hw2_p1

February 16, 2018

```
In [1]: import numpy as np
        import tensorflow as tf
        %matplotlib inline
        import matplotlib
        import matplotlib.pyplot as plt
```

```
In [2]: from tensorflow.examples.tutorials.mnist import input_data
        mnist = input_data.read_data_sets("MNIST_data/", one_hot=False)
        print(mnist.train.images.shape, mnist.train.labels.shape)
```

```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
(55000, 784) (55000,)
```

```
In [3]: def showimage(image, label):
            plt.gray()
            plt.imshow(image.reshape(28, 28))
            plt.show()
            print(label)

        def extract_target_data(X, Y, target, num):
            p = (Y == target)
            x_target = X[p,:]
            y_target = Y[p]
            y_target = np.expand_dims(y_target, axis=1)
            return x_target[:num,:], y_target[:num,:]

        # computer euclidean distance matrix
        def euclidean_distance_matrix(x):
            r = tf.reduce_sum(x*x, 1)
            r = tf.reshape(r, [-1, 1])
            distance_mat = r - 2*tf.matmul(x, tf.transpose(x)) + tf.transpose(r)
            #return tf.sqrt(distance_mat)
            return distance_mat
```

```
In [4]: train_images, train_labels = extract_target_data(mnist.train.images, mnist.train.labels,

        # get 1000 data from each category
        for i in range(1, 10):
            cur_train_images, cur_train_labels = extract_target_data(mnist.train.images, mnist.t
            train_images = np.concatenate((train_images, cur_train_images), axis=0)
            train_labels = np.concatenate((train_labels, cur_train_labels), axis=0)

        train_images /= 255.
        #print(train_images.shape, train_labels.shape)
```

```
In [5]: distance_mat = euclidean_distance_matrix(train_images)
        with tf.Session() as sess:
            M = sess.run(distance_mat)
```

```
In [6]: with tf.device('/device:GPU:0'):
            x = tf.placeholder(tf.float32, shape=[None, 784], name='x')
            w = tf.get_variable('w', shape=[784, 2], initializer=tf.contrib.layers.xavier_initia
            b = tf.get_variable('b', shape=[2], initializer=tf.zeros_initializer())

            z = tf.matmul(x, w) + b
            M_ = euclidean_distance_matrix(z)
            # MSE cost function
            cost = tf.reduce_mean(tf.square(M_ - M))
            optimizer = tf.train.AdamOptimizer().minimize(cost)
```

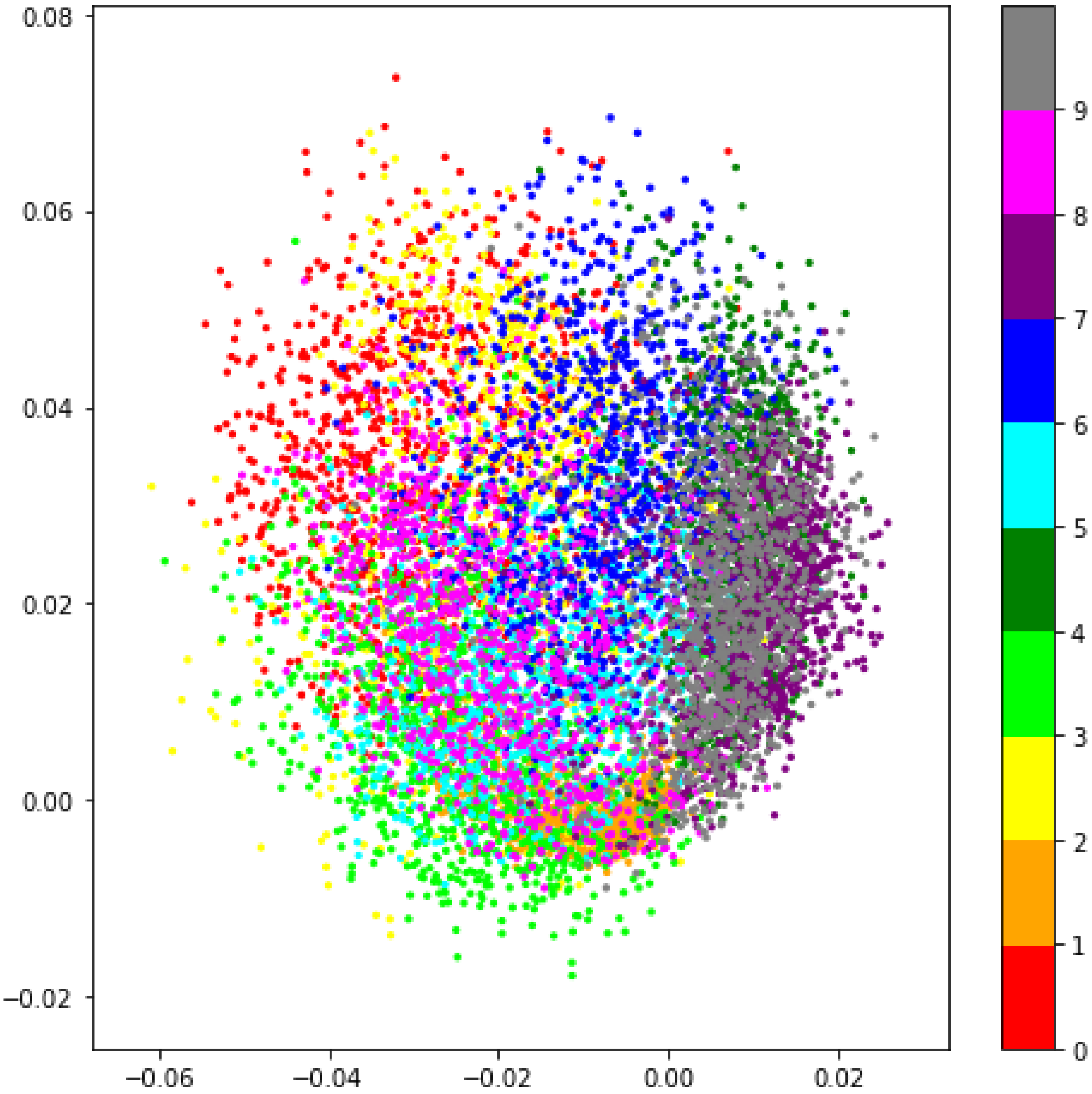
```
In [7]: epoch = 1000
        config = tf.ConfigProto()
        config.gpu_options.allow_growth = True
        with tf.Session(config=config) as sess:
            sess.run(tf.global_variables_initializer())
            for i in range(epoch):
                _, epoch_cost, transform_mat = sess.run([optimizer, cost, w], feed_dict = {x:tra
```

```
In [8]: emb_images = np.dot(train_images, transform_mat)
        print(emb_images.shape)
```

```
(10000, 2)
```

```
In [9]: x = emb_images[:,0]
        y = emb_images[:,1]
        label = train_labels
        colors = ['red','orange','yellow','lime','green','cyan','blue','purple','magenta','grey'

        fig = plt.figure(figsize=(8,8))
        plt.scatter(x, y, c=label, cmap=matplotlib.colors.ListedColormap(colors),s=5)

        cb = plt.colorbar()
```

```
loc = np.arange(0,max(label),max(label)/float(len(colors)))
cb.set_ticks(loc)
cb.set_ticklabels(list(range(10)))
```



We can see from the image above, the datapoints of same category are clustering together. This infers the embedding, even though with only 2 dimensions, perserve essential distance information from 786 dimensional matrix and are able to dissimilate from other categories.

hw2_p2

February 16, 2018

```
In [1]: import numpy as np
        import pandas as pd
        import tensorflow as tf

        %matplotlib inline
        import matplotlib
        import matplotlib.pyplot as plt

        import pymesh
        from pyntcloud import PyntCloud
```

```
In [2]: # compute triangle mesh surface area
        def triangle_area(x):
            a = x[:,0,:] - x[:,1,:]
            b = x[:,0,:] - x[:,2,:]
            cross = np.cross(a, b)
            area = 0.5 * np.linalg.norm(np.cross(a, b), axis=1)
            return area

        # compute euclidean distance matrix
        def euclidean_distance_matrix(x):
            r = np.sum(x*x, 1)
            r = r.reshape(-1, 1)
            distance_mat = r - 2*np.dot(x, x.T) + r.T
            #return np.sqrt(distance_mat)
            return distance_mat

        # update distance matrix and select the farthest point from set S after a new point is s
        def update_farthest_distance(far_mat, dist_mat, s):
            for i in range(far_mat.shape[0]):
                far_mat[i] = dist_mat[i,s] if far_mat[i] > dist_mat[i,s] else far_mat[i]
            return far_mat, np.argmax(far_mat)

        # initialize matrix to keep track of distance from set s
        def init_farthest_distance(far_mat, dist_mat, s):
            for i in range(far_mat.shape[0]):
                far_mat[i] = dist_mat[i,s]
            return far_mat
```

```
In [3]: # get sample from farthest point on every iteration
        def farthest_point_sampling(obj_file, num_samples=1000):
            mesh = pymesh.load_mesh(obj_file)
            faces = mesh.vertices[mesh.faces]
            area = triangle_area(faces)
            total_area = np.sum(area)

            set_P = []
            for i in range(faces.shape[0]):
                num_gen = area[i] / total_area * 10000
                for j in range(int(num_gen)+1):
                    r1, r2 = np.random.rand(2)
                    d = (1-np.sqrt(r1)) * faces[i,0] + np.sqrt(r1)*(1-r2) * faces[i,1] + np.sqrt
                    set_P.append(d)

            set_P = np.array(set_P)
            num_P = set_P.shape[0]

            distance_mat = euclidean_distance_matrix(set_P)

            set_S = []
            s = np.random.randint(num_P)
            far_mat = init_farthest_distance(np.zeros((num_P)), distance_mat, s)

            for i in range(num_samples):
                set_S.append(set_P[s])
                far_mat, s = update_farthest_distance(far_mat, distance_mat, s)

            return np.array(set_S)
```

```
In [4]: teapot_pts = farthest_point_sampling('teapot.obj')
```

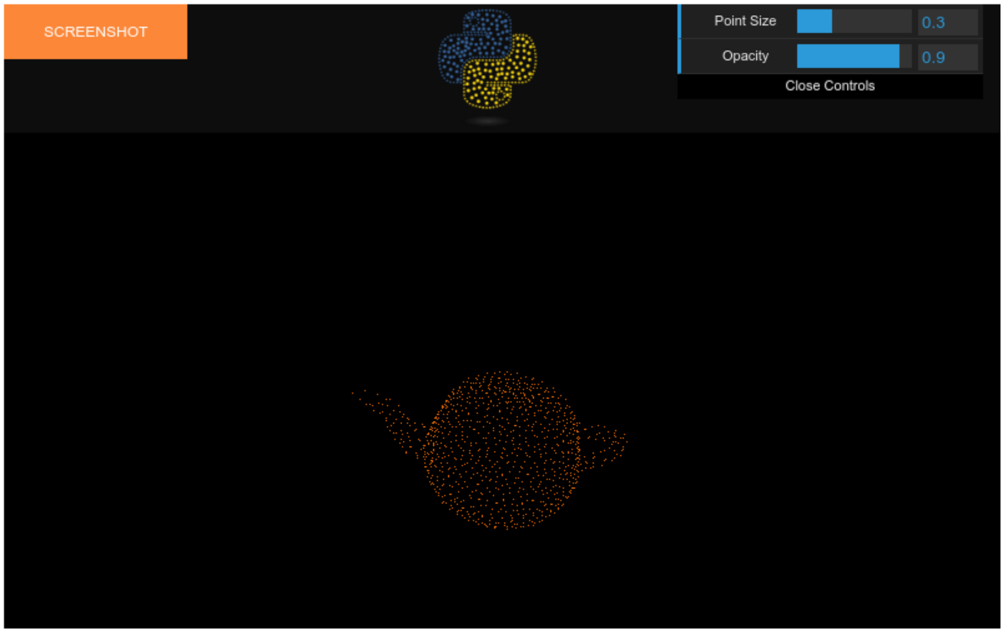
```
In [5]: points = pd.DataFrame(teapot_pts, columns=['x', 'y', 'z'])
        cloud = PyntCloud(points)
        cloud.plot(line_color='')
```

```
Out[5]: <IPython.lib.display.IFrame at 0x7fc5efb855f8>
```

```
In [6]: violin_pts = farthest_point_sampling('violin_case.obj')
```

```
In [7]: points = pd.DataFrame(violin_pts, columns=['x', 'y', 'z'])
        cloud = PyntCloud(points)
        cloud.plot(line_color='')
```

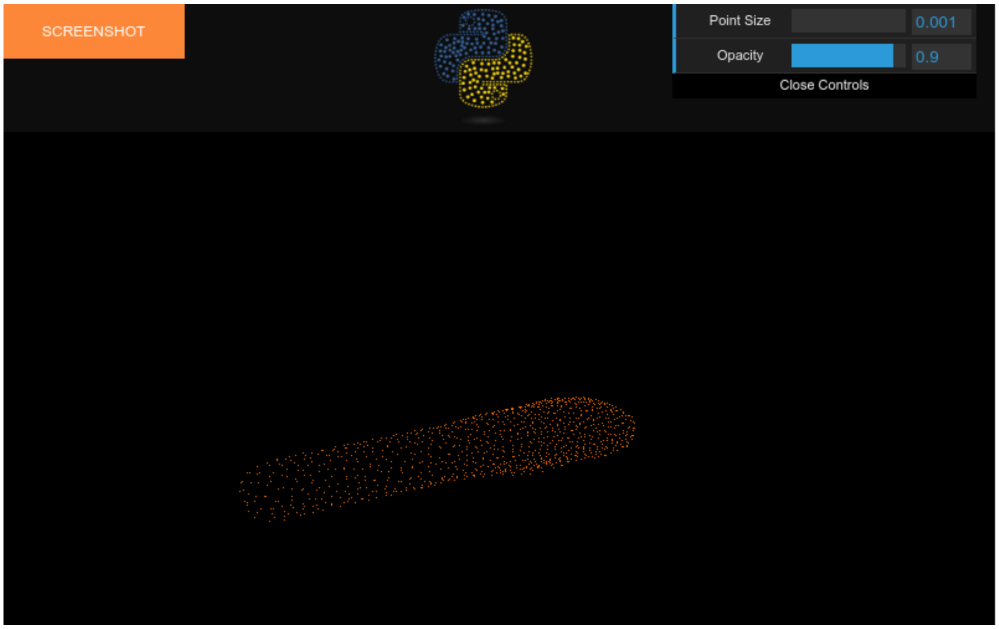
```
Out[7]: <IPython.lib.display.IFrame at 0x7fc5ef922630>
```



Point Size 0.3

Opacity 0.9

teapot



Point Size 0.001

Opacity 0.9

violin case

hw2_p3

February 17, 2018

```
In [1]: import numpy as np
        import pandas as pd
        import tensorflow as tf
        import time
        from scipy.optimize import linear_sum_assignment

        %matplotlib inline
        import matplotlib
        import matplotlib.pyplot as plt

        import pymesh
        from pyntcloud import PyntCloud

        from tf_emddistance import emd_distance
```

```
In [2]: # This section of code is copied from problem 2
        def triangle_area(x):
            a = x[:,0,:] - x[:,1,:]
            b = x[:,0,:] - x[:,2,:]
            cross = np.cross(a, b)
            area = 0.5 * np.linalg.norm(np.cross(a, b), axis=1)
            return area

        def euclidean_distance_matrix(x):
            r = np.sum(x*x, 1)
            r = r.reshape(-1, 1)
            distance_mat = r - 2*np.dot(x, x.T) + r.T
            #return np.sqrt(distance_mat)
            return distance_mat

        def update_farthest_distance(far_mat, dist_mat, s):
            for i in range(far_mat.shape[0]):
                far_mat[i] = dist_mat[i,s] if far_mat[i] > dist_mat[i,s] else far_mat[i]
            return far_mat, np.argmax(far_mat)

        def init_farthest_distance(far_mat, dist_mat, s):
            for i in range(far_mat.shape[0]):
                far_mat[i] = dist_mat[i,s]
            return far_mat

        def farthest_point_sampling(obj_file, num_samples=1000):
            mesh = pymesh.load_mesh(obj_file)
            faces = mesh.vertices[mesh.faces]
            area = triangle_area(faces)
            total_area = np.sum(area)

            set_P = []
            for i in range(faces.shape[0]):
                num_gen = area[i] / total_area * 10000
                for j in range(int(num_gen)+1):
                    r1, r2 = np.random.rand(2)
                    d = (1-np.sqrt(r1)) * faces[i,0] + np.sqrt(r1)*(1-r2) * faces[i,1] + np.sqrt
                    set_P.append(d)

            set_P = np.array(set_P)
            num_P = set_P.shape[0]

            distance_mat = euclidean_distance_matrix(set_P)

            set_S = []
            s = np.random.randint(num_P)
            far_mat = init_farthest_distance(np.zeros((num_P)), distance_mat, s)

            for i in range(num_samples):
                set_S.append(set_P[s])
                far_mat, s = update_farthest_distance(far_mat, distance_mat, s)

            return np.array(set_S, dtype=np.float32)

        def pointcloud_distance_matrix(x1, x2):
            a = tf.expand_dims(x1, axis=1)
            b = tf.expand_dims(x2, axis=0)
            distance_mat = tf.norm(a - b, axis=-1)
            return distance_mat
```

```
In [3]: # Sample cloud points -- same procedure as problem 2
        teapot_pts = farthest_point_sampling('../teapot.obj', 200)
        violin_pts = farthest_point_sampling('../violin_case.obj', 200)
```

```
In [4]: # compute distance matrix between 2 cloud points
        with tf.Session() as sess:
            distance_mat = sess.run(pointcloud_distance_matrix(teapot_pts, violin_pts))
```

```
In [5]: class hungrary():
```

```
def __init__(self, weight):
    self.n = weight.shape[0]

    self.w = np.copy(weight)
    # cost matrix
    self.c = np.copy(weight)
    self.m = np.zeros((self.n, self.n), dtype=int)

    # record row and col covers
    self.RowCover = np.zeros((self.n), dtype=bool)
    self.ColCover = np.zeros((self.n), dtype=bool)
    # record augment paths
    self.path = np.zeros((2*self.n, 2), dtype=int)

# main program, run the algo through steps
def run_hungrary(self):
    done = False
    step = 1
    while not done:
        if step == 1:
            step = self.step1()
        elif step == 2:
            step = self.step2()
        elif step == 3:
            step = self.step3()
        elif step == 4:
            step = self.step4()
        elif step == 5:
            step = self.step5()
        elif step == 6:
            step = self.step6()
        elif step == 7:
            done = True

# Each row subtract smallest elements
def step1(self):
    self.c -= np.min(self.c, axis=1, keepdims=True)
    return 2

# star zeros
def step2(self):
    for u in range(self.n):
        for v in range(self.n):
            if self.c[u,v] == 0 and not self.RowCover[u] and not self.ColCover[v]:
                self.m[u, v] = 1
                self.RowCover[u] = True
                self.ColCover[v] = True
                break
        self.clear_covers()
        return 3

    # cover cols with starred zeros. check if done
    def step3(self):
        for u in range(self.n):
            for v in range(self.n):
                if self.m[u, v] == 1:
                    self.ColCover[v] = True

        colcnt = np.sum(self.ColCover)

        if colcnt >= self.n:
            return 7
        else:
            return 4

    # find noncovered zero and prime it (starred as 2)
    def step4(self):
        while True:
            row, col = self.find_a_zero()
            if row == -1:
                return 6
            else:
                self.m[row, col] = 2
                if self.star_in_row(row):
                    col = self.find_star_in_row(row)
                    self.RowCover[row] = True
                    self.ColCover[col] = False
                else:
                    self.path_row_0 = row
                    self.path_col_0 = col
                    return 5

    # use augment algo to increase matches
    def step5(self):
        done = False
        self.path_count = 1
        self.path[self.path_count-1, 0] = self.path_row_0
        self.path[self.path_count-1, 1] = self.path_col_0

        while not done:
            row = self.find_star_in_col(self.path[self.path_count-1, 1])
            if row > -1:
                self.path_count += 1
                self.path[self.path_count-1, 0] = row
                self.path[self.path_count-1, 1] = self.path[self.path_count-2, 1]
            else:
                done = True
            if not done:
                col = self.find_prime_in_row(self.path[self.path_count-1, 0])
                self.path_count += 1
                self.path[self.path_count-1, 0] = self.path[self.path_count-2, 0]
                self.path[self.path_count-1, 1] = col

        self.augment_path()
        self.clear_covers()
        self.erase_prime()
        return 3

    # add minval val to double covered elements and subtract it to noncovered elements
    def step6(self):
        minval = self.find_smallest()
        for u in range(self.n):
            for v in range(self.n):
                if self.RowCover[u]:
                    self.c[u,v] += minval
                if not self.ColCover[v]:
                    self.c[u,v] -= minval
        return 4

    # find first uncovered zero
    def find_a_zero(self):
        for u in range(self.n):
            for v in range(self.n):
                if self.c[u,v] == 0 and not self.RowCover[u] and not self.ColCover[v]:
                    return u, v
        return -1, -1

    def star_in_row(self, row):
        for v in range(self.n):
            if self.m[row, v] == 1:
                return True
        return False

    def find_star_in_row(self, row):
        for v in range(self.n):
            if self.m[row, v] == 1:
                return v
        return -1

    def find_star_in_col(self, col):
        for u in range(self.n):
            if self.m[u, col] == 1:
                return u
        return -1
```

```
            def find_prime_in_row(self, row):
                for v in range(self.n):
                    if self.m[row, v] == 2:
                        return v
                return -1

            def augment_path(self):
                for p in range(self.path_count):
                    if self.m[self.path[p,0], self.path[p,1]]  == 1:
                        self.m[self.path[p,0], self.path[p,1]] = 0
                    else:
                        self.m[self.path[p,0], self.path[p,1]] = 1

            def clear_covers(self):
                self.RowCover = np.zeros((self.n), dtype=bool)
                self.ColCover = np.zeros((self.n), dtype=bool)

            def erase_prime(self):
                for u in range(self.n):
                    for v in range(self.n):
                        if self.m[u,v] == 2:
                            self.m[u,v] = 0

            def find_smallest(self):
                minval = np.max(self.c)
                for u in range(self.n):
                    for v in range(self.n):
                        if self.c[u,v] < minval and not self.RowCover[u] and not self.ColCover[v
                            minval = self.c[u,v]
                return minval
```

```
In [7]: H = hungrary(distance_mat)
        H.run_hungrary()
```

```
In [8]: # EMD computed by my hungarian algorithm
        np.sum(H.w * H.m)
```

```
Out[8]: 10703.283664703369
```

```
In [9]: # EMD computed by scipy linear_sum_assignment
        row_ind, col_ind = linear_sum_assignment(distance_mat)
        print(distance_mat[row_ind, col_ind].sum())
```

```
10703.3
```

hw2_p4

February 16, 2018

```
In [1]: import numpy as np
        import tensorflow as tf
        %matplotlib inline
        import matplotlib
        import matplotlib.pyplot as plt
```

```
In [2]: from tensorflow.examples.tutorials.mnist import input_data
        mnist = input_data.read_data_sets("MNIST_data/", one_hot=False)
        print(mnist.train.images.shape, mnist.train.labels.shape)
```

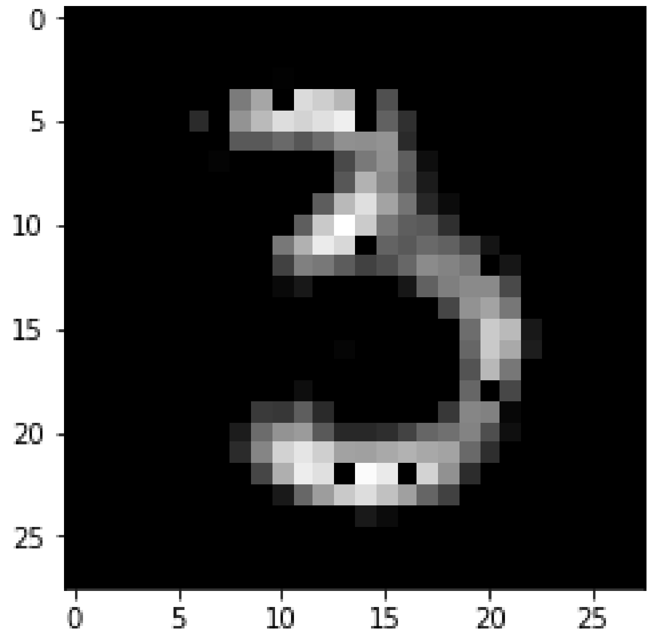
```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
(55000, 784) (55000,)
```

```
In [3]: def show_image(image):
            plt.gray()
            plt.imshow(image.reshape(28, 28))
            plt.show()

        # add gaussian noise to image -- mean = 0, variance = 0.1
        def add_noise(image):
            mu, sigma = 0, 0.1
            gauss = np.random.normal(mu, sigma, 28*28)
            return image + gauss
```

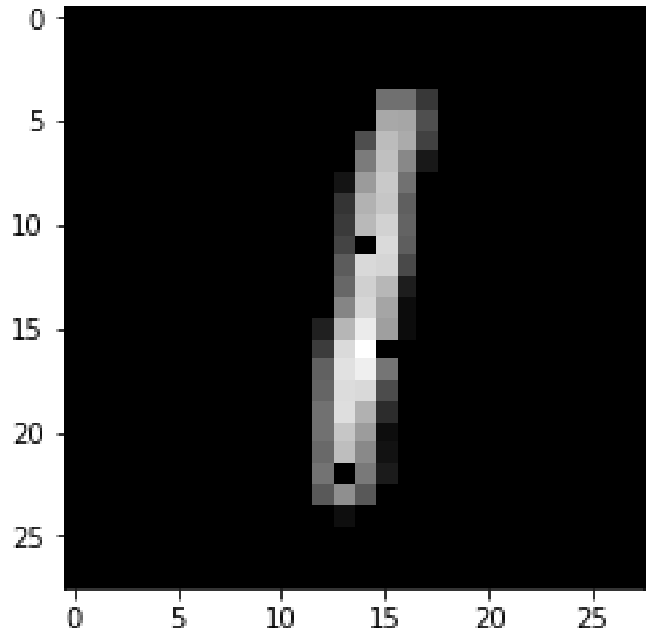
```
In [4]: # This is an encoder
        # 2 convolutional layers and 1 fully connected layer
        def encoder(x):
            x_image = tf.reshape(x, [-1, 28, 28, 1])

            conv1 = tf.contrib.layers.conv2d(x_image, 16, [3,3], stride=2, padding='VALID')

            conv2 = tf.contrib.layers.conv2d(conv1, 32, [3,3], stride=2, padding='VALID')

            pool2_flat = tf.reshape(conv2, [-1, 6*6*32])
            fc = tf.contrib.layers.fully_connected(pool2_flat, 100)

            return fc

        # This is a decoder -- well commented!
        # 1 fully connected layer, 2 convolutional layers and 1 fc layer
        def decoder(x):
            fc = tf.contrib.layers.fully_connected(x, 6*6*32)
            fc_layer = tf.reshape(fc, [-1, 6, 6, 32])

            deconv1 = tf.contrib.layers.conv2d_transpose(fc_layer, 16, [3,3], stride=2, padding=
            deconv2 = tf.contrib.layers.conv2d_transpose(deconv1, 1, [3,3], stride=2, padding='V
            deconv2_flat = tf.reshape(deconv2, [-1, 27*27])
            fc = tf.contrib.layers.fully_connected(deconv2_flat, 28*28)
            return fc
```

```
In [5]: with tf.device('/device:GPU:0'):
            # noisy image
            x = tf.placeholder(tf.float32, shape=[None, 784], name='x')
            # origin image (before adding noise)
            y = tf.placeholder(tf.float32, shape=[None, 784], name='y')

        # reconstruct images
        pred = decoder(encoder(x))
        # MSE cost function
        #compute difference between reconstructed images and imges before adding noise
        cost = tf.reduce_mean(tf.squared_difference(y, pred))
        optimizer = tf.train.AdamOptimizer(1e-4).minimize(cost)
```

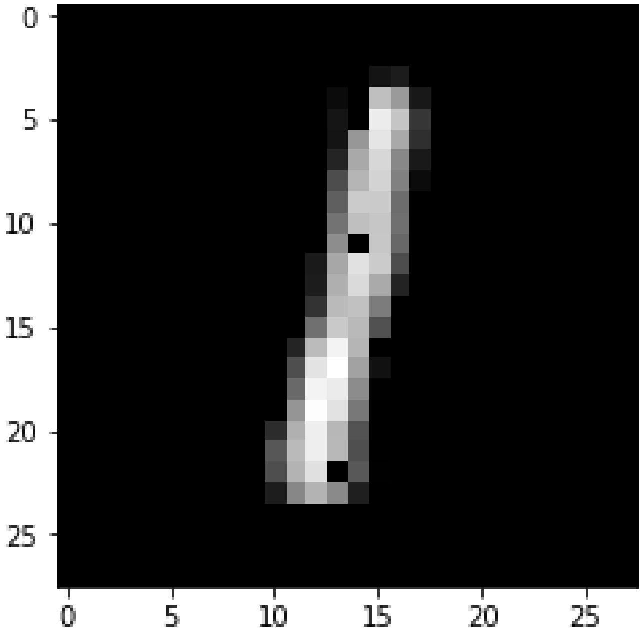
```
In [6]: epoch = 15000
        batch_size = 16
        config = tf.ConfigProto()
        config.gpu_options.allow_growth = True

        sess = tf.Session(config=config)
        sess.run(tf.global_variables_initializer())

        for i in range(epoch):
            batch_x, _ = mnist.train.next_batch(batch_size)
            _, batch_cost = sess.run([optimizer, cost], feed_dict = {x:add_noise(batch_x), y:bat

            # show reconstructed images after every 5000 iterations of training
            if (i+1) % 5000 == 0:
                batch_x, _ = mnist.test.next_batch(1)
                noise_image = add_noise(batch_x)
                gen_image, test_cost = sess.run([pred,cost], feed_dict = {x:noise_image, y:batch
                #show_image(noise_image)
                show_image(gen_image)
                print('%d epochs-- train cost: %f    test cost: %f' % (i+1, batch_cost, test_cos
```



```
5000 epochs-- train cost: 0.015710    test cost: 0.017580
```



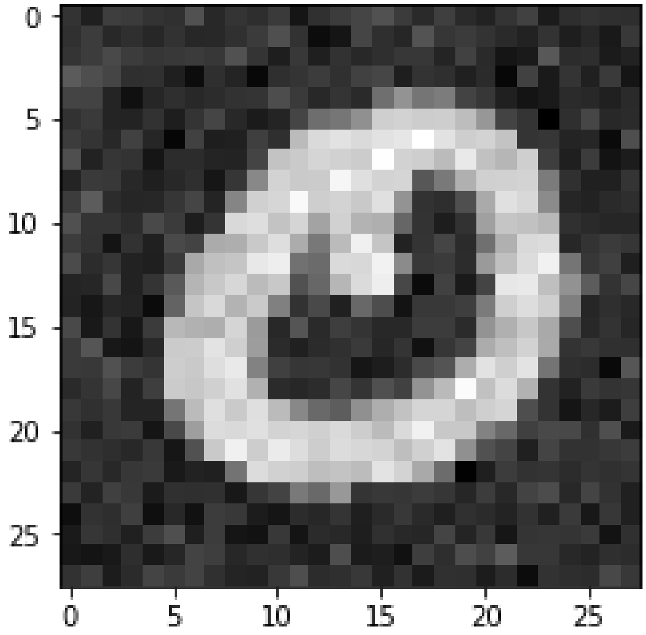
```
10000 epochs-- train cost: 0.013543    test cost: 0.005779
```



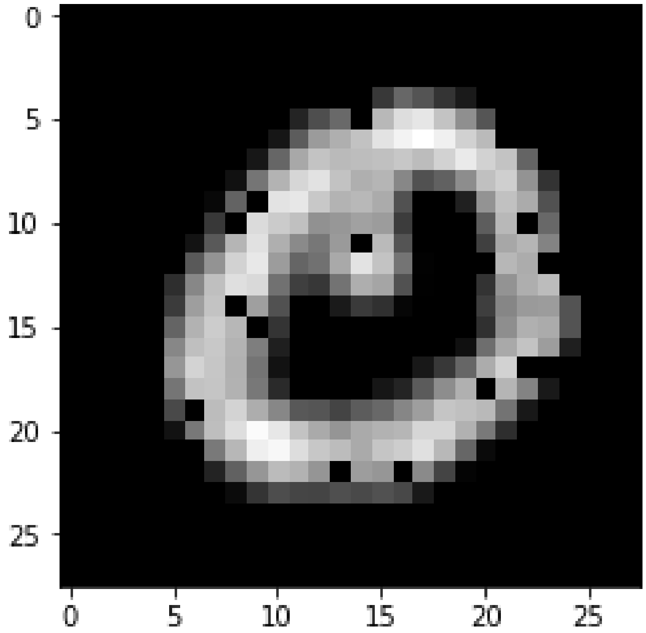
```
15000 epochs-- train cost: 0.014058    test cost: 0.006856
```

```
In [7]: # Testing -- reconstruct images from test set
        batch_x, _ = mnist.test.next_batch(1)
        noise_image = add_noise(batch_x)
        gen_image, test_cost = sess.run([pred,cost], feed_dict = {x:noise_image, y:batch_x})
        print('Noisy image')
        show_image(noise_image)
        print('Generated image')
        show_image(gen_image)
```

```
Noisy image
```

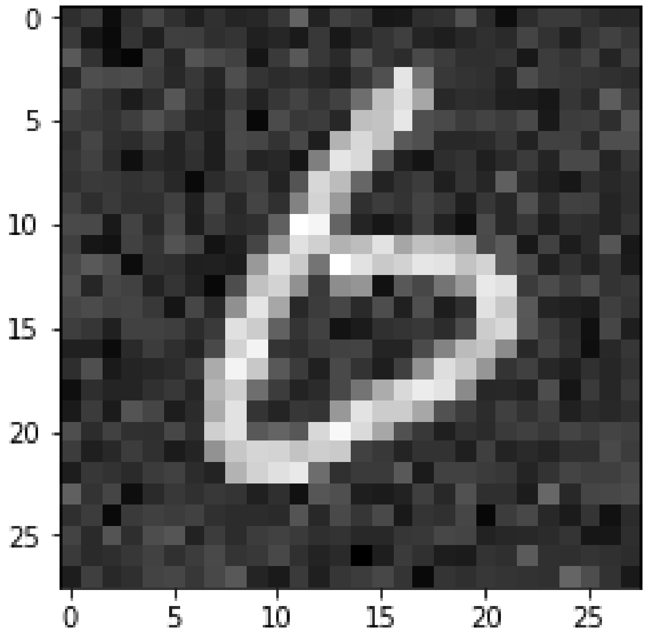


Generated image

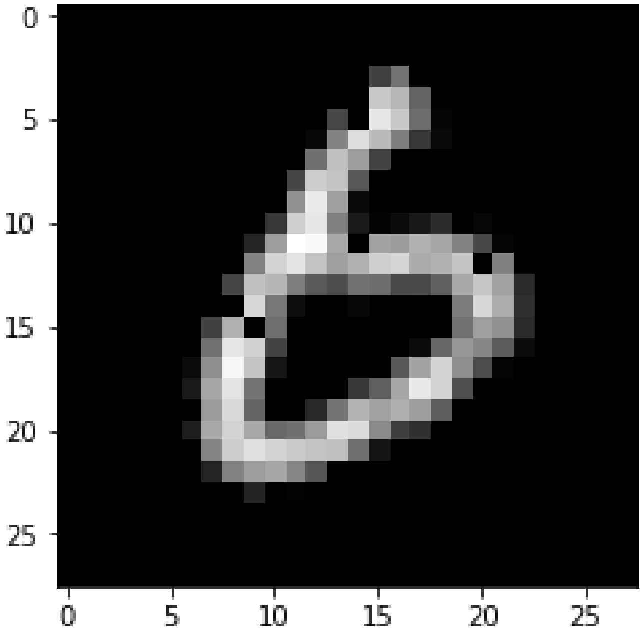


```
In [8]: # Testing -- reconstruct images from test set
        batch_x, _ = mnist.test.next_batch(1)
        noise_image = add_noise(batch_x)
        gen_image, test_cost = sess.run([pred,cost], feed_dict = {x:noise_image, y:batch_x})
        print('Noisy image')
        show_image(noise_image)
        print('Generated image')
        show_image(gen_image)
```

```
Noisy image
```



```
Generated image
```


0
5
10
15
20
25
0
5
10
15
20
25