  - 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 6, which has one hidden layer with two neurons.
  - The following example code (xor.py) trains the neural network to learn to perform XOR. To demonstrate convergence to the global minimum shown in page 12 in Lecture note #6, 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. xor.py

- In the following, we explain the code.
- Parameter specification
  - This part defines the hyper parameters 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 #6). 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 12 of Lecture note #6. 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 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]
```

- MNIST handwritten digit recognition using TensorFlow (mnist_*.py)
  - This is an exercise. You don’t need to submit anything.
  - MNIST is a 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. When you run the program for the first time, it will download the MNIST dataset and will take some more time to run.

```
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. 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 (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. mnist_conv1.py

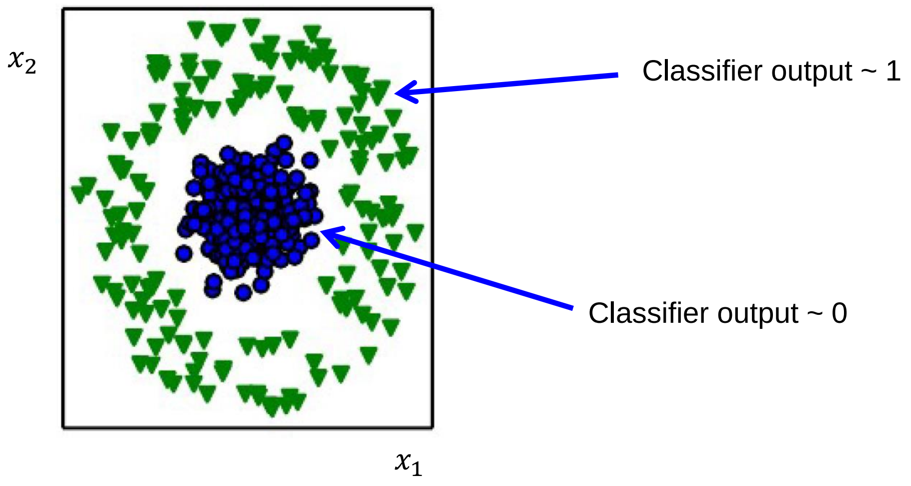
- In the following, we explain the code mnist_conv1.py.
  - Convolutional layer
    - ‘x_image = tf.reshape(x, [-1,28,28,1])
      First we should reshape the input image x to (batch_size)x28x28x1. -1 means indefinite, which will be automatically calculated to match the total size, e.g., it becomes equal 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 mnist_conv1.py to see how much improvement you get in accuracy over 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 (mnist_conv2.py): The above code (mnist_conv1.py) has only one convolutional layer. In 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 mnist_conv2.py. Although it has one more convolutional layer than mnist_conv1.py, its accuracy will be similar. This is because mnist_conv1.py already achieves good enough performance.
- Dropout (mnist_dropout.py): Finally, we show how to add a dropout layer. Using dropout can reduce overfitting. This operation randomly drops units from the neural network during training. In the code 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 mnist_dropout.py. The number of epochs is set to 20 in mnist_dropout.py for better performance and the test accuracy will be about 99%.

- Task #1 (Classification – 10 points)
  - Write a TensorFlow program that can perform the following classification.



  - Use the following network architecture
    - Number of layers: 2 (both are fully connected layers)
    - Number of inputs: 2 (x_1 and x_2)
    - Number of neurons in the hidden layer: 20
    - Activation function for hidden layer: ReLU
    - Number of outputs: 1
    - Activation function for the output layer: sigmoid
  - Training, validation, and test datasets should be generated as follows:
    - For label "0": $(x_1, x_2) = (r\cos t\,, r\sin t)$
    - For label "1": $(x_1, x_2) = ((r+5)\cos t\,, (r+5)\sin t)$
    - where r is Gaussian with mean 0 and variance 1 and t is uniform in $[0, 2\pi)$. r and t are independent.
  - Pick hyper-parameters, training set size, optimization method, etc. so that your accuracy is as high as possible.
  - Design your neural network as specified above. Do not include anything else. For example, do ***not*** perform explicit coordinate transform from Cartesian to polar.
  - Print out training and validation accuracies during training.
  - Plot your test data in 2 dimensions (similarly as shown above). In the plot, also plot lines dividing active and inactive regions for hidden-layer neurons, i.e., plot (x_1, x_2) values as a line for each hidden-layer neuron such that the input to ReLU is 0.
  - Can you figure out how your neural network is performing classification task by looking at the lines?
  - In your report (hard copy that you bring to class on the due date), include the following:
    - Source code and plot
    - Output of your code that shows training and validation accuracies during training, final test accuracy, and trained parameters of your neural network
    - Explanations on how you designed your code
    - Interpretations of your results

- Task #1 (continued)
  - In addition to submitting hard copy of your report, submit your source electronically as follows:
    - Place your source code in your home directory of your computer in N5. Its name should be “project1_task1.py”.
    - For example, if your machine’s IP address is 143.248.141.85 and your id is s25, then your home directory should contain “project1_task1.py” as shown below.

```
s25@EE2354-5:~$ ls -l
total 20
-rw-r--r-- 1 s25 s25 8980  9월 29 19:30 examples.desktop
-rw-rw-r-- 1 s25 s25   25 10월 26 16:04 project1_task1.py
s25@EE2354-5:~$
```

    - Even if you use your own computer, you should copy your file to your home directory of your machine in N5. Refer to Appendix A 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 incorrect, it will not be collected.

- Task #2 (Transfer learning – 10 points)
  - Modify mnist_conv1.py so that it can learn to classify 9 digits instead of 10.
  - Choose any digit from 0 to 9. In the MNIST dataset, remove all data containing the digit. Train your neural network using this modified dataset. Your neural network should have 9 outputs instead of 10. Make sure the classification accuracy is higher than 98%.
  - Now, let's use transfer learning to see if your network can be trained to classify a new digit, i.e., the one that you excluded above. Add one more output neuron in your neural network. The new output neuron should be connected to the last hidden layer.
  - Train only the final layer of the new network using the original MNIST data with all 10 digits.
  - In your report (hard copy that you bring to class on the due date), include the following:
    - Source code
    - Output of your code that shows training and validation accuracies during training, and final test accuracy
    - Explanations on how you designed your code
    - Interpretations of your results
  - In addition to submitting hard copy of your report, submit your source electronically as follows:
    - Place your source code in your home directory of your computer in N5. Its name should be "project1_task2.py".

- Task #3 (PCA – 5 points)
  - Modify mnist_conv1.py so that after training, PCA is applied to the last hidden layer's output. Use the test dataset to obtain principle components, i.e., do the following:
    - For each test example, get the 500-dimensional output of the last hidden layer.
    - Collect all 10,000 such vectors using all 10,000 test examples and form a 10,000 by 500 matrix. Let Ψ denote the matrix.
    - For each column of Ψ, subtract the mean. Let Φ denote the resulting matrix.
    - Apply SVD to $\Phi^T\Phi$, i.e., obtain $\Phi^T\Phi = W\Sigma^2W^T$, where W is a 500 by 500 orthogonal matrix and $\Sigma$is a 500 by 500 diagonal matrix containing 500 singular values of Φ in descending order. You can use np.linalg.svd to perform SVD.
  - Calculate $Z = \Phi W$ and use 2D scatter plot to plot the first two columns of Z, i.e., plot each row of Z as a dot in a 2D plane. There should be a total of 10,000 dots.
  - Pick 10 examples for each digit in the test set and construct a new dataset with 100 examples. Make sure it is ordered by label, i.e., first 10 examples in the data set are "0" and the next 10 examples are "1" and so on. For each example, get the 500-dimensional output of the last hidden layer. Collect all 100 such vectors and form a 100 by 500 matrix Ω. Calculate $Q = \Omega W$, where W is the 500 by 500 orthogonal matrix obtained above (i.e., from $\Phi^T\Phi = W\Sigma^2W^T$) and use 2D scatter plot to plot the first two columns of Q, i.e., plot each row of Q as a colored dot in a 2D plane. Use different colors for different labels. There should be a total of 100 dots, one for each example.
  - In your report (hard copy that you bring to class on the due date), include the following:
    - Source code and plots
    - Explanations on how you designed your code
    - Interpretations of your results
  - In addition to submitting hard copy of your report, submit your source electronically as follows:
    - Place your source code in your home directory of your computer in N5. Its name should be "project1_task3.py".

Appendix A

Computing Resources

- Computers in Haedong lounge in E3-4
  - Refer to https://ee.kaist.ac.kr/node/15074 and “Haedong lounge computers usage guideline.pdf” (download from KLMS) for instructions and help.
  - Total # of computers in Haedong lounge is 35.
  - Each computer as two GTX 1070 GPU’s.
  - Username is your student id and the initial password is 2017F_EE488_your_student_id
  - Python version is 2.7.12 and TensorFlow version is 1.0.1.

- Computers in LG hall
  - You can only use these remotely. Use the same username/password as above.
  - Remote login instructions are given in Appendix B.
  - There are 14 computers and their addresses for remote login are eelabg1.kaist.ac.kr ~ eelabg14.kaist.ac.kr.
  - Python version is 2.7.13 and TensorFlow version is 1.2.1.
  - Each computer in LG hall has four GTX 1070 GPU’s. Never use more than one GPU since each computer can be shared by many students simultaneously.
  - To specify which GPU to use, type the following at linux command, where 2 means GPU 2. It can be either 0, 1, 2 or 3.

```
$ CUDA_VISIBLE_DEVICES=2 python your_program.py
```

  - ***If you simply run python as below without defining CUDA_VISIBLE_DEVICES as above, it will use all the GPU’s available in the computer. If it is discovered that you are using more than one GPU at a time for computers in LG hall, then you will get penalty.***

```
$ python your_program.py
```

- Computers in N5
  - You can only use these remotely.
  - There are 24 computers with IP addresses from 143.248.141.81 to 143.248.141.104
  - Each computer in N5 has one GTX 960 GPU.
  - Python version is 2.7.12 and TensorFlow version is 0.11.
  - Your id is “s35” if your clicker id is 35. Your initial password is “ee488_2017”.
  - For computers in N5, you need to use the computer that has your id. Use the computer with IP address 143.248.141.x, where x is 81 + mod(your_clicker_id, 20).

- ***Use only one computer at a time among all computers in Haedong lounge and in LG hall. Otherwise, you will get penalty.***
  - This does not apply to computers in N5.

- ***For computers in LG hall, if you use more than one GPU, then you will get penalty.***

- When using computers in E3-4 or in LG hall, make sure of the following:
  - ***Insert*** the following two lines at the end of the file “~./bashrc” if the file does not contain the following already.

```
export PATH=/usr/local/cuda/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda/lib64:/usr/local/cuda/extras/CUPTI/lib64
```

  - ***Remove*** the following two lines in the file “~./bashrc” if the file contains the following.

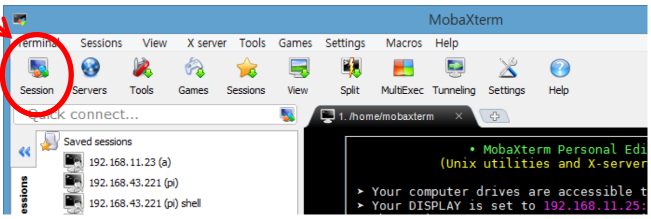
```
# added by Anaconda2 installer
export PATH="/opt/anaconda2/bin:$PATH"
```

  - If you never logged in before October 27th, then you probably don’t need to do above.

Appendix B

Remote Login

- To access a linux machine remotely, first download and install on your computer a free X terminal program called MobaXTerm from http://mobaxterm.mobatek.net/.
  - If you use MAC OS, then you can simply open “Terminal” and type “ssh –X username@server_address”
- Click on “Session” icon to define a new session.

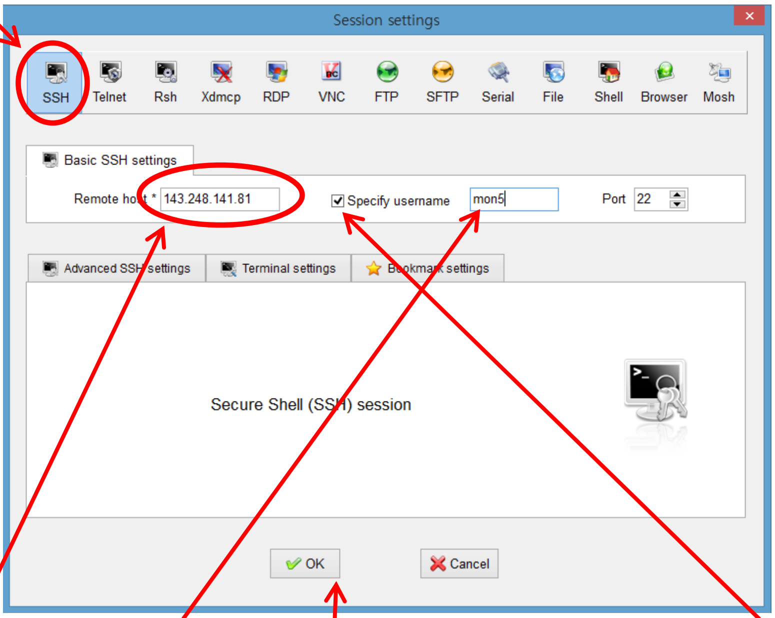

MobaXterm
Terminal Sessions View X server Tools Games Settings Macros Help
Session Servers Tools Games Sessions View Split MultiExec Tunneling Settings Help
Quick connect...
1. /home/mobaxterm
Sessions

Saved sessions
192.168.11.23 (a)
192.168.43.221 (pi)
192.168.43.221 (pi) shell

```
• MobaXterm Personal Edi
(Unix utilities and X-server

➤ Your computer drives are accessible t
➤ Your DISPLAY is set to 192.168.11.25:
```

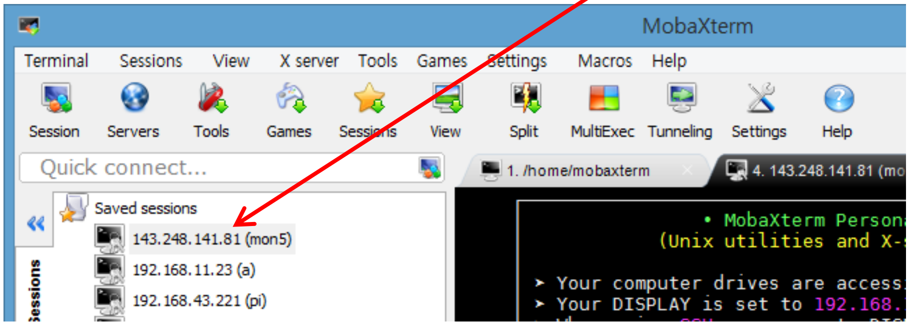
- Click on “SSH” icon.



Remote host * 143.248.141.81 ☑ Specify username mon5 Port 22

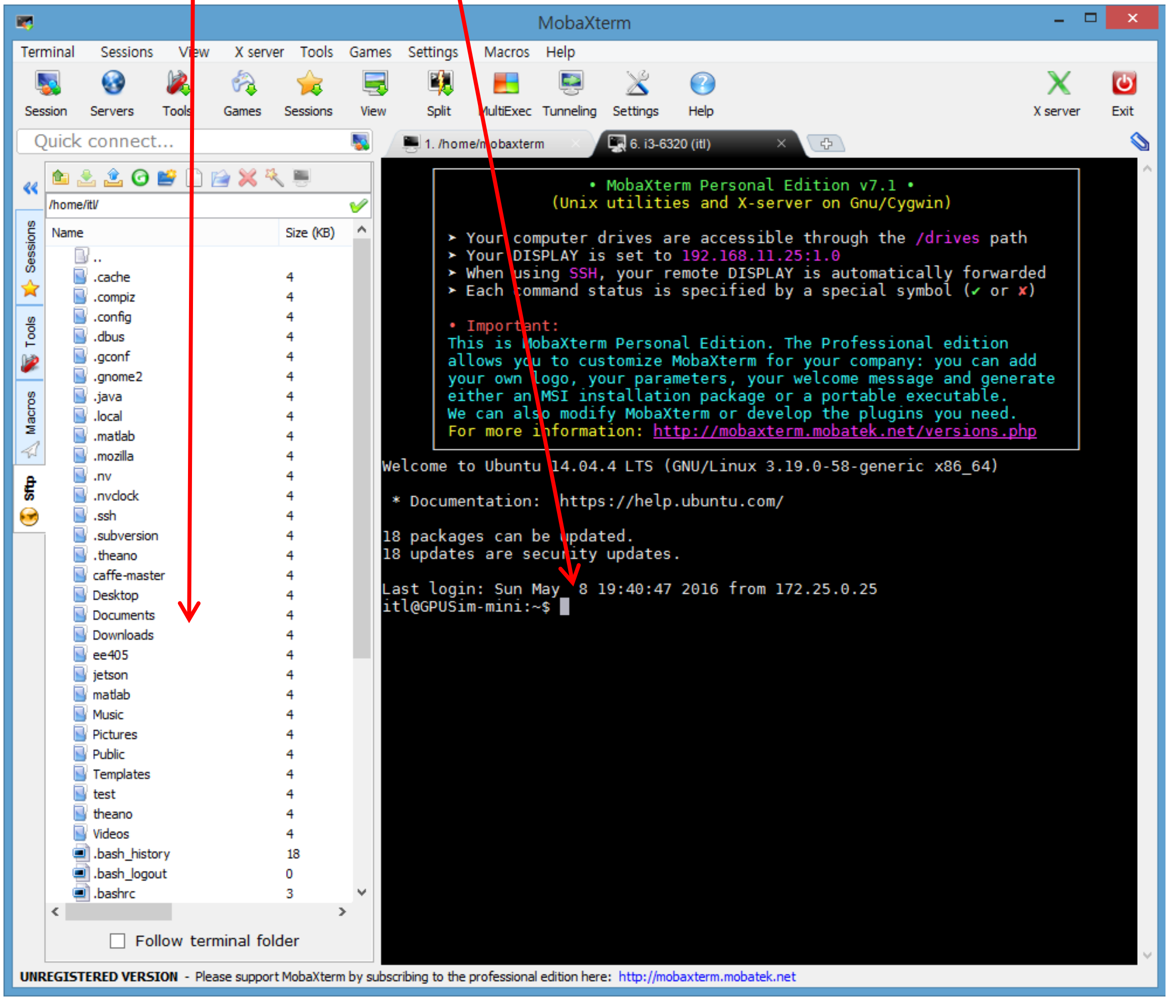
- Type the IP address of the computer you are trying to access here and check this box.
- Enter your username here and click “OK”.

- If it is the first time, it will ask your password and then the session will start. It will remember your password, so you don't need to type the password next time.
- Next time you launch MobaXTerm, you can double click a session to start the session.


MobaXterm
Terminal Sessions View X server Tools Games Settings Macros Help
Session Servers Tools Games Sessions View Split MultiExec Tunneling Settings Help
Quick connect...
1. /home/mobaxterm
4. 143.248.141.81 (mo
Saved sessions
143.248.141.81 (mon5)
192.168.11.23 (a)
192.168.43.221 (pi)
Sessions

```
• MobaXterm Person
(Unix utilities and X-
➤ Your computer drives are access
➤ Your DISPLAY is set to 192.168.
```

- You can type linux commands here.
- You can copy files between your computer and the remote linux computer by dragging files to/from this Sftp window.


MobaXterm
Terminal Sessions View X server Tools Games Settings Macros Help
Session Servers Tools Games Sessions View Split MultiExec Tunneling Settings Help X server Exit
Quick connect...
1. /home/mobaxterm
6. i3-6320 (itl)
Sessions
Tools
Macros
Sftp
Follow terminal folder
UNREGISTERED VERSION - Please support MobaXterm by subscribing to the professional edition here: http://mobaxterm.mobatek.net

/home/itl/

Name	Size (KB)
..	
.cache	4
.compiz	4
.config	4
.dbus	4
.gconf	4
.gnome2	4
.java	4
.local	4
.matlab	4
.mozilla	4
.nv	4
.nvclock	4
.ssh	4
.subversion	4
.theano	4
caffe-master	4
Desktop	4
Documents	4
Downloads	4
ee405	4
jetson	4
matlab	4
Music	4
Pictures	4
Public	4
Templates	4
test	4
theano	4
Videos	4
.bash_history	18
.bash_logout	0
.bashrc	3

```
                • MobaXterm Personal Edition v7.1 •
              (Unix utilities and X-server on Gnu/Cygwin)

 ➤ Your computer drives are accessible through the /drives path
 ➤ Your DISPLAY is set to 192.168.11.25:1.0
 ➤ When using SSH, your remote DISPLAY is automatically forwarded
 ➤ Each command status is specified by a special symbol (✓ or ✗)

 • Important:
 This is MobaXterm Personal Edition. The Professional edition
 allows you to customize MobaXterm for your company: you can add
 your own logo, your parameters, your welcome message and generate
 either an MSI installation package or a portable executable.
 We can also modify MobaXterm or develop the plugins you need.
 For more information: http://mobaxterm.mobatek.net/versions.php

Welcome to Ubuntu 14.04.4 LTS (GNU/Linux 3.19.0-58-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

18 packages can be updated.
18 updates are security updates.

Last login: Sun May  8 19:40:47 2016 from 172.25.0.25
itl@GPUSim-mini:~$
```

- If you are outside KAIST campus, you can remotely access the computers on campus using VPN. To use VPN, visit http://kvpn.kaist.ac.kr and follow instructions (VPN manual).
  - First, you need to setup id/password for KVPN (KAIST VPN) by clicking on “Join KAIST VPN”.
  - Then, you visit http://kvpn.kaist.ac.kr and login. NC Client will be downloaded and installed.
  - Now, you can access KAIST network from outside.
  - Run MobaXTerm to access computers on campus.

Appendix C

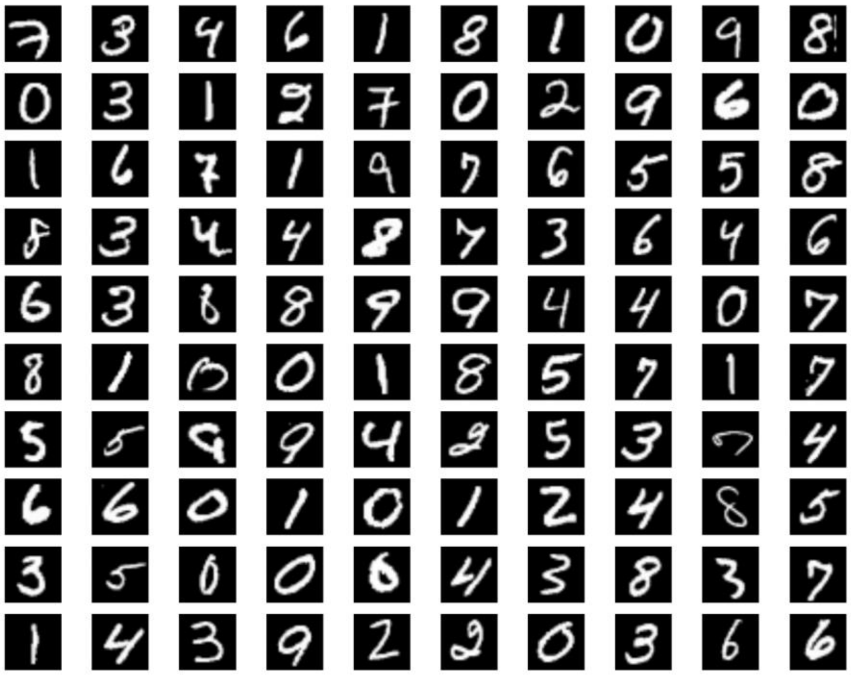
Installing TensorFlow

- Installing TensorFlow on your personal machine
  - ***You don’t need to read this if you use a server computer for this class.*** You only need to read this if you want to install TensorFlow on your personal machine. You can install TensorFlow on Windows, MAC OS X, and Ubuntu.
  - Basic installation instructions can be found at https://www.tensorflow.org/install/. We just give some tips and cautions here.
  - About CUDA 8.0
    - If your computer has an NVIDIA GPU whose CUDA Compute Capability is 3.0 or higher, then you can have a significant speedup, but you need to have CUDA toolkit installed on your computer.
    - Uninstall CUDA Toolkit on your computer if you already have one installed whose version is not 8.0.
    - You can download and install CUDA 8.0 from https://developer.nvidia.com/accelerated-computing-toolkit
    - Since CUDA 9.0 is downloaded by default, you need to click “CUDA Toolkit Download”, then click “Legacy Releases” under “Additional Resources”, then choose “CUDA Toolkit 8.0 GA2 [Feb 2017]” to download CUDA 8.0.
    - CUDA 9.0 is the latest version as of now, but you need to use 8.0 for the current version of TensorFlow, which may change in the future.
  - About cuDNN v6 or v6.1
    - You also need this in addition to CUDA toolkit for speedup when using GPU.
    - The current version as of now is v7, but you need to use v6 or v6.1 for the current version of TensorFlow (this may change in the future)
    - You can download cuDNN v6 or v6.1 from https://developer.nvidia.com/cudnn and unzip the *.zip file to any directory (preferably CUDA directory, e.g., “C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA” if you are using Windows).
    - Set the path variable to include the “bin” directory of cuDNN.
  - Download and install python 3.6.* from https://www.python.org
    - Choose 64-bit version if your OS is 64 bits.
    - You can use Anaconda instead. Read instructions at https://www.tensorflow.org/install/.
  - (For Windows) At a command prompt, type “pip3 install –upgrade tensorflow-gpu”
  - Start python and type “import tensorflow as tf” to see if TensorFlow is correctly installed.

Appendix D

Useful Tips

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



- Running a python program in Ubuntu
  - You can run a python script by using “python mnist_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(‘mnist_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('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)
```