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')

1. 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

1. 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=[])

1. 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()