CS 189: Introduction to Machine Learning

Fall 2017

•

Homework 14

•

Solutions by

Jinhong Du

3033483677

Question 1

(a)

Jinhong Du
jaydu@berkeley.edu

(b)

I certify that all solutions are entirely in my words and that I have not looked at another student's solutions. I have credited all external sources in this write up. Jinhong Du

Question 2

(a)

$\because$

$$
\begin{aligned}
&\sum_{k=1}^{K}\sum_{i\in\pi_k}\|x_i-\mu_k\|_2^2\\
&=\left\|\begin{bmatrix} x_1^T-\mu_{x_1}^T\\ \vdots\\ x_n^T-\mu_{x_n}^T\end{bmatrix}\right\|_F^2\\
&=\left\|X-\begin{bmatrix}\mu_{x_1}^T\\ \vdots\\ \mu_{x_n}^T\end{bmatrix}\right\|_F^2\\
&=\left\|X-\begin{bmatrix}y_1^T\\ \vdots\\ y_n^T\end{bmatrix}\mu\right\|_F^2\\
&=\|X-Y\mu\|_F^2
\end{aligned}
$$

where $\mu_{x_i}^T$ denote μ_k^T for k such that $i\in\pi_k$.

$\therefore$

$$\min_{\pi,\mu}\sum_{k=1}^{K}\sum_{i\in\pi_k}\|x_i-\mu_k\|_2^2$$

is equivalent to

$$\min_{\mu,Y}\|X-Y\mu\|_F^2$$

suject to

$$y_i\in\{0,1\}^K,\|y_i\|_0=1$$

(b)

$\because$

$$\|x-\sum_{D_j\in B_l\cup\{D_k\}}\beta_jD_j\|_2^2\leqslant\|x-\sum_{D_j\in B_l\cup\{D_k\}}\beta_jD_j\|_2^2+\|\beta_kD_k\|_2^2$$

i.e.

$$R_l^2\leqslant R_{l-1}^2+\|\beta_kD_k\|_2^2$$

$\therefore$

$$\min_{\beta,k}R_l^2\leqslant\min_{\beta,k}R_{l-1}^2$$

by setting $\beta_k=0$. I.e., each step cannot increase the residue error of linear regression.

(c)

From (b), we have z'_n from Sparse-Coding-Single(x_n, s) cannot increase the value of objective function (1). And also in Algorithm 2, objective function of z'_n will be updated when its value is smaller than the previous one. Therefore, Algorithm 2 cannot increase the value of the objective function.

(d)

From the Eckart-Young theorem, the cloest 1-rank matrix of E_k^R is $u_1\Lambda_{11}v_1^T$ (in the sense of F norm). Therefore, so if replace Z_k with $u_1\Lambda_{11}$ and D_k with v_1,

$$\left\| X - \sum_{j\neq k} Z_j D_j^T - u_1 \Lambda v_1^T \right\|_F^2 \leqslant \|X - ZD\|_F^2$$

the value of objective function won't increase. And with only replace the nonzero value of Z_k, the sparsity constraint is preserved.

(e)

From the previous part, we have Algorithm 2 and 4 won't increase the error. Thus the K-SVD can converge.

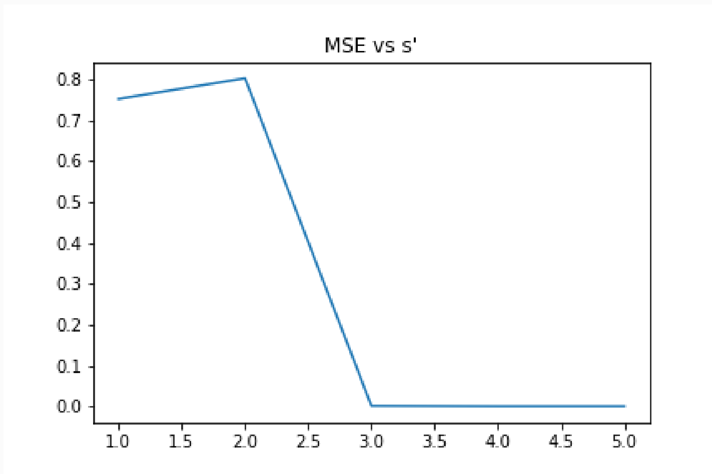
(f)

```
def update_codebook(Z, X, D):
N, K = np.shape(Z)
for k in range(K):
    wk = Z[:,k]!=0
    if np.any(wk):
        Ek = X - Z.dot(D) + Z[:,k][:,np.newaxis].dot(
          D[k,:][np.newaxis,:])
        EkR = Ek[wk,:]
        u,s,vT = np.linalg.svd(EkR)
        D[k,:] = vT[0,:]
        Z[wk,k] = u[:,0]*s[0]
    return D

def KSVD(X, K, s):
    N, d = np.shape(X)
    D = X[np.random.choice(N, K, replace=False),:]
    error = np.inf
    while 1:
        Z = sparse_coding(D, X, s)
        D = update_codebook(Z, X, D)
        new_error = compute_error(X,D,Z)
        if np.abs(error-new_error)<1e-1:
```

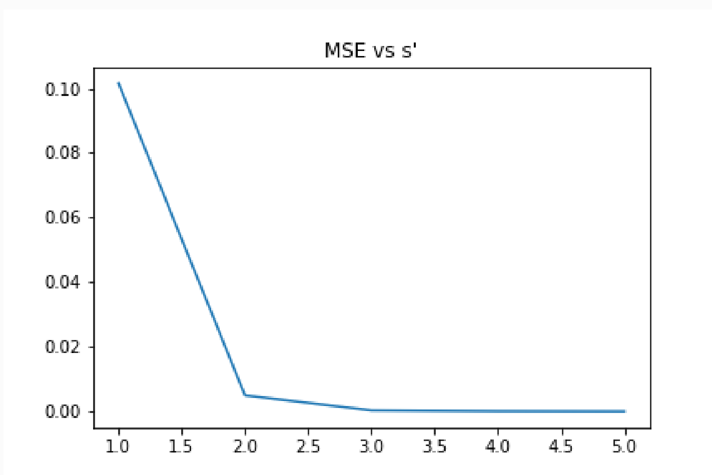
Solution (cont.)

```
                break
            error = new_error
        return D,Z
```

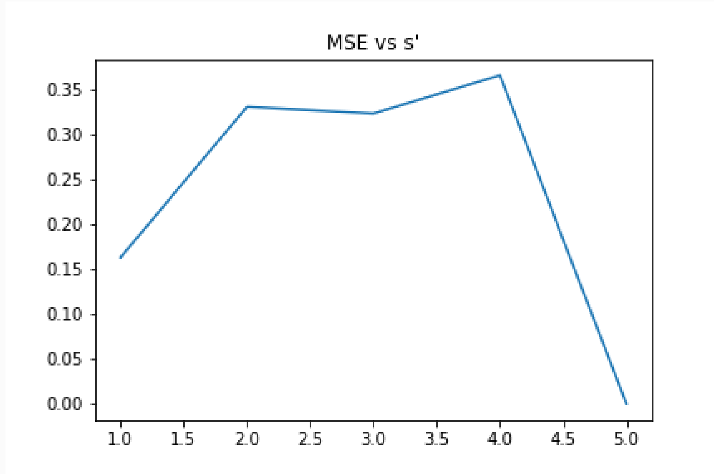
(g)



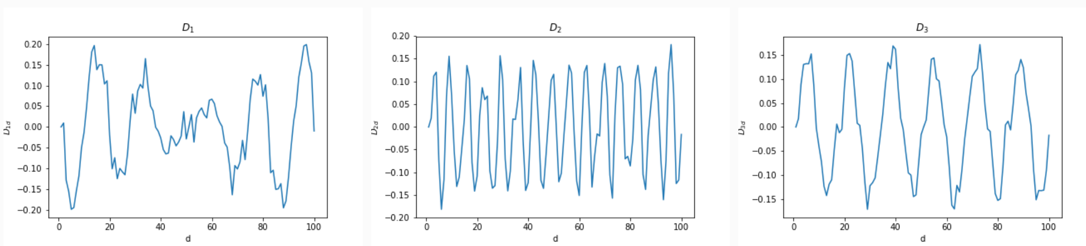
(h)



(i)

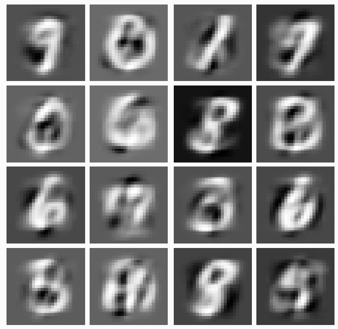

MSE vs s'
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0

(j)


D_1
D_2
D_3
d
d
d

Question 3

(a)



(b)

```python
class GAN():
    def __init__(self,input_size = 784, random_size = 100):
        self.input_size = input_size
        self.random_size = random_size

    def xavier_init(self,size):
        in_dim = size[0]
        xavier_stddev = 1. / tf.sqrt(in_dim / 2.)
        return tf.random_normal(shape=size, stddev=xavier_stddev)

    def sample_Z(self,m, n):
        return np.random.uniform(-1., 1., size=[m, n])

    def generator(self,z):
        ###IMPLEMENT THE GENERATOR USING THE G_ VARIABLES#####
        G_h1 = tf.nn.relu(tf.matmul(z, self.G_W1) + self.G_b1)
        G_log_prob = tf.matmul(G_h1, self.G_W2) + self.G_b2
        self.G_prob = tf.nn.sigmoid(G_log_prob)
        return self.G_prob


    def discriminator(self,x):
        ###IMPLEMENT THE DISCRIMENATOTR USING THE D_ VARIABLES#####
        D_h1 = tf.nn.relu(tf.matmul(x, self.D_W1) + self.D_b1)
        D_logit = tf.matmul(D_h1, self.D_W2) + self.D_b2
        D_prob = tf.nn.sigmoid(D_logit)
        return D_prob, D_logit

```

Solution (cont.)

```python
    def init_training(self):

        self.X = tf.placeholder(tf.float32,
            shape=[None, self.input_size])

        self.Z = tf.placeholder(tf.float32,
            shape=[None, self.random_size])

        self.G_W1 = tf.Variable(
            self.xavier_init([self.random_size, 128]))
        self.G_b1 = tf.Variable(
            tf.zeros(shape=[128]))

        self.G_W2 = tf.Variable(
            self.xavier_init([128, self.input_size]))
        self.G_b2 = tf.Variable(
            tf.zeros(shape=[self.input_size]))


        self.theta_G = [self.G_W1, self.G_W2, self.G_b1, self.G_b2]

        self.D_W1 = tf.Variable(
            self.xavier_init([self.input_size,128]))
        self.D_b1 = tf.Variable(tf.zeros(shape=[128]))

        self.D_W2 = tf.Variable(self.xavier_init([128, 1]))
        self.D_b2 = tf.Variable(tf.zeros(shape=[1]))

        self.theta_D = [self.D_W1, self.D_W2, self.D_b1, self.D_b2]


        self.G_sample = self.generator(self.Z)
        D_real, D_logit_real = self.discriminator(self.X)
        D_fake, D_logit_fake = self.discriminator(self.G_sample)


    # Implement the loss functions for training a GAN
    # ---------------------------------------------
        D_loss_real = tf.reduce_mean(
            tf.nn.sigmoid_cross_entropy_with_logits(
            logits = D_logit_real,
            labels = tf.ones_like(D_logit_real)))
        D_loss_fake = tf.reduce_mean(
```

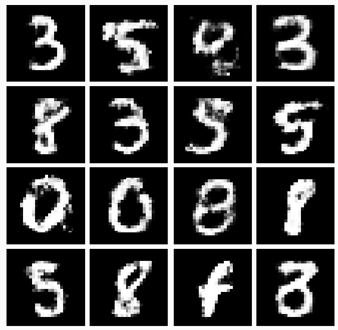
Solution (cont.)

```
                tf.nn.sigmoid_cross_entropy_with_logits(
                logits = D_logit_fake,
                labels = tf.zeros_like(D_logit_fake)))
            self.D_loss = D_loss_real + D_loss_fake
            self.G_loss = tf.reduce_mean(
                tf.nn.sigmoid_cross_entropy_with_logits(
                logits = D_logit_fake,
                labels = tf.ones_like(D_logit_fake)))

            self.D_solver = tf.train.AdamOptimizer(
              ).minimize(self.D_loss, var_list=self.theta_D)
            self.G_solver = tf.train.AdamOptimizer(
              ).minimize(self.G_loss, var_list=self.theta_G)

        def generate_sample(self,num_samples):
            ####GENERATE SAMPLES FROM THE GAN############
            samples = self.sess.run(self.G_sample,
              feed_dict={self.Z: self.sample_Z(num_samples, self.Z_dim)})
            return samples

        def train_model(self,data):

            mb_size = 128
            self.Z_dim = self.random_size

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

            for it in range(100000):
                X_mb, _ = data.train.next_batch(mb_size)

                _, D_loss_curr = self.sess.run(
                  [self.D_solver, self.D_loss],
                  feed_dict={self.X: X_mb, self.Z: self.sample_Z(mb_size,
                    self.Z_dim)})
                _, G_loss_curr = self.sess.run(
                  [self.G_solver, self.G_loss],
                  feed_dict={self.Z: self.sample_Z(mb_size, self.Z_dim)})

                if it % 10000 == 0:
                    print('Iter: {}'.format(it))
                    print('D_loss: {:.4}'. format(D_loss_curr))
                    print('G_loss: {:.4}'.format(G_loss_curr))
```

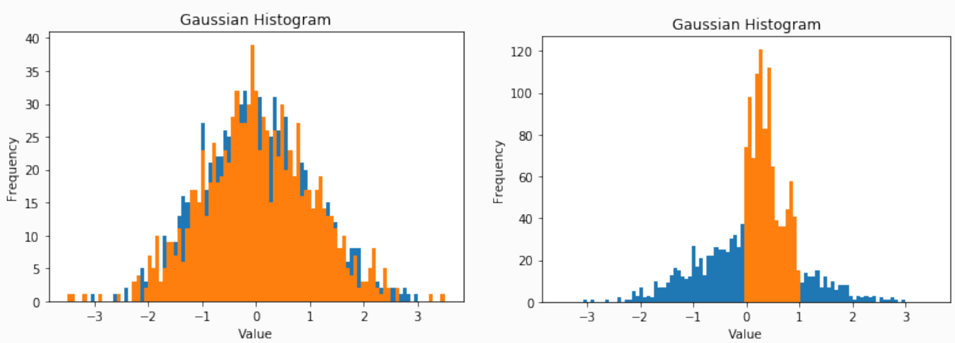
Solution (cont.)

```
                        samples = self.sess.run(self.G_sample, feed_dict={
                                    self.Z: self.sample_Z(16, self.Z_dim)})
                        fig = plot(samples)
                        plt.show()
```

(c)



(d)



Question 4

Question What are the differences between VAE and GAN?
Solution
Introduction:

(1) Variational Autoencoder (VAE)

Various techniques exist to prevent autoencoders from learning the identity function and to improve their ability to capture important information and learn richer representations.

(2) Generative Adversarial Network (GAN)

Two neural networks contesting with each other in a zero-sum game framework.

Differences:

(1) For generator, VAE $\min_P D_{KL}(Q\|P)$ while GAN $\min_P D_{KL}(P\|Q)$ where Q is posterior and P is inference.

(2) It is more esier to find out the map in VAE by looking into the encoder and decoder. For GAN, we basically don't know what output will correspond to a given input.

(3) Pictures generated by VAE tend to be vague since the loss of VAE is the square root of the generated pictures and the corresponding raw pictures.

HW 14

December 4, 2017

1 Question 2

1.1 (f)

```
In [1]: import sklearn.decomposition
        import matplotlib.pyplot as plt
        import numpy as np
        import scipy

In [2]: def sparse_coding(D,X,s):
            '''
            This function implements sparse coding in the pseudo code.
            '''
            Z = sklearn.decomposition.sparse_encode(X,
                                                    D,
                                                    algorithm='omp',
                                                    alpha = 1.,
                                                    n_nonzero_coefs=s,
                                                   max_iter = 100)
            return Z


        def compute_error(X,D,Z):
            """
            Compute reconstruction MSE.
            """

            error = np.linalg.norm(X - Z.dot(D), ord='fro')**2/len(X)
            return error

        def generate_Z(N,K,s_true):
            """
            Generate random coefficient matrix.
            """
            Z = np.zeros((N,K))
            zero_one_vec = np.zeros(K).astype(int)
            zero_one_vec[:s_true] = 1
```

```
        sparse_indicator = np.array([np.random.permutation(zero_one_vec) for i in range(N)])
        Z[np.where(sparse_indicator)]=1.0
        return Z


    def generate_test_data(N, s, K, d):
        """
        Generate a dataset for the testing purpose.
        """
        Z = generate_Z(N,K,s)
        D = np.random.randn(K,d)
        X = Z.dot(D)
        return X

    def generate_toy_data(N, s, K, d, c):
        def f(j):
            return 10/j
        sampling_loc = np.expand_dims(np.arange(0,1,1.0/d),0)
        freq = np.expand_dims(np.arange(1,K+1),1)

        D = np.sin(2*np.pi*freq.dot(sampling_loc))
        D = D / np.expand_dims(np.sqrt(np.sum(D**2, axis = 1)),1)
        Z = generate_Z(N,K,s)

        X = Z.dot(D) + c * np.random.randn(N,d)
        return X,D,Z
```

```
In [3]: def update_codebook(Z, X, D):
            N, K = np.shape(Z)
            for k in range(K):
                wk = Z[:,k]!=0
                if np.any(wk):
                    Ek = X - Z.dot(D) + Z[:,k][:,np.newaxis].dot(D[k,:][np.newaxis,:])
                    EkR = Ek[wk,:]
                    u,s,vT = np.linalg.svd(EkR)
                    D[k,:] = vT[0,:]
                    Z[wk,k] = u[:,0]*s[0]
            return D
```

```
In [4]: def KSVD(X, K, s):
            N, d = np.shape(X)
            D = X[np.random.choice(N, K, replace=False),:]
            error = np.inf
            while 1:
                Z = sparse_coding(D, X, s)
                D = update_codebook(Z, X, D)
                new_error = compute_error(X,D,Z)
                if np.abs(error-new_error)<1e-1:
```

```
                break
            error = new_error
        return D,Z
```

```
In [5]: N = 1000
        d = 10
        K = 10
        s = 2
        Z = np.zeros((N,K))
        for i in range(N):
            Z[i,np.random.choice(K,s,replace=False)]=np.ones((s))
        D = np.random.randn(d,K)
        X = Z.dot(D)
        Dh, Zh = KSVD(X,K,s)
        print(compute_error(X,Dh,Zh))
```

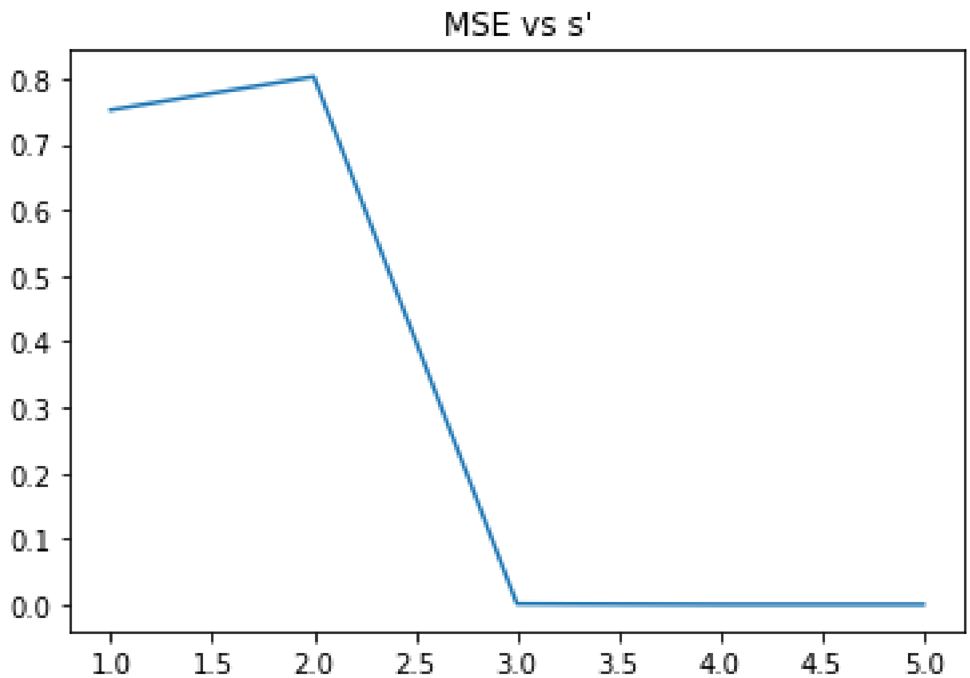
```
/anaconda/lib/python3.6/site-packages/sklearn/decomposition/dict_learning.py:152: RuntimeWarning
dependence in the dictionary. The requested precision might not have been met.

  copy_Xy=copy_cov).T


2.20034014249
```

1.2 (g)

```
In [6]: X,D,Z = generate_toy_data(200,3,5,20,0)
        MSE = []
        for i in range(1,6):
            Dh, Zh = KSVD(X,K,i)
            MSE.append(compute_error(X,Dh,Zh))
```

```
/anaconda/lib/python3.6/site-packages/sklearn/decomposition/dict_learning.py:152: RuntimeWarning
dependence in the dictionary. The requested precision might not have been met.

  copy_Xy=copy_cov).T
```

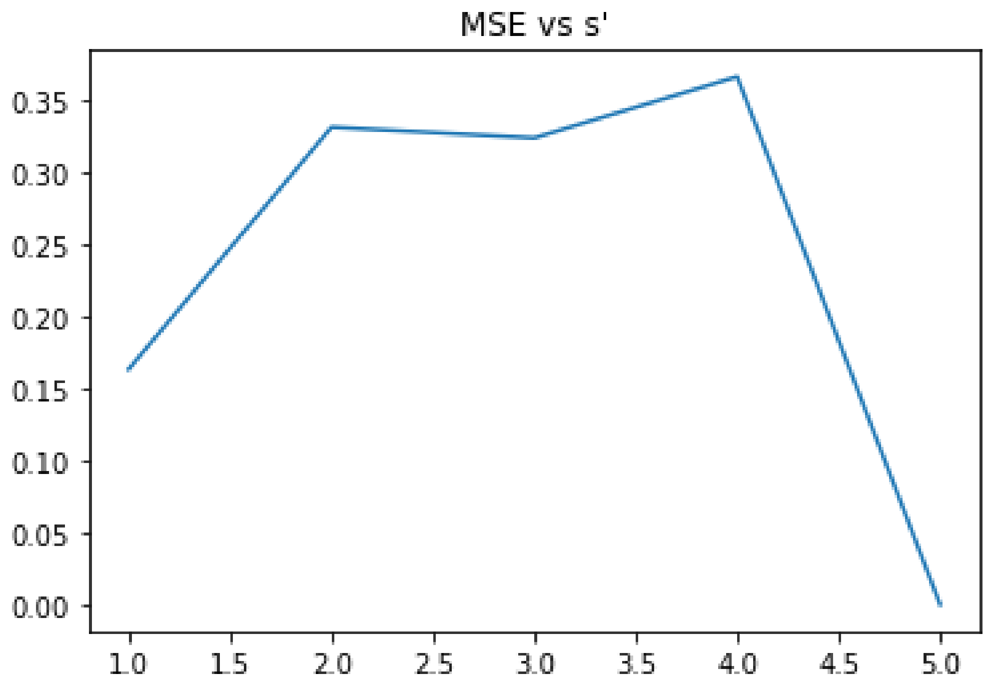
```
In [7]: plt.plot(range(1,6),MSE)
        plt.title('MSE vs s\'')
        plt.savefig('2g.png')
        plt.show()
```



1.3 (h)

```
In [8]: X,D,Z = generate_toy_data(200,2,20,5,0)
        MSE = []
        for i in range(1,6):
            Dh, Zh = KSVD(X,K,i)
            MSE.append(compute_error(X,Dh,Zh))
        plt.plot(range(1,6),MSE)
        plt.title('MSE vs s\'')
        plt.savefig('2h.png')
        plt.show()
```

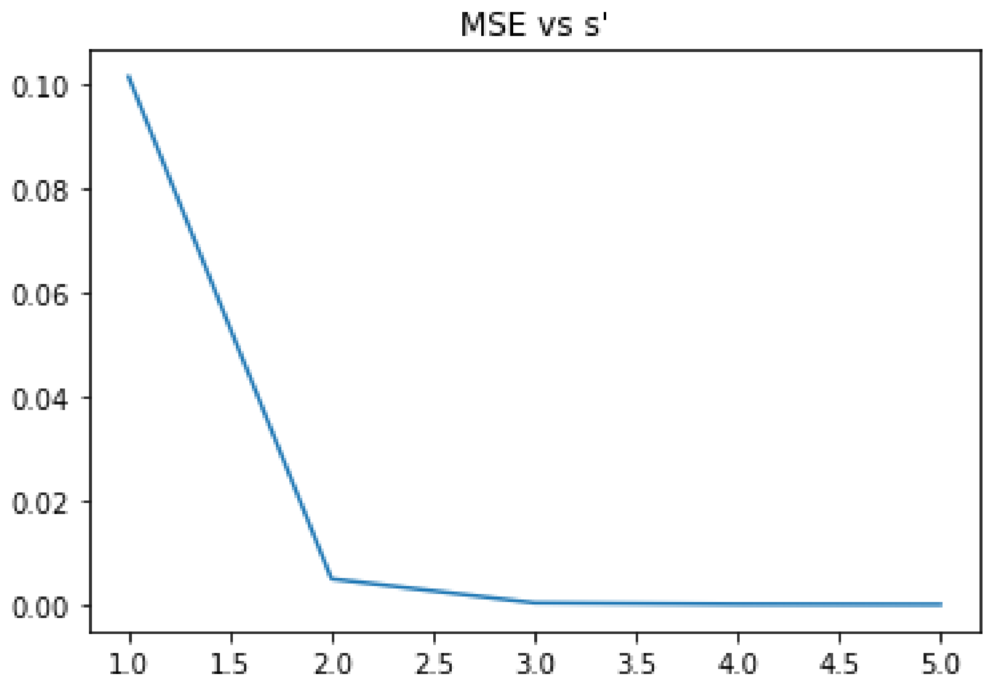
```
/anaconda/lib/python3.6/site-packages/sklearn/decomposition/dict_learning.py:152: RuntimeWarning
dependence in the dictionary. The requested precision might not have been met.

  copy_Xy=copy_cov).T
```



1.4 (i)

```
In [9]: X,D,Z = generate_toy_data(200,3,20,5,0.0001)
        MSE = []
        for i in range(1,6):
            Dh, Zh = KSVD(X,K,i)
            MSE.append(compute_error(X,Dh,Zh))
        plt.plot(range(1,6),MSE)
        plt.title('MSE vs s\'')
        plt.savefig('2i.png')
        plt.show()
```

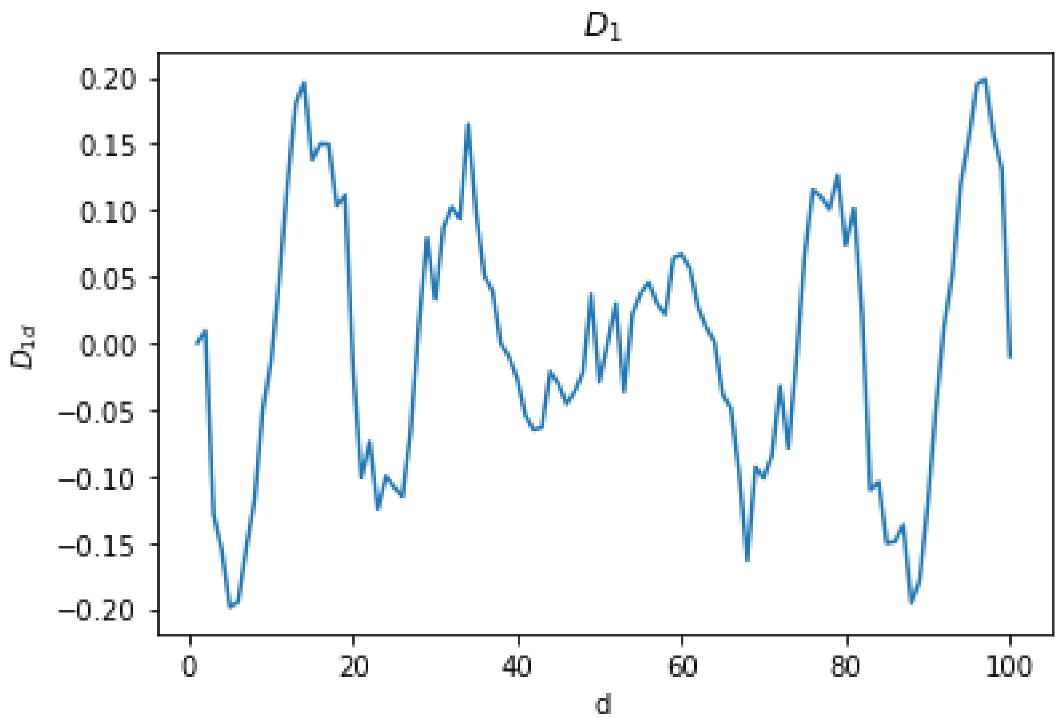
```
/anaconda/lib/python3.6/site-packages/sklearn/decomposition/dict_learning.py:152: RuntimeWarning
dependence in the dictionary. The requested precision might not have been met.

  copy_Xy=copy_cov).T
```

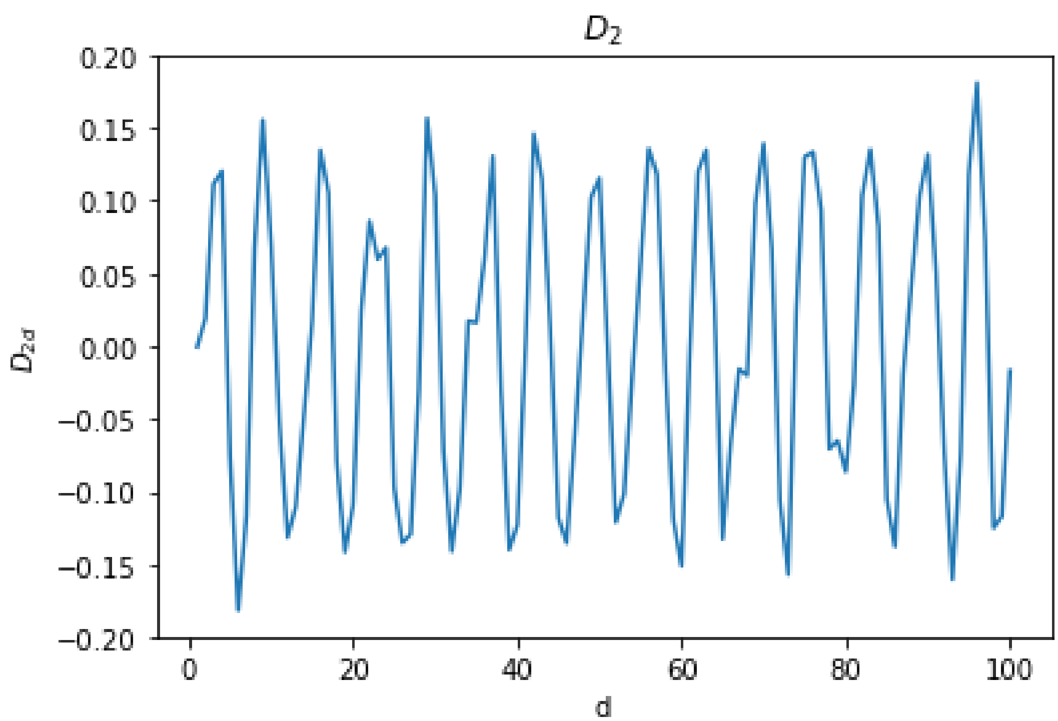


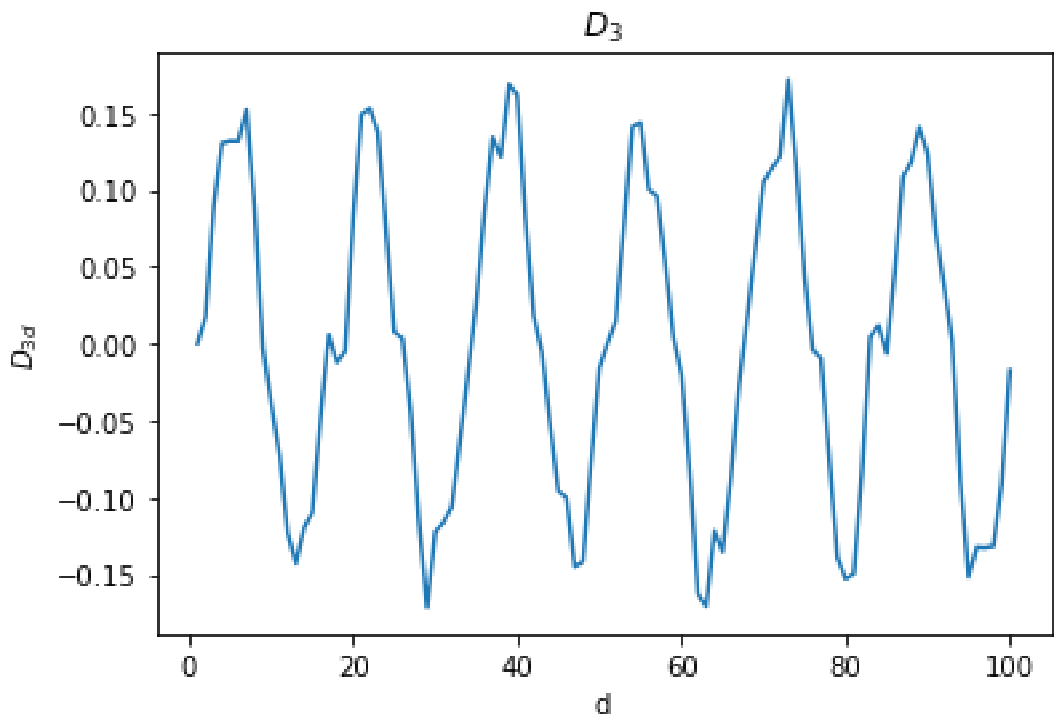
1.5 (j)

```
In [11]: X,D,Z = generate_toy_data(200,2,200,100,0.001)
         Dh, Zh = KSVD(X,K,i)
         title = ['$D_1$','$D_2$','$D_3$']
         ylabel = ['$D_{1d}$','$D_{2d}$','$D_{3d}$']
         for i in range(3):
             plt.plot(range(1,101),Dh[i,:])
             plt.title(title[i])
             plt.xlabel('d')
             plt.ylabel(ylabel[i])
             plt.savefig('2j'+str(i+1)+'.png')
             plt.show()
```

```
/anaconda/lib/python3.6/site-packages/sklearn/decomposition/dict_learning.py:152: RuntimeWarning
dependence in the dictionary. The requested precision might not have been met.

  copy_Xy=copy_cov).T
```


D_1
D_{1d}
d
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
0
20
40
60
80
100


D_2
D_{2d}
d
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
0
20
40
60
80
100



2 Question 3

2.1 (a)

```
In [11]: from sklearn.datasets import load_digits
         from sklearn.neighbors import KernelDensity
         from sklearn.decomposition import PCA
         from sklearn.model_selection import GridSearchCV


         class KDE():
             def __init__(self,use_pca = True):

                 self.use_pca = use_pca


             def train_model(self,data,pca=True):
                 # project the 64-dimensional data to a lower dimension

                 if self.use_pca:
                     self.pca = PCA(n_components=15, whiten=False)

                     data = self.pca.fit_transform(data)
```

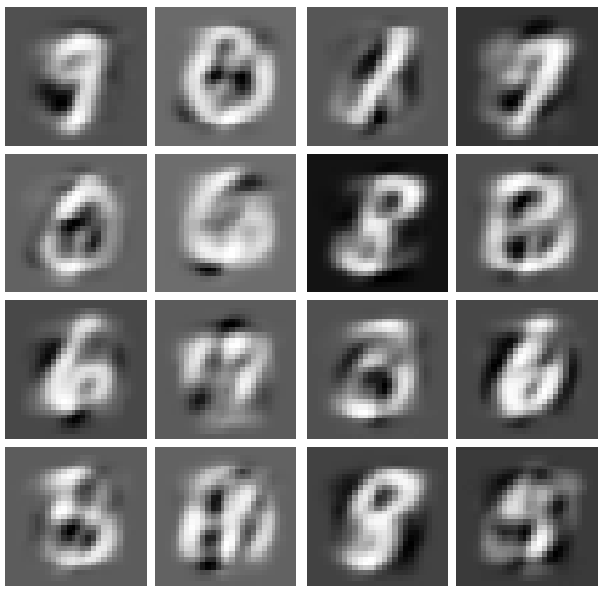
```
            # use grid search cross-validation to optimize the bandwidth
            params = {'bandwidth': np.logspace(-1, 1, 20)}

            ###FILL IN KDE FITTING AND GRIDSEARCH OPTIMIZATION
            KD = KernelDensity()
            self.clf = GridSearchCV(KD,params)
            self.clf.fit(data)

        def generate_sample(self,K):
            ###GENERATE SAMPLES FROM KDE
            new_data = self.clf.best_estimator_.sample(K)
            if self.use_pca:
                new_data = self.pca.inverse_transform(new_data)

            return new_data
```

In [12]:
```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
import matplotlib.gridspec as gridspec
import os

def plot(samples):
    fig = plt.figure(figsize=(4, 4))
    gs = gridspec.GridSpec(4, 4)
    gs.update(wspace=0.05, hspace=0.05)

    for i, sample in enumerate(samples):
        ax = plt.subplot(gs[i])
        plt.axis('off')
        ax.set_xticklabels([])
        ax.set_yticklabels([])
        ax.set_aspect('equal')
        plt.imshow(sample.reshape(28, 28), cmap='Greys_r')

    return fig


mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
train_data = mnist.train.images[0:2000,:]


# ####TRAIN KDE######
kde_model = KDE()
kde_model.train_model(train_data)
samples = kde_model.generate_sample(16)
```

```
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.
Extracting MNIST_data/train-images-idx3-ubyte.gz
```

```
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

In [13]:
```
fig = plot(samples)
plt.savefig('kde_mnist.png', bbox_inches='tight')
plt.show()
```



2.2 (b)

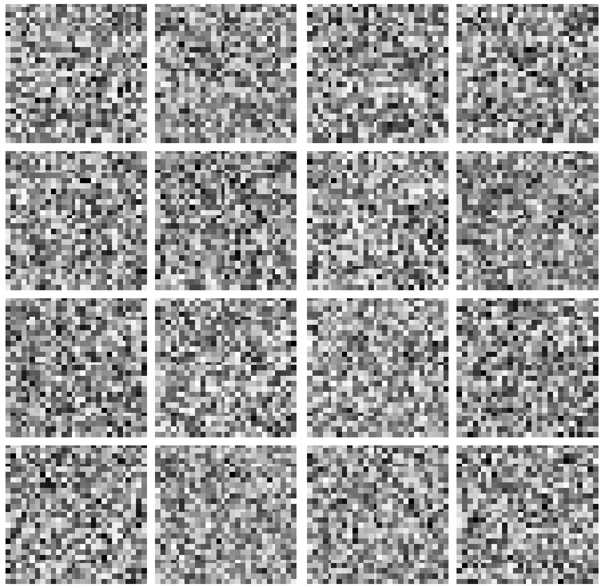
In [28]:
```
class GAN():
    def __init__(self,input_size = 784, random_size = 100):
        self.input_size = input_size
        self.random_size = random_size

    def xavier_init(self,size):
        in_dim = size[0]
        xavier_stddev = 1. / tf.sqrt(in_dim / 2.)
        return tf.random_normal(shape=size, stddev=xavier_stddev)

    def sample_Z(self,m, n):
        return np.random.uniform(-1., 1., size=[m, n])
```

```
def generator(self,z):
    ###IMPLEMENT THE GENERATOR USING THE G_ VARIABLES#####
    G_h1 = tf.nn.relu(tf.matmul(z, self.G_W1) + self.G_b1)
    G_log_prob = tf.matmul(G_h1, self.G_W2) + self.G_b2
    self.G_prob = tf.nn.sigmoid(G_log_prob)
    return self.G_prob


def discriminator(self,x):
    ###IMPLEMENT THE DISCRIMENATOTR USING THE D_ VARIABLES#####
    D_h1 = tf.nn.relu(tf.matmul(x, self.D_W1) + self.D_b1)
    D_logit = tf.matmul(D_h1, self.D_W2) + self.D_b2
    D_prob = tf.nn.sigmoid(D_logit)
    return D_prob, D_logit

def init_training(self):

    self.X = tf.placeholder(tf.float32, shape=[None, self.input_size])

    self.Z = tf.placeholder(tf.float32, shape=[None, self.random_size])

    self.G_W1 = tf.Variable(self.xavier_init([self.random_size, 128]))
    self.G_b1 = tf.Variable(tf.zeros(shape=[128]))

    self.G_W2 = tf.Variable(self.xavier_init([128, self.input_size]))
    self.G_b2 = tf.Variable(tf.zeros(shape=[self.input_size]))


    self.theta_G = [self.G_W1, self.G_W2, self.G_b1, self.G_b2]

    self.D_W1 = tf.Variable(self.xavier_init([self.input_size,128]))
    self.D_b1 = tf.Variable(tf.zeros(shape=[128]))

    self.D_W2 = tf.Variable(self.xavier_init([128, 1]))
    self.D_b2 = tf.Variable(tf.zeros(shape=[1]))

    self.theta_D = [self.D_W1, self.D_W2, self.D_b1, self.D_b2]


    self.G_sample = self.generator(self.Z)
    D_real, D_logit_real = self.discriminator(self.X)
    D_fake, D_logit_fake = self.discriminator(self.G_sample)


# Implement the loss functions for training a GAN
# --------------------
    D_loss_real = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
        logits = D_logit_real, labels = tf.ones_like(D_logit_real)))
```

```
            D_loss_fake = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
                logits = D_logit_fake, labels = tf.zeros_like(D_logit_fake)))
            self.D_loss = D_loss_real + D_loss_fake
            self.G_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
                logits = D_logit_fake, labels = tf.ones_like(D_logit_fake)))

            self.D_solver = tf.train.AdamOptimizer().minimize(self.D_loss, var_list=self.th
            self.G_solver = tf.train.AdamOptimizer().minimize(self.G_loss, var_list=self.th

        def generate_sample(self,num_samples):

            ####GENERATE SAMPLES FROM THE GAN############
            samples = self.sess.run(self.G_sample, feed_dict={self.Z: self.sample_Z(num_sam
            return samples

        def train_model(self,data):

            mb_size = 128
            self.Z_dim = self.random_size

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

            for it in range(100000):
                X_mb, _ = data.train.next_batch(mb_size)

                _, D_loss_curr = self.sess.run([self.D_solver, self.D_loss], feed_dict={sel
                _, G_loss_curr = self.sess.run([self.G_solver, self.G_loss], feed_dict={sel

                if it % 10000 == 0:
                    print('Iter: {}'.format(it))
                    print('D loss: {:.4}'. format(D_loss_curr))
                    print('G_loss: {:.4}'.format(G_loss_curr))

                    samples = self.sess.run(self.G_sample, feed_dict={
                                self.Z: self.sample_Z(16, self.Z_dim)})  # 16*784
                    fig = plot(samples)
                    plt.show()
```

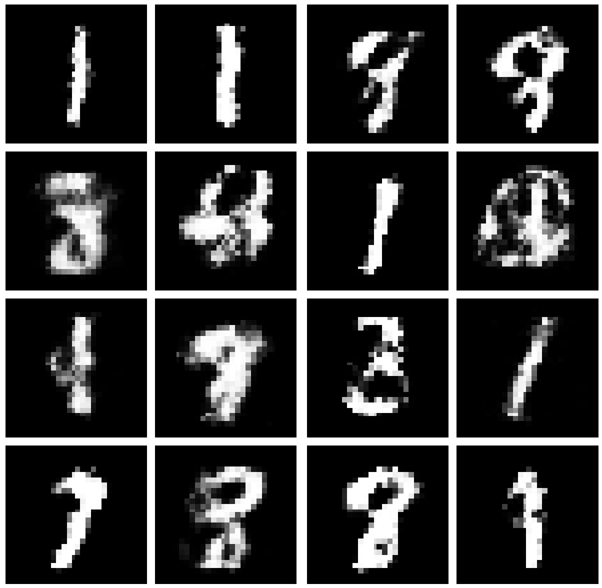
In [29]:
```
#####TRAIN GAN#######
tf.reset_default_graph()
gan_model = GAN()
gan_model.init_training()
gan_model.train_model(mnist)
```

```
Iter: 0
D loss: 1.381
G_loss: 3.037
```

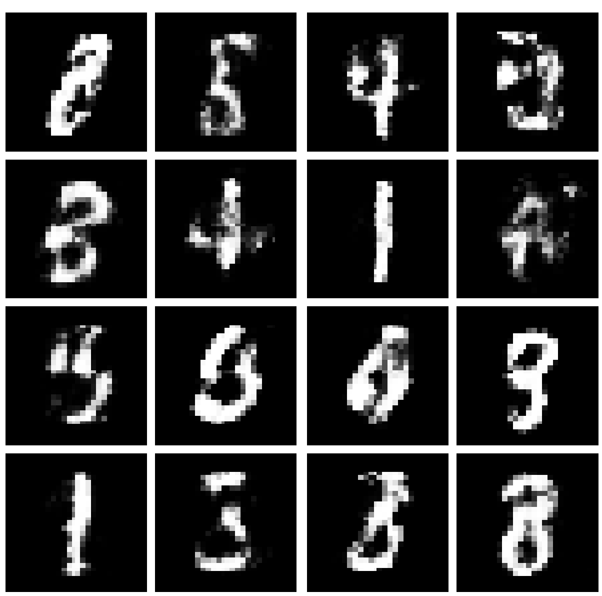


```
Iter: 10000
D loss: 0.6006
G_loss: 3.195
```

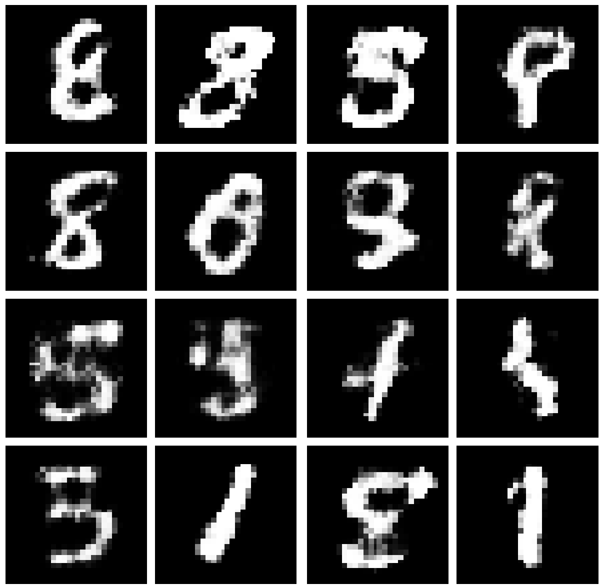
```
Iter: 20000
D loss: 1.086
G_loss: 1.895
```



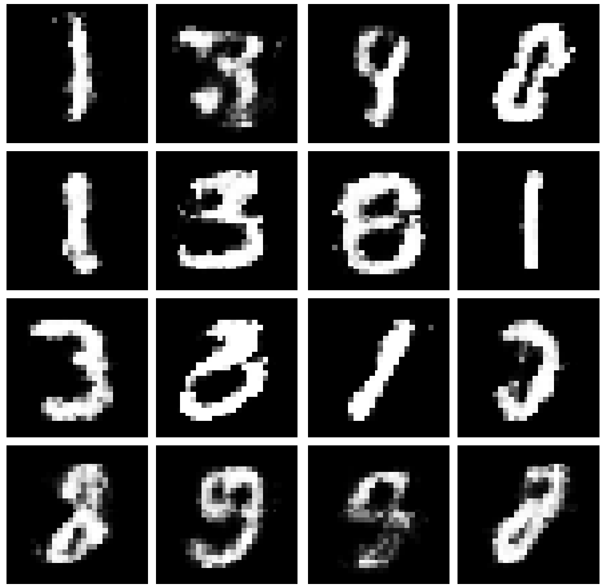
```
Iter: 30000
D loss: 0.7468
G_loss: 1.836
```



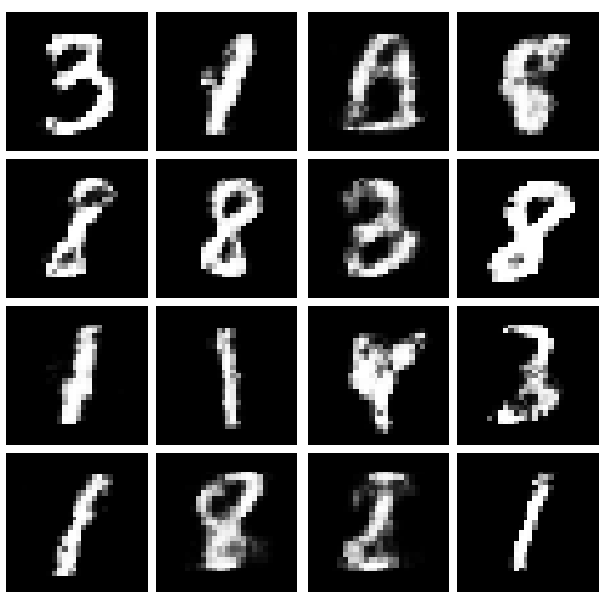
```
Iter: 40000
D loss: 0.8666
G_loss: 2.095
```



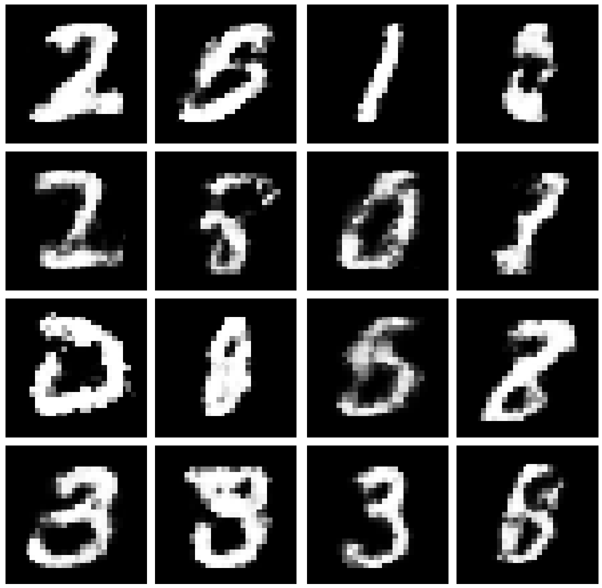
```
Iter: 50000
D loss: 0.5851
G_loss: 2.561
```



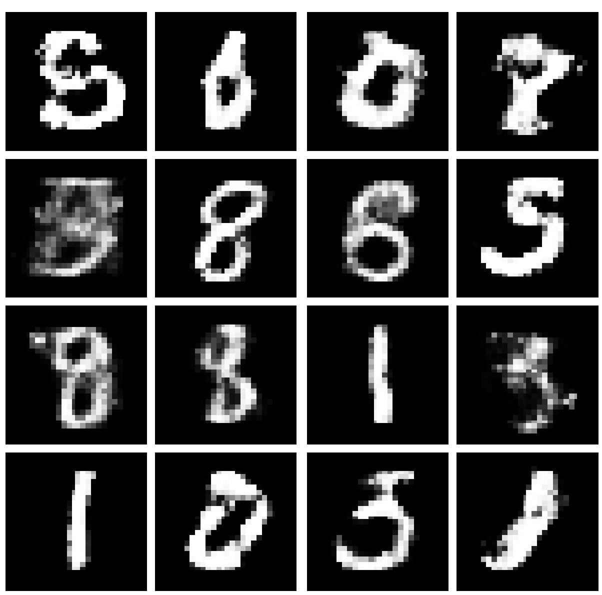
```
Iter: 60000
D loss: 0.6243
G_loss: 2.047
```



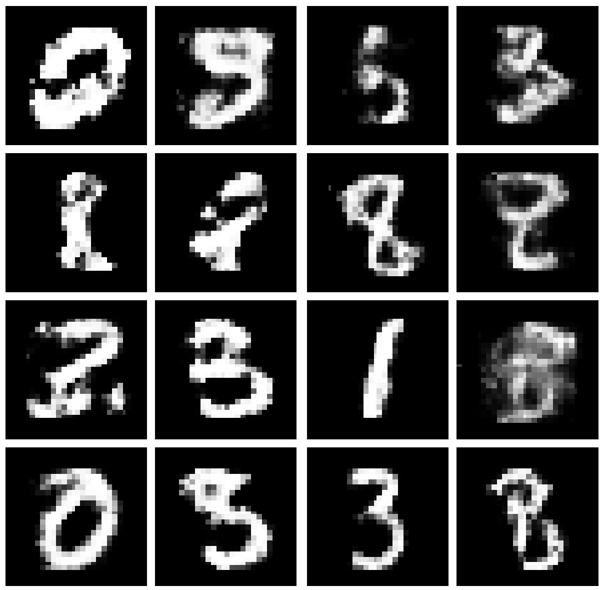
```
Iter: 70000
D loss: 0.6467
G_loss: 2.371
```



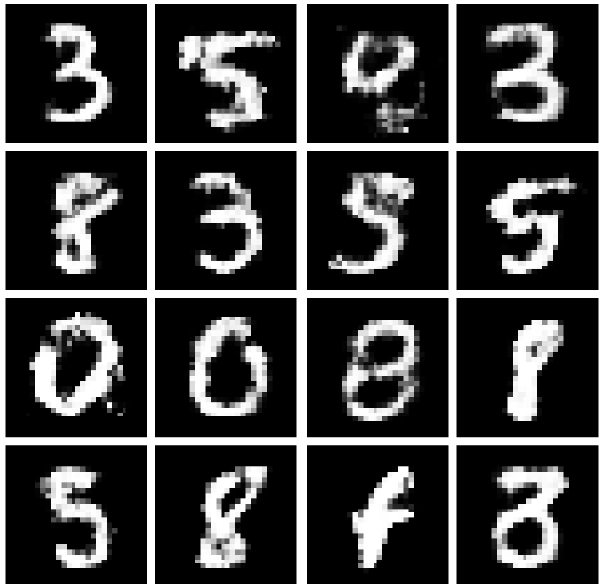
```
Iter: 80000
D loss: 0.5369
G_loss: 2.291
```



```
Iter: 90000
D loss: 0.6721
G_loss: 2.451
```



```
In [30]: samples = gan_model.generate_sample(16)
         fig = plot(samples)
         plt.savefig('gan_mnist.png', bbox_inches='tight')
         plt.show()
```



2.3 (d)

```
In [31]: class Dataset():

             def __init__(self,states):

                 self.train = Trainer(states)


         class Trainer():

             def __init__(self,states):
                 self.states = states

             def next_batch(self,size):

                 data = self.states[np.random.randint(self.states.shape[0], size=size), :]

                 return data, None

In [33]: class GAN():
             def __init__(self,input_size = 784, random_size = 100):
                 self.input_size = input_size
                 self.random_size = random_size

             def xavier_init(self,size):
                 in_dim = size[0]
```

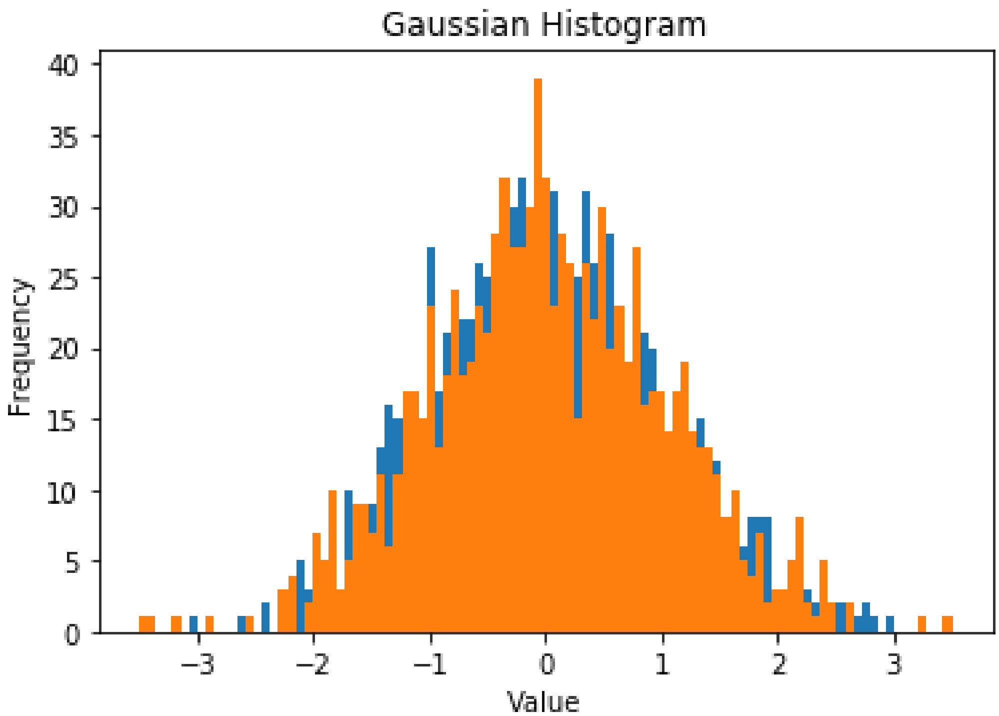
```
        xavier_stddev = 1. / tf.sqrt(in_dim / 2.)
        return tf.random_normal(shape=size, stddev=xavier_stddev)

    def sample_Z(self,m, n):
        return np.random.uniform(-1., 1., size=[m, n])

    def generator(self,z):
        ###IMPLEMENT THE GENERATOR USING THE G_ VARIABLES#####
        G_h1 = tf.nn.relu(tf.matmul(z, self.G_W1) + self.G_b1)
        G_log_prob = tf.matmul(G_h1, self.G_W2) + self.G_b2
        self.G_prob = tf.nn.sigmoid(G_log_prob)
        return self.G_prob


    def discriminator(self,x):
        ###IMPLEMENT THE DISCRIMENATOTR USING THE D_ VARIABLES#####
        D_h1 = tf.nn.relu(tf.matmul(x, self.D_W1) + self.D_b1)
        D_logit = tf.matmul(D_h1, self.D_W2) + self.D_b2
        D_prob = tf.nn.sigmoid(D_logit)
        return D_prob, D_logit

    def init_training(self):

        self.X = tf.placeholder(tf.float32, shape=[None, self.input_size])

        self.Z = tf.placeholder(tf.float32, shape=[None, self.random_size])

        self.G_W1 = tf.Variable(self.xavier_init([self.random_size, 128]))
        self.G_b1 = tf.Variable(tf.zeros(shape=[128]))

        self.G_W2 = tf.Variable(self.xavier_init([128, self.input_size]))
        self.G_b2 = tf.Variable(tf.zeros(shape=[self.input_size]))


        self.theta_G = [self.G_W1, self.G_W2, self.G_b1, self.G_b2]

        self.D_W1 = tf.Variable(self.xavier_init([self.input_size,128]))
        self.D_b1 = tf.Variable(tf.zeros(shape=[128]))

        self.D_W2 = tf.Variable(self.xavier_init([128, 1]))
        self.D_b2 = tf.Variable(tf.zeros(shape=[1]))

        self.theta_D = [self.D_W1, self.D_W2, self.D_b1, self.D_b2]


        self.G_sample = self.generator(self.Z)
        D_real, D_logit_real = self.discriminator(self.X)
        D_fake, D_logit_fake = self.discriminator(self.G_sample)
```

```
        # Implement the loss functions for training a GAN
        # --------------------
            D_loss_real = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
                logits = D_logit_real, labels = tf.ones_like(D_logit_real)))
            D_loss_fake = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
                logits = D_logit_fake, labels = tf.zeros_like(D_logit_fake)))
            self.D_loss = D_loss_real + D_loss_fake
            self.G_loss = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(
                logits = D_logit_fake, labels = tf.ones_like(D_logit_fake)))

            self.D_solver = tf.train.AdamOptimizer().minimize(self.D_loss, var_list=self.th
            self.G_solver = tf.train.AdamOptimizer().minimize(self.G_loss, var_list=self.th

        def generate_sample(self,num_samples):

            ####GENERATE SAMPLES FROM THE GAN############
            samples = self.sess.run(self.G_sample, feed_dict={self.Z: self.sample_Z(num_sam
            return samples

        def train_model(self,data):

            mb_size = 128
            self.Z_dim = self.random_size

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

            for it in range(100000):
                X_mb, _ = data.train.next_batch(mb_size)

                _, D_loss_curr = self.sess.run([self.D_solver, self.D_loss], feed_dict={sel
                _, G_loss_curr = self.sess.run([self.G_solver, self.G_loss], feed_dict={sel

                if it % 10000 == 0:
                    print('Iter: {}'.format(it))
                    print('D loss: {:.4}'. format(D_loss_curr))
                    print('G_loss: {:.4}'.format(G_loss_curr))
```

In [34]:
```
from numpy.random import normal
def plot(ground_truth_samples,generated_samples):
    #IPython.embed()

    bins = np.linspace(-3.5, 3.5, 100)
    plt.hist(ground_truth_samples,bins)
    plt.hist(generated_samples,bins)
```

```
    plt.title("Gaussian Histogram")
    plt.xlabel("Value")
    plt.ylabel("Frequency")

    print('got here')

    fig = plt.gcf()

    return fig

N_SAMPLES = 1000

train_data = normal(size=(1,N_SAMPLES))
train_data = train_data.T



####TRAIN KDE####v##
kde_model = KDE(use_pca=False)
kde_model.train_model(train_data)
samples = kde_model.generate_sample(N_SAMPLES)

fig = plot(train_data,samples)
plt.savefig('gaussian_kde.png', bbox_inches='tight')
plt.show()


#####TRAIN GAN#######
gan_model = GAN(input_size = 1, random_size = 1)
gan_model.init_training()

#Create dataset
train_data_m = Dataset(train_data)
gan_model.train_model(train_data_m)
samples = gan_model.generate_sample(N_SAMPLES)


fig = plot(train_data,samples)
plt.savefig('gaussian_gan.png', bbox_inches='tight')
plt.show()
```

```
got here
```




```
Iter: 0
D loss: 1.647
G_loss: 0.3977
Iter: 10000
D loss: 0.7344
G_loss: 1.336
Iter: 20000
D loss: 0.7166
G_loss: 1.38
Iter: 30000
D loss: 0.6831
G_loss: 1.461
Iter: 40000
D loss: 0.7104
G_loss: 1.413
Iter: 50000
D loss: 0.8907
G_loss: 1.535
Iter: 60000
D loss: 0.7847
G_loss: 1.346
Iter: 70000
D loss: 0.6666
G_loss: 1.47
```

```
Iter: 80000
D loss: 0.7453
G_loss: 1.388
Iter: 90000
D loss: 0.6945
G_loss: 1.4
got here
```