Qiaojun Feng A53235331

1. Multidimensional Scaling:

```
import numpy as np
import random
import scipy
import pickle
import os
import matplotlib.pyplot as plt
from sklearn import manifold, datasets, decomposition, ensemble, random_projection
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data

def plot_embedding_2d(X, Y, title=None):
    color_space = ['k','grey','r','y','g','c','b','m','orange','navy']
    fig = plt.figure(figsize=[8,8])
    ax = fig.add_subplot(1, 1, 1)
    for i in range(X.shape[0]):
        ax.text(X[i, 0], X[i, 1],str(Y[i]),
                 color=color_space[Y[i]],
                 fontdict={'weight': 'bold', 'size': 9})
    if title is not None:
        plt.title(title)
    x_range = np.max(np.abs(X))
    plt.xlim((-x_range, x_range))
    plt.ylim((-x_range, x_range))
    plt.savefig(title+".png")
    plt.show()

# generate the data
if os.path.isfile('mnist_X.pickle') and os.path.isfile('mnist_Y.pickle'):
    mnist_X = pickle.load(open('mnist_X.pickle', 'rb'))
    mnist_Y = pickle.load(open('mnist_Y.pickle', 'rb'))
else:
    mnist = input_data.read_data_sets("../MNIST_data/", reshape=True)
    # notice that here we don't shuffle the data so the samples with same label are together
    sample_per_label = 1000
    random.seed(291)
    index_sampling = []
    for target_label in range(10):
        target_label_position = np.where(mnist.train.labels==target_label)[0]
        target_label_sampling = random.sample(list(target_label_position), sample_per_label)
        index_sampling.extend(target_label_sampling)
    mnist_X = mnist.train.images[index_sampling,:]
    mnist_Y = mnist.train.labels[index_sampling]
```

```
if os.path.isfile('mnist_Dist.pickle'):
    mnist_Dist = pickle.load(open('mnist_Dist.pickle', 'rb'))
else:
    mnist_Dist = scipy.spatial.distance.cdist(mnist_X,mnist_X)

# build the graph
# generate the embedding space point automatically and
# calculate the difference
N = 10000
D = tf.placeholder(tf.float32,[N,N])
D_vec = tf.reshape(D,[-1,1])
x = tf.Variable(tf.truncated_normal([N, 2],mean=0.0,stddev=0.1), dtype = tf.float32)
xsq = tf.reduce_sum(x*x, 1)
xsq_vec = tf.reshape(xsq,[-1,1])
dsq = xsq_vec - 2*tf.matmul(x,tf.transpose(x)) + tf.transpose(xsq_vec)
d_vec = tf.sqrt(tf.maximum(dsq,1e-20))
d_vec = tf.reshape(d_vec,[-1,1])
model_loss = tf.reduce_sum(tf.square(D_vec-d_vec))/2
avg_dist = model_loss/(N*(N-1)/2)

train = tf.train.GradientDescentOptimizer(1e-5).minimize(model_loss)
# train = tf.train.AdamOptimizer(1e-3).minimize(model_loss)

# begin the training
iter_times = 1000
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    x_init,d_init,model_loss_init = sess.run([x,d_vec,model_loss],{D: mnist_Dist[:N,:N]})
    for iters in range(iter_times):
        sess.run(train, {D: mnist_Dist[:N,:N]})
        loss,dist = sess.run([model_loss,avg_dist],{D: mnist_Dist[:N,:N]})
        if iters%10 == 0:
            print('iter: ', iters, ' loss: ', loss, " avg_distance: ", dist)
    x_final = sess.run(x,{D: mnist_Dist[:N,:N]})

# visualization
plot_embedding_2d(x_init, mnist_Y, 'initial')
plot_embedding_2d(x_final, mnist_Y, 'final')
```

2. Farthest Point Sampling

```
### use PyMesh to read data and store by pickle
# Author:   Qiaojun Feng
# Date:     02/10/2018

# read in the vertices and the faces from .obj files
import pymesh
import pickle
print('read teapot')
teapot_mesh = pymesh.load_mesh("teapot.obj")
print('teapot imported')
print('read violin')
violin_mesh = pymesh.load_mesh("violin_case.obj")
print('violin imported')

print('teapot num_vertices: ', teapot_mesh.num_vertices, \
    ' teapot num_faces: ', teapot_mesh.num_faces, \
    ' teapot num_voxels: ', teapot_mesh.num_voxels)
teapot_vertices = teapot_mesh.vertices
teapot_faces = teapot_mesh.faces
teapot_mesh.enable_connectivity()
print(teapot_mesh.get_vertex_adjacent_vertices(0))

print('violin num_vertices: ', violin_mesh.num_vertices, \
    ' violin num_faces: ', violin_mesh.num_faces, \
    ' violin num_voxels: ', violin_mesh.num_voxels)
violin_case_vertices = violin_mesh.vertices
violin_case_faces = violin_mesh.faces


output = open('teapot_vertices.pickle', 'wb')
pickle.dump(teapot_vertices, output)
output.close()
output = open('teapot_faces.pickle', 'wb')
pickle.dump(teapot_faces, output)
output.close()
output = open('violin_case_vertices.pickle', 'wb')
pickle.dump(violin_case_vertices, output)
output.close()
output = open('violin_case_faces.pickle', 'wb')
pickle.dump(violin_case_faces, output)
output.close()
```

```
### main function
# Author:   Qiaojun Feng
# Date:             02/11/2018

import numpy as np
import pickle
import os
import random
import scipy
import scipy.spatial
import heapq
import pandas as pd
from pyntcloud import PyntCloud
from Q2_func import *

# change this to change the read object and a series of corresponding files
obj = 'teapot'
# obj = 'violin_case'

# read the vertices and faces data
# have been stored in the .pkl form
# calculate the area of all the triangles in the mesh
vertices = pickle.load(open(obj+'_vertices.pkl', 'rb'))
faces = pickle.load(open(obj+'_faces.pkl', 'rb'))
if os.path.isfile(obj+'_area.pkl'):
   area = pickle.load(open(obj+'_area.pkl', 'rb'))
else:
   area = triangle_faces_area(vertices,faces)


# transform to weight and assign number of sampling point on each faces
# N is total sampling points' number: point set P
if os.path.isfile('P_'+obj+'.pkl'):
    P = pickle.load(open('P_'+obj+'.pkl', 'rb'))
else:
    N = 11000
    sample_number = N*np.round(area/np.sum(area),np.log10(N).astype(int))
    sample_number = sample_number.astype(int)
    N_sample = np.sum(sample_number).astype(int)

    P = np.zeros([N_sample,3])
    index_sample = 0
    for index_face in range(faces.shape[0]):
```

```
        vertice_coordinate = vertices[faces[index_face]]
        for index_sample_mesh in range(sample_number[index_face]):
            r1 = np.random.uniform()
            r2 = np.random.uniform()
            P[index_sample] = (1-np.sqrt(r1))*vertice_coordinate[0] + \
                              np.sqrt(r1)*(1-r2)*vertice_coordinate[1] + \
                              np.sqrt(r1)*r2*vertice_coordinate[2]
            index_sample = index_sample + 1

# Farthest Point Sampling
# Given a point set and a distance matrix
# Return a sampled point set
# dist_file = dist_'+obj'_all.pkl
# d = pickle.load(open(dist_file, 'rb'))
d = scipy.spatial.distance.cdist(P,P)
n = 1000
N = P.shape[0]
S_index = random.sample(range(N),1)
for iters in range(n-1):
    min_distance = np.min(d[S_index,],0)
    index_max = np.argmax(min_distance)
    S_index.extend([index_max])
S = P[S_index]

# for display visualization
# only in jupyter notebook
points = pd.DataFrame(S, columns=['x', 'y', 'z'])
cloud = PyntCloud(points)
cloud.plot(lines=[], line_color=[])

### some dependent function
# function to compute the area of an triangle
# input is a 3*3 matrix, with each row a vertices in 3d space
def triangles_area(vertices):
    v1 = vertices[1] - vertices[0]
    v2 = vertices[2] - vertices[0]
    cross = np.cross(v1,v2)
    area = np.linalg.norm(cross)/2
    return area

# function to compute the area of an triangle
# input is a 3*3 matrix, with each row a vertices in 3d space
# using Heron's formula
def triangles_area_Heron(vertices):
```

```
    a = np.linalg.norm(vertices[2]-vertices[0])
    b=np.linalg.norm(vertices[2]-vertices[1])
    c=np.linalg.norm(vertices[1]-vertices[0])
    p = (a+b+c)/2
    area = np.sqrt(p*(p-a)*(p-b)*(p-c))
    return area

# function that return all the triangle faces' area
def triangle_faces_area(vertices,faces):
    n_faces = faces.shape[0]
    area = np.zeros(n_faces)
    for index_face in range(n_faces):
        area[index_face] = triangles_area_Heron(vertices[faces[index_face]])
    return area

# Given a point cloud with n points, using k nearest-neighbors algorithm to build a graph on
it
# return the distance matrix(n*n) of the graph. Inf means not edge between 2 points
# 1. make sure each edge is bilateral
# 2. make sure the full connectivity, i.e. any point can reach any point
# Because of the request above, degree of each point may vary
# if return nan, means that some points are not connected to the others
# maybe you may want to increase k

def knn_distance_init(points,k):
    k = 5
    N = points.shape[0]
    # calculate the distance matrix between each pair of points
    Dist = scipy.spatial.distance.cdist(points,points)
    # initialize a distance matrix and set all the elements to zero
    D = np.zeros(Dist.shape)
    D[:] = np.inf
    # for each point, find its (k+1) values that are the minimal(because including 0)
    # assign them to matrix D
    for index_point in range(N):
        k_smallest_index = np.argpartition(Dist[index_point], k+1)[:k+1]
        D[index_point][k_smallest_index] = Dist[index_point][k_smallest_index]
    # 1. make sure each edge is bilateral
    # by comparing D[i][j] and D[j][i]
    # let D[i][j]=D[j][i] = min(D[i][j],D[j][i])
    for index_i in range(N):
        for index_j in range(index_i+1,N):
            if D[index_i][index_j] != D[index_j][index_i]:
                D_min = np.min([D[index_i][index_j],D[index_j][index_i]])
```

```
                D[index_i][index_j] = D_min
                D[index_j][index_i] = D_min
    # 2. make sure the full connectivity
    # by expanding from one point till the end
    check_connection = np.zeros(N)
    check_connection[0] = 1
    open_list = [0]
    for iters in range(N-1):
        if len(open_list) == 0:
            return np.array(np.nan)
        else:
            open_node = open_list[0]
        open_list.remove(open_node)
        neighbor_nodes = list(np.where(np.isfinite(D[open_node]))[0])
        neighbor_nodes.remove(open_node)
        # only keep the neighbor that hasn't been visited before
        neighbor_keep = []
        for index_neighbor in range(len(neighbor_nodes)):
            if check_connection[neighbor_nodes[index_neighbor]] != 1:
                neighbor_keep.extend([index_neighbor])
        neighbor_nodes_keep = [neighbor_nodes[i] for i in neighbor_keep]
        open_list.extend(neighbor_nodes_keep)
        check_connection[neighbor_nodes_keep] = 1
        if np.sum(check_connection) == N:
            return D
    if np.sum(check_connection) == N:
        return D
    else:
        return np.array(np.nan)

# Given a initial distance matrix D with inf (meaning no connection)
# Return a new distance matrix d by Dijkstra algorithm
# Compute the distance by adding the edge length given in D
# Just too slow

def Dijkstra_distance(D):
    N = D.shape[0]
    d = np.copy(D)
    # for each point i explore the distance
    for index_point in range(N):
        print(index_point)
        close_list = np.zeros(N)
        close_list[index_point] = 1
        open_list = list(np.where(np.isfinite(d[index_point]))[0])
```

```
        open_list_distance = list(d[index_point][open_list])
        for iters in range(N-1):
            # find the node with smallest distance
            # name it open_node
            # remove from open_list, add to close_list
            # print(open_list)
            nearest_index = np.argmin(np.array(open_list_distance))
            open_node = open_list[nearest_index]
            open_list.remove(open_node)
            open_list_distance.remove(open_list_distance[nearest_index])
            close_list[open_node] = 1
            # find the neighbors of open_list
            neighbor_nodes = list(np.where(np.isfinite(d[open_node]))[0])
            neighbor_nodes.remove(open_node)
            # only keep the neighbors that are not in close_list
            # actually we should also discard those in open_list
            neighbor_keep = []
            for index_neighbor in range(len(neighbor_nodes)):
                if close_list[neighbor_nodes[index_neighbor]] != 1:
                    neighbor_keep.extend([index_neighbor])
            neighbor_nodes_keep = [neighbor_nodes[i] for i in neighbor_keep]
            for index_neighbor in neighbor_nodes_keep:
                if d[index_point][open_node]+d[open_node][index_neighbor] <
d[index_point][index_neighbor]:
                    d[index_point][index_neighbor] =
d[index_point][open_node]+d[open_node][index_neighbor]
                    open_list.extend([index_neighbor])
                    open_list_distance.extend([d[index_point][index_neighbor]])
            if np.sum(close_list) == N:
                continue
        # connect point[index_point] with all the other points
        d[:,index_point] = d[index_point,:]

    return d

# Floyd algorithm to build distance matrix
def Floyd_distance(D):
    N = D.shape[0]
    d = np.copy(D)
    for index_middle in range(N):
        for index_i in range(N):
            d[index_i] =
np.min(np.vstack([d[index_i],d[index_i][index_middle]+d[index_middle]]),0)
    return d
```

3. Earth Mover's Distance

```
### Part 1: Hungarian Algorithm (mostly from scikit-learn package)

import numpy as np
import scipy
import scipy.spatial
import scipy.optimize
import pickle
import random


"""
Solve the unique lowest-cost assignment problem using the
Hungarian algorithm (also known as Munkres algorithm).

"""
# Based on original code by Brain Clapper, adapted to NumPy by Gael Varoquaux.
# Heavily refactored by Lars Buitinck.
#
# TODO: a version of this algorithm has been incorporated in SciPy; use that
# when SciPy 0.17 is released.

# Copyright (c) 2008 Brian M. Clapper <bmc@clapper.org>, Gael Varoquaux
# Author: Brian M. Clapper, Gael Varoquaux
# LICENSE: BSD


def linear_assignment(X):
  indices = _hungarian(X)
  return indices


class _HungarianState(object):
  """State of one execution of the Hungarian algorithm.

  Parameters
  ----------
  cost_matrix : 2D matrix
    The cost matrix. Does not have to be square.
  """

  def __init__(self, cost_matrix):
    cost_matrix = np.atleast_2d(cost_matrix)
```

```
        self.C = cost_matrix.copy()
        n, m = self.C.shape
        self.row_uncovered = np.ones(n, dtype=np.bool)
        self.col_uncovered = np.ones(m, dtype=np.bool)
        self.Z0_r = 0
        self.Z0_c = 0
        self.path = np.zeros((n + m, 2), dtype=int)
        self.marked = np.zeros((n, m), dtype=int)

    def _find_prime_in_row(self, row):
        """
        Find the first prime element in the specified row. Returns
        the column index, or -1 if no starred element was found.
        """
        col = np.argmax(self.marked[row] == 2)
        if self.marked[row, col] != 2:
            col = -1
        return col

    def _clear_covers(self):
        """Clear all covered matrix cells"""
        self.row_uncovered[:] = True
        self.col_uncovered[:] = True


def _hungarian(cost_matrix):
    """The Hungarian algorithm.

    Calculate the Munkres solution to the classical assignment problem and
    return the indices for the lowest-cost pairings.

    Parameters
    ----------
    cost_matrix : 2D matrix
        The cost matrix. Does not have to be square.

    Returns
    -------
    indices : 2D array of indices
        The pairs of (row, col) indices in the original array giving
        the original ordering.
    """
    state = _HungarianState(cost_matrix)
```

```
    # No need to bother with assignments if one of the dimensions
    # of the cost matrix is zero-length.
    step = None if 0 in cost_matrix.shape else _step1

    while step is not None:
        step = step(state)

    # Look for the starred columns
    row_ind = np.where(state.marked == 1)[0]
    col_ind = np.where(state.marked == 1)[1]
    # results = np.array(np.where(state.marked == 1)).T

    return row_ind, col_ind


# Individual steps of the algorithm follow, as a state machine: they return
# the next step to be taken (function to be called), if any.

def _step1(state):
    """Steps 1 and 2 in the Wikipedia page."""

    # Step1: For each row of the matrix, find the smallest element and
    # subtract it from every element in its row.
    state.C -= state.C.min(axis=1)[:, np.newaxis]
    # Step2: Find a zero (Z) in the resulting matrix. If there is no
    # starred zero in its row or column, star Z. Repeat for each element
    # in the matrix.
    for i, j in zip(*np.where(state.C == 0)):
        if state.col_uncovered[j] and state.row_uncovered[i]:
            state.marked[i, j] = 1
            state.col_uncovered[j] = False
            state.row_uncovered[i] = False
    state._clear_covers()
    return _step3


def _step3(state):
    """
    Cover each column containing a starred zero. If n columns are covered,
    the starred zeros describe a complete set of unique assignments.
    In this case, Go to DONE, otherwise, Go to Step 4.
    """
    marked = (state.marked == 1)
```

```
    state.col_uncovered[np.any(marked, axis=0)] = False
    if marked.sum() < state.C.shape[0]:
        return _step4



def _step4(state):
    """
    Find a noncovered zero and prime it. If there is no starred zero
    in the row containing this primed zero, Go to Step 5. Otherwise,
    cover this row and uncover the column containing the starred
    zero. Continue in this manner until there are no uncovered zeros
    left. Save the smallest uncovered value and Go to Step 6.
    """
    # We convert to int as numpy operations are faster on int
    C = (state.C == 0).astype(np.int)
    covered_C = C * state.row_uncovered[:, np.newaxis]
    covered_C *= state.col_uncovered.astype(dtype=np.int, copy=False)
    n = state.C.shape[0]
    m = state.C.shape[1]
    while True:
        # Find an uncovered zero
        row, col = np.unravel_index(np.argmax(covered_C), (n, m))
        if covered_C[row, col] == 0:
            return _step6
        else:
            state.marked[row, col] = 2
            # Find the first starred element in the row
            star_col = np.argmax(state.marked[row] == 1)
            if not state.marked[row, star_col] == 1:
                # Could not find one
                state.Z0_r = row
                state.Z0_c = col
                return _step5
            else:
                col = star_col
                state.row_uncovered[row] = False
                state.col_uncovered[col] = True
                covered_C[:, col] = C[:, col] * (
                    state.row_uncovered.astype(dtype=np.int, copy=False))
                covered_C[row] = 0


def _step5(state):
```

```
"""
Construct a series of alternating primed and starred zeros as follows.
Let Z0 represent the uncovered primed zero found in Step 4.
Let Z1 denote the starred zero in the column of Z0 (if any).
Let Z2 denote the primed zero in the row of Z1 (there will always be one).
Continue until the series terminates at a primed zero that has no starred
zero in its column. Unstar each starred zero of the series, star each
primed zero of the series, erase all primes and uncover every line in the
matrix. Return to Step 3
"""
count = 0
path = state.path
path[count, 0] = state.Z0_r
path[count, 1] = state.Z0_c

while True:
   # Find the first starred element in the col defined by
   # the path.
   row = np.argmax(state.marked[:, path[count, 1]] == 1)
   if not state.marked[row, path[count, 1]] == 1:
      # Could not find one
      break
   else:
      count += 1
      path[count, 0] = row
      path[count, 1] = path[count - 1, 1]

   # Find the first prime element in the row defined by the
   # first path step
   col = np.argmax(state.marked[path[count, 0]] == 2)
   if state.marked[row, col] != 2:
      col = -1
   count += 1
   path[count, 0] = path[count - 1, 0]
   path[count, 1] = col

# Convert paths
for i in range(count + 1):
   if state.marked[path[i, 0], path[i, 1]] == 1:
      state.marked[path[i, 0], path[i, 1]] = 0
   else:
      state.marked[path[i, 0], path[i, 1]] = 1

state._clear_covers()
```

```
    # Erase all prime markings
    state.marked[state.marked == 2] = 0
    return _step3


def _step6(state):
    """
    Add the value found in Step 4 to every element of each covered row,
    and subtract it from every element of each uncovered column.
    Return to Step 4 without altering any stars, primes, or covered lines.
    """
    # the smallest uncovered value in the matrix
    if np.any(state.row_uncovered) and np.any(state.col_uncovered):
        minval = np.min(state.C[state.row_uncovered], axis=0)
        minval = np.min(minval[state.col_uncovered])
        state.C[np.logical_not(state.row_uncovered)] += minval
        state.C[:, state.col_uncovered] -= minval
    return _step4

X1 = pickle.load(open('X1.pkl', 'rb'))
X2 = pickle.load(open('X2.pkl', 'rb'))
D = scipy.spatial.distance.cdist(X1,X2)


[row_a,col_a] = linear_assignment(D)
cost = np.sum(D[row_a,col_a])
print(cost)

row_ind, col_ind = scipy.optimize.linear_sum_assignment(D)
cost = np.sum(D[row_ind,col_ind])
print(cost)



### Part 2: tensorflow optimization

import tf_emddistance
import numpy as np
import tensorflow as tf
import random
import pandas as pd
from pyntcloud import PyntCloud

# generate a series of point clouds sampled from a series of circle
```

```
def generate_circle_points(n_clouds,n_points,r_min,r_max):
    S = np.zeros([n_clouds,n_points,3])
    np.random.seed(291)
    for index_cloud in range(n_clouds):
        r = np.random.uniform(r_min,r_max)
        for index_point in range(n_points):
            theta = np.random.uniform(0,2*np.pi)
            S[index_cloud][index_point][0:2] = [r*np.cos(theta),r*np.sin(theta)]
    return S

# define the parameter for generating circle point set
n_clouds = 100
n_points = 500
r_min = 1
r_max = 10

# define the optimization problem
S = tf.placeholder(tf.float32,[None,n_points,3])
x = tf.Variable(tf.truncated_normal([1,n_points,3],mean=0.0,stddev=0.1), dtype = tf.float32)
X = tf.tile(x,[tf.shape(S)[0],1,1])
dist,_,_ = tf_emddistance.emd_distance(S,X)
model_loss = tf.reduce_mean(dist)*10000
avg_r = tf.reduce_mean(tf.norm(x,axis = 2))
train = tf.train.GradientDescentOptimizer(1e-3).minimize(model_loss)

# start to optimize
iter_times = 300
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    S_circle = generate_circle_points(n_clouds,n_points,r_min,r_max)
    for iters in range(iter_times):
        sess.run(train, {S: S_circle})
        loss,r = sess.run([model_loss,avg_r],{S: S_circle})
        if iters%10 == 0:
            print('iter: ', iters, '\tloss: ', loss, '\tavg_r: ', r)

    x_final = sess.run(x,{S: S_circle})
    x_final = np.reshape(x_final,[n_points,3])


# for display visualization
# only in jupyter notebook
red_color = np.tile(np.array([255,0,0]),[n_points,1])
```

```
grey_color = np.tile(np.array([255,255,255]),[n_clouds*n_points,1])
colors = (np.vstack([red_color,grey_color])).astype(np.uint8)
P = np.vstack([x_final,np.reshape(S_circle,[n_clouds*n_points,3])])
points = pd.DataFrame(P, columns=['x', 'y', 'z'])
points[['red', 'blue', 'green']] = pd.DataFrame(colors, index=points.index)
cloud = PyntCloud(points)
cloud.plot(point_size = 0.01,lines=[], line_color=[])
```

4. Denoising Autoencoder

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import random

# read the data
# using 60000 images for training
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('../MNIST_data', validation_size=0)

# the network architecture
inputs_ = tf.placeholder(tf.float32, (None, 28, 28, 1), name='inputs')
targets_ = tf.placeholder(tf.float32, (None, 28, 28, 1), name='targets')

### Encoder
## Encoder has 3 convolution+maxpooling layer combination
# 28x28x1
conv1 = tf.layers.conv2d(inputs_, 32, (3,3), padding='same', activation=tf.nn.relu)
maxpool1 = tf.layers.max_pooling2d(conv1, (2,2), (2,2), padding='same')
# 14x14x32
conv2 = tf.layers.conv2d(maxpool1, 32, (3,3), padding='same', activation=tf.nn.relu)
maxpool2 = tf.layers.max_pooling2d(conv2, (2,2), (2,2), padding='same')
# 7x7x32
conv3 = tf.layers.conv2d(maxpool2, 16, (3,3), padding='same', activation=tf.nn.relu)
encoded = tf.layers.max_pooling2d(conv3, (2,2), (2,2), padding='same')
# 4x4x16
### Decoder
## Decoder has 3 unpooling+convolution layer combination
## Implement unpooling by nearest-neighbor resize
# 4x4x16
upsample1 = tf.image.resize_nearest_neighbor(encoded, (7,7))
conv4 = tf.layers.conv2d(upsample1, 16, (3,3), padding='same', activation=tf.nn.relu)
# 7x7x16
upsample2 = tf.image.resize_nearest_neighbor(conv4, (14,14))
conv5 = tf.layers.conv2d(upsample2, 32, (3,3), padding='same', activation=tf.nn.relu)
# 14x14x32
upsample3 = tf.image.resize_nearest_neighbor(conv5, (28,28))
conv6 = tf.layers.conv2d(upsample3, 32, (3,3), padding='same', activation=tf.nn.relu)
# 28x28x32
logits = tf.layers.conv2d(conv6, 1, (3,3), padding='same', activation=None)
# 28x28x1
```

```
# finally include a sigmoid for display but not for calculating loss
# to keep the symmetric of encoder & decoder
decoded = tf.nn.sigmoid(logits, name='decoded')
loss = tf.nn.sigmoid_cross_entropy_with_logits(labels=targets_, logits=logits)
cost = tf.reduce_mean(loss)
opt = tf.train.AdamOptimizer(0.001).minimize(cost)

# start training
epochs = 10
batch_size = 200
# noise_factor
noise_factor = 0.5
sess = tf.Session()
sess.run(tf.global_variables_initializer())
for e in range(epochs):
    for ii in range(mnist.train.num_examples//batch_size):
        batch = mnist.train.next_batch(batch_size)
        imgs = batch[0].reshape((-1, 28, 28, 1))
        # make sure the noisy_image is still in [0,1]
        noisy_imgs = imgs + noise_factor * np.random.randn(*imgs.shape)
        noisy_imgs = np.clip(noisy_imgs, 0., 1.)
        batch_cost, _ = sess.run([cost, opt], feed_dict={inputs_: noisy_imgs,targets_: 
imgs})
    print("Epoch: {}/{}...".format(e+1, epochs),"Training loss: {:.4f}".format(batch_cost))
# don't closed the session after training
# because we need to test

fig, axes = plt.subplots(nrows=3, ncols=10, sharex=True, sharey=True, figsize=(20,4))
test_index = [random.sample(list(np.where(mnist.test.labels==i)[0]),1)[0] for i in range(10)]
in_imgs = mnist.test.images[test_index]
noisy_imgs = in_imgs + 1*noise_factor * np.random.randn(*in_imgs.shape)
noisy_imgs = np.clip(noisy_imgs, 0., 1.)

reconstructed = sess.run(decoded, feed_dict={inputs_: noisy_imgs.reshape((10, 28, 28, 1))})

for images, row in zip([in_imgs, noisy_imgs, reconstructed], axes):
    for img, ax in zip(images, row):
        ax.imshow(img.reshape((28, 28)), cmap='Greys_r')
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)

fig.tight_layout(pad=0.1)
plt.savefig('denoising.png')
plt.show()
```