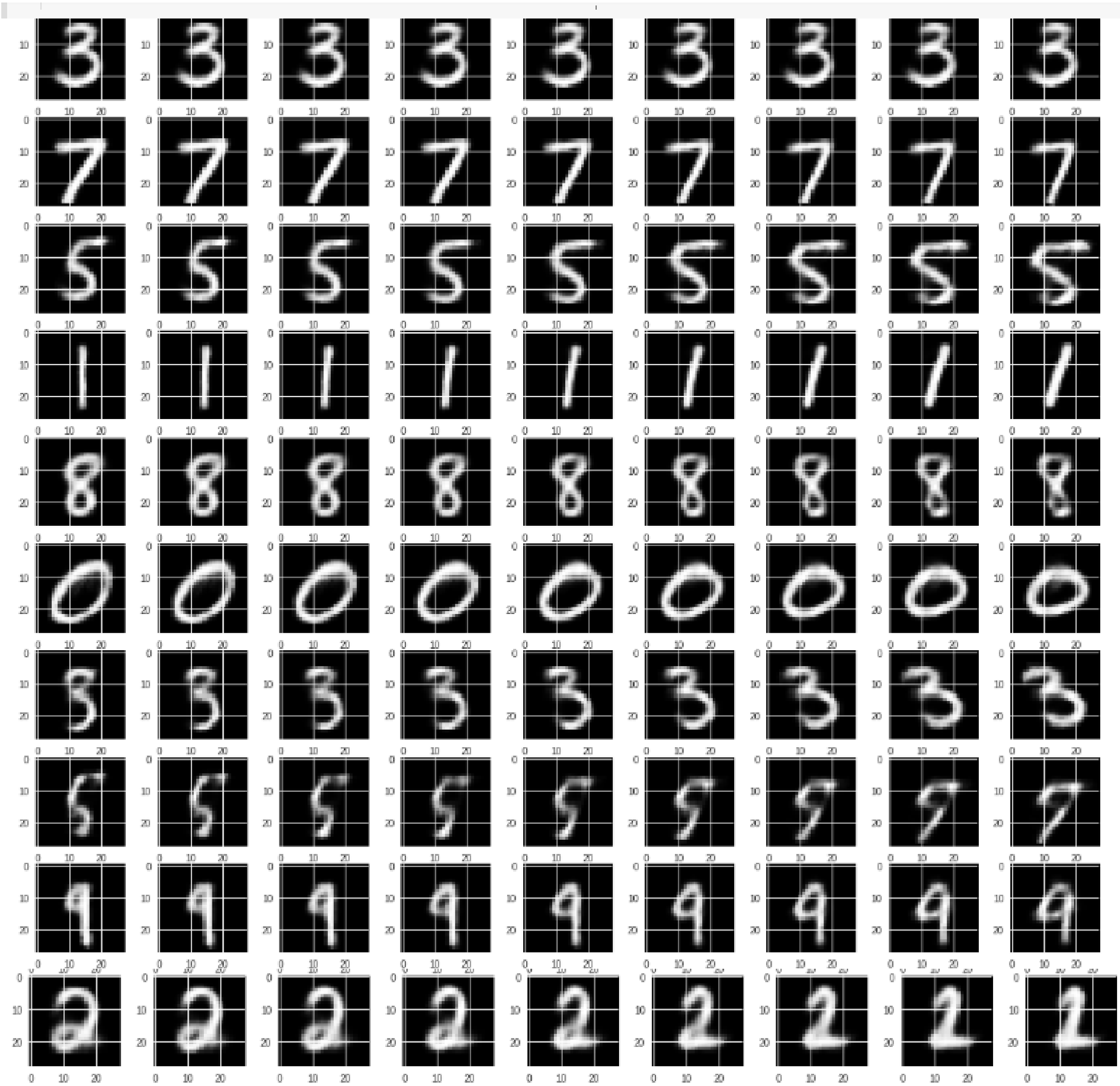
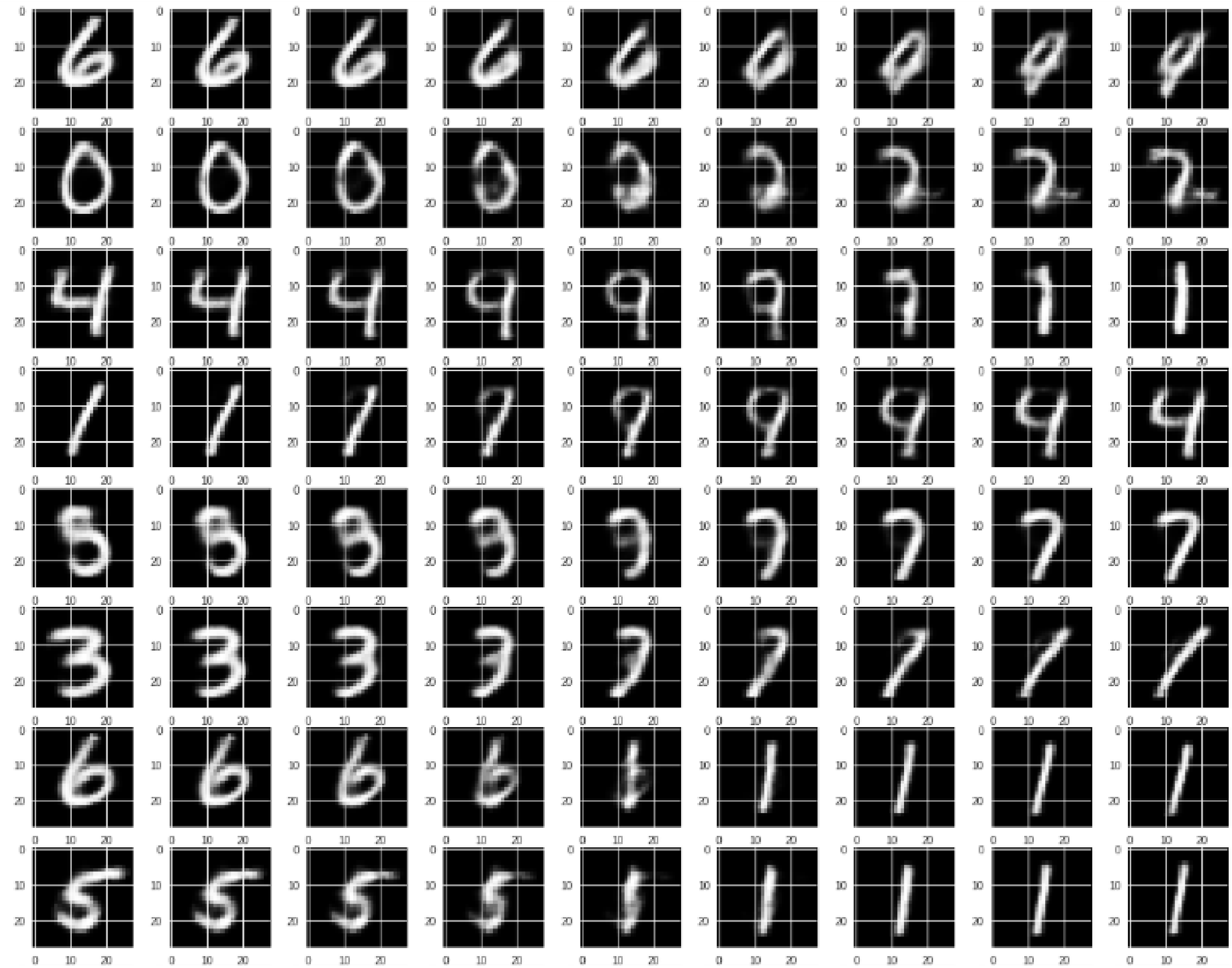
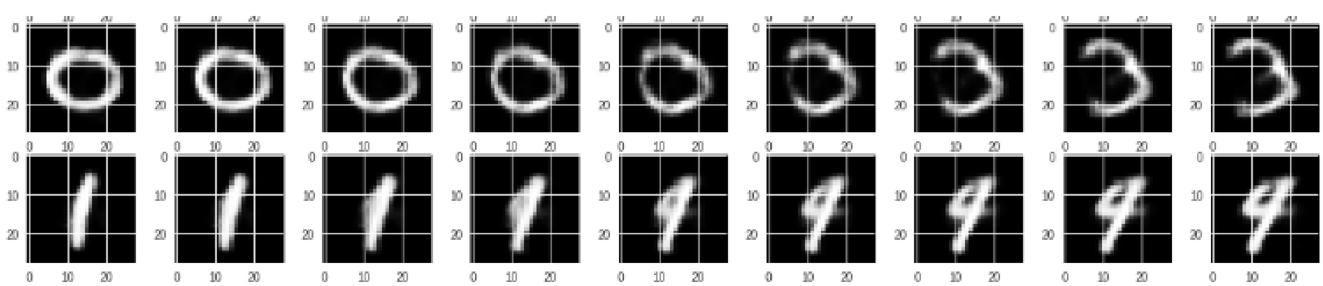
1. Page 1: Part 1. Show grid of 10 x 9 for a single MNIST digit- decoded images.



Page 2: Part 2. Show a grid of 10 x 9 for different digits - decoded images.





Page 3: Important lines of your code:

1. Creating Dictionary for storing indexes

```
#Creating the Dictionary for storing the indexes of labels
 x=mnist.test.images
 y=mnist.test.labels

 dic={"0":[],"1":[],"2":[],"3":[],"4":[],"5":[],"6":[],"7":[],"8":[],"9":[]}
 x=mnist.test.images

 y=mnist.test.labels
 for i in range(len(y)):
   if y[i]==0:
       dic["0"].append(i)

   if y[i]==1:
       dic["1"].append(i)

   if y[i]==2:
       dic["2"].append(i)

   if y[i]==3:
       dic["3"].append(i)

   if y[i]==4:
       dic["4"].append(i)

   if y[i]==5:
       dic["5"].append(i)

   if y[i]==6:
       dic["6"].append(i)

   if y[i]==7:
       dic["7"].append(i)

   if y[i]==8:
       dic["8"].append(i)

   if y[i]==9:
       dic["9"].append(i)
```

2. Plot and Interpolate the vectors (Part-1 Choosing same Images randomly)

```
fig_size = plt.rcParams["figure.figsize"]


# Set figure width to 20 and height to 20
fig_size[0] = 20
fig_size[1] = 20
plt.rcParams["figure.figsize"] = fig_size

for h in range(1,91,9):

    l=[]
    #Random sample the digit
    ran=random.sample([0,1,2,3,4,5,6,7,8,9],1)

    #Now sample the above digits randomly from the dictionary

    m=list(dic[str(ran[0])])
    num=random.sample(m,2)

    #Append the randomly selected digits to the list
    l.append(x[num[0],:])
    l.append(x[num[1],:])
    print(y[num[0]])
    print(y[num[1]])

    #ouput from the encoder of the selected images
    a=sess.run(z,feed_dict={input_image:l})




    #Plot the first digit (output of the decoder)
    f=sess.run(decoder,feed_dict={noise_input: a[0].reshape(-1,7)})
    plt.subplot(10, 9, h)

    plt.imshow(f.reshape(28,28),cmap='gray')



    w=h
    #creating the 7 interpolating vectors ,then passing it ot the decoder for output

    for s in range(1,8):
            q=a[0,:]+ ((s/8)*(a[1,:]-a[0,:]))
            f=sess.run(decoder,feed_dict={noise_input: q.reshape(-1,7)})
            w=w+1
            plt.subplot(10, 9,w)

            plt.imshow(f.reshape(28,28),cmap='gray')


    #Plot the last image
    f=sess.run(decoder,feed_dict={noise_input: a[1].reshape(-1,7)})


    plt.subplot(10, 9,w+1)
    plt.imshow(f.reshape(28,28),cmap='gray')
```

4. Plot and Interpolate the vectors (Part-2 Choosing different Images randomly)

```
fig_size = plt.rcParams["figure.figsize"]

# Set figure width to 20 and height to 20
fig_size[0] = 20
fig_size[1] = 20
plt.rcParams["figure.figsize"] = fig_size

for h in range(1,91,9):
    #d=d+1
    #print(d)
    #for k in d:

    #Randomly sample the two different digits
    l=[]
    ran=random.sample([0,1,2,3,4,5,6,7,8,9],2)
          #print("vv",k)
    m=list(dic[str(ran[0])])
    num1=random.sample(m,1)
          #print(num)
    m=list(dic[str(ran[1])])
    num2=random.sample(m,1)
    print(y[num1[0]])
    print(y[num2[0]])
    #Appending the pixels of the two randomly selected digits
    l.append(x[num1])
    #print(np.array(l).shape)
    l.append(x[num2])

    #print(np.array(l).shape)

    #Creating the encoded images
    a=sess.run(z,feed_dict={input_image:(np.array(l).reshape(-1,784))})



    # Plotting the first decoded image
    f=sess.run(decoder,feed_dict={noise_input: a[0].reshape(-1,7)})
    plt.subplot(10, 9, h)

    plt.imshow(f.reshape(28,28),cmap='gray')



    w=h
    #Creating 7 evenly separated interpolates and plotting the decoded image of each
    for s in range(1,8):
            q=a[0,:]+ ((s/8)*(a[1,:]-a[0,:]))
            f=sess.run(decoder,feed_dict={noise_input: q.reshape(-1,7)})
            w=w+1
            plt.subplot(10, 9,w)

            plt.imshow(f.reshape(28,28),cmap='gray')


    #Plotting the final decoded image
    f=sess.run(decoder,feed_dict={noise_input: a[1].reshape(-1,7)})


    plt.subplot(10, 9,w+1)
    plt.imshow(f.reshape(28,28),cmap='gray')
```

Page 4 - onwards: Your entire code

Variational Autoenoder code:

Reference: https://github.com/aymericdamien/TensorFlow-Examples/blob/master/examples/3_NeuralNetworks/variational_autoencoder.py

```python
from __future__ import division, print_function, absolute_import

import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import norm
import tensorflow as tf
import random

# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/")
sess = tf.InteractiveSession()
# Parameters
learning_rate = 0.001
num_steps = 30000
batch_size = 64

# Network Parameters
image_dim = 784 # MNIST images are 28x28 pixels
hidden_dim = 512
latent_dim = 7

# A custom initialization (see Xavier Glorot init)
def glorot_init(shape):
    return tf.random_normal(shape=shape, stddev=1. / tf.sqrt(shape[0] / 2.))

# Variables
weights = {
    'encoder_h1': tf.Variable(glorot_init([image_dim, hidden_dim])),
    'z_mean': tf.Variable(glorot_init([hidden_dim, latent_dim])),
    'z_std': tf.Variable(glorot_init([hidden_dim, latent_dim])),
    'decoder_h1': tf.Variable(glorot_init([latent_dim, hidden_dim])),
    'decoder_out': tf.Variable(glorot_init([hidden_dim, image_dim]))
}
biases = {
    'encoder_b1': tf.Variable(glorot_init([hidden_dim])),
    'z_mean': tf.Variable(glorot_init([latent_dim])),
    'z_std': tf.Variable(glorot_init([latent_dim])),
    'decoder_b1': tf.Variable(glorot_init([hidden_dim])),
    'decoder_out': tf.Variable(glorot_init([image_dim]))
}

# Building the encoder
input_image = tf.placeholder(tf.float32, shape=[None, image_dim])
encoder = tf.matmul(input_image, weights['encoder_h1']) + biases['encoder_b1']
encoder = tf.nn.tanh(encoder)
z_mean = tf.matmul(encoder, weights['z_mean']) + biases['z_mean']
#print(z_mean.shape)
z_std = tf.matmul(encoder, weights['z_std']) + biases['z_std']

# Sampler: Normal (gaussian) random distribution
eps = tf.random_normal(tf.shape(z_std), dtype=tf.float32, mean=0., stddev=1.0,
                       name='epsilon')
z = z_mean + tf.exp(z_std / 2) * eps
```

```
                         name='epsilon')
z = z_mean + tf.exp(z_std / 2) * eps

# Building the decoder (with scope to re-use these layers later)
decoder = tf.matmul(z, weights['decoder_h1']) + biases['decoder_b1']
decoder = tf.nn.tanh(decoder)
decoder = tf.matmul(decoder, weights['decoder_out']) + biases['decoder_out']
decoder = tf.nn.sigmoid(decoder)


# Define VAE Loss
def vae_loss(x_reconstructed, x_true):
    # Reconstruction loss
    encode_decode_loss = x_true * tf.log(1e-10 + x_reconstructed) \
                         + (1 - x_true) * tf.log(1e-10 + 1 - x_reconstructed)
    encode_decode_loss = -tf.reduce_sum(encode_decode_loss, 1)
    # KL Divergence loss
    kl_div_loss = 1 + z_std - tf.square(z_mean) - tf.exp(z_std)
    kl_div_loss = -0.5 * tf.reduce_sum(kl_div_loss, 1)
    return tf.reduce_mean(encode_decode_loss + kl_div_loss)

loss_op = vae_loss(decoder, input_image)
optimizer = tf.train.RMSPropOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()

# Start training
with tf.Session() as sess:

    # Run the initializer
        sess.run(init)

        for i in range(1, num_steps+1):
        # Prepare Data
        # Get the next batch of MNIST data (only images are needed, not labels)
              batch_x, _ = mnist.train.next_batch(batch_size)

        # Train
              feed_dict = {input_image: batch_x}
              _, l = sess.run([train_op, loss_op], feed_dict=feed_dict)
              if i % 1000 == 0 or i == 1:
                  print('Step %i, Loss: %f' % (i, l))
```