CSE 291 I00: Machine Learning for 3D Data Homework 2

Qiaojun Feng
A53235331
Department of Electrical and Computer Engineering
University of California, San Diego
qjfeng@ucsd.edu

1. Multidimensional Scaling

For this problem my codes combines a lot of codes share from the reference resource [2].

First we design a way to calculate the distance matrix among various of multidimensional objects. Suppose we have N objects with d features, we can represent them as a $N \times d$ matrix X. And we call the i-th row of it x_i(notice here we regard it as a row vector). Now we want to get an $N \times N$ matrix D, in which D_{ij} represent the euclidean distance between x_i and x_j. By definition:

$$D_{ij} = \|x_i - x_j\|_2 = \sqrt{(x_i - x_j)(x_i - x_j)^T}$$

We can rewrite the square of the euclidean distance as:

$$D_{ij}{}^2 = x_i x_i^T + x_j x_j^T - 2x_i x_j$$

We can easily generate this to matrix form and implement this to calculate the distance matrix.

Then we form the problem and write down the quantity we want to minimize. We want to embed it to 2-d space so we first generated a $N \times 2$ matrix. Here we use truncated normal distribution with standard deviation = 0.1(for tensorflow it will be truncated at twice the stddev). The calculation of loss is quite straight-forward, the square of the difference between 2 distance matrix at different degree of space. One thing we find that should take care of is the calculation of square root. Since we have found a way to compute the square of euclidean distance, we need to get its square root to calculate the loss. Since we are using gradient, then we have:

$$y = \sqrt{x} \qquad \frac{\partial y}{\partial x} = \frac{1}{2\sqrt{x}}$$

If we somehow get a relative small x, then its gradient may be easily explode to a relative large value, dropping out NAN error. Instead, we want to have a minimum threshold on x:

$$x = \max(x, \epsilon) \qquad \frac{\partial y}{\partial x} = \min(\frac{1}{2\sqrt{x}}, \frac{1}{2\sqrt{\epsilon}})$$

so that the gradient will never exceed some specific value.

The visualization is shown at Fig. 1. We can find some classes of digits automatically cluster together, which exactly make sense. Intuitively intra-class distance should be smaller than inter-class distance. After embedding to 2-d space, we should keep this property. That's why we get this clustering-like result.

2. Farthest Point Sampling

Here we simply play with some point cloud data.

First we use PyMesh[3] to load the data and store vertices and faces information. Then we use the method in [5] to sample points from each mesh(face) based on its area. Here we sample more than 10000 points. For the dense sampled points, we calculate its distance matrix. The easy way is directly use euclidean distance between each pair of points, like building a graph that between each pair of points there is a direct edge. Another much more computational consuming method is to calculate approximate on-mesh distance. Here we use k-nearest-neighbor to connect each point with some of its nearest neighboring points to build a graph. For this graph we want to make it bilateral and fully connected. After that, we use Dijkstra algorithm or Floyd algorithm to calculate the shortest path.

After getting the distance matrix, farthest point sampling is easy to implement. Need to notice that we usually define the distance between one point to a point cloud by the shortest distance between this point to any point in the point cloud, but not the average distance.

Fig. 2 completely shows our result, including the original vertices(we ignore the faces structure), the dense sampled points and the sparse sampled points. Surprisingly, we find that points sampled using euclidean distance is better than using approximated on-mesh distance. It may show that our method to approximate on-mesh distance is both inefficient and inaccurate. Also we can't ignore that euclidean distance is a good metric for farthest point sampling, when our object is quite simple and convex.

3. Earth Movers Distance

For the implementation of Hungarian algorithm, we largely reuse the implementation of linear_assignment function of package scikitlearn[6]. It is not trivial to implement Hungarian algorithm from scratch. Below we try to introduce the implementation method in details.

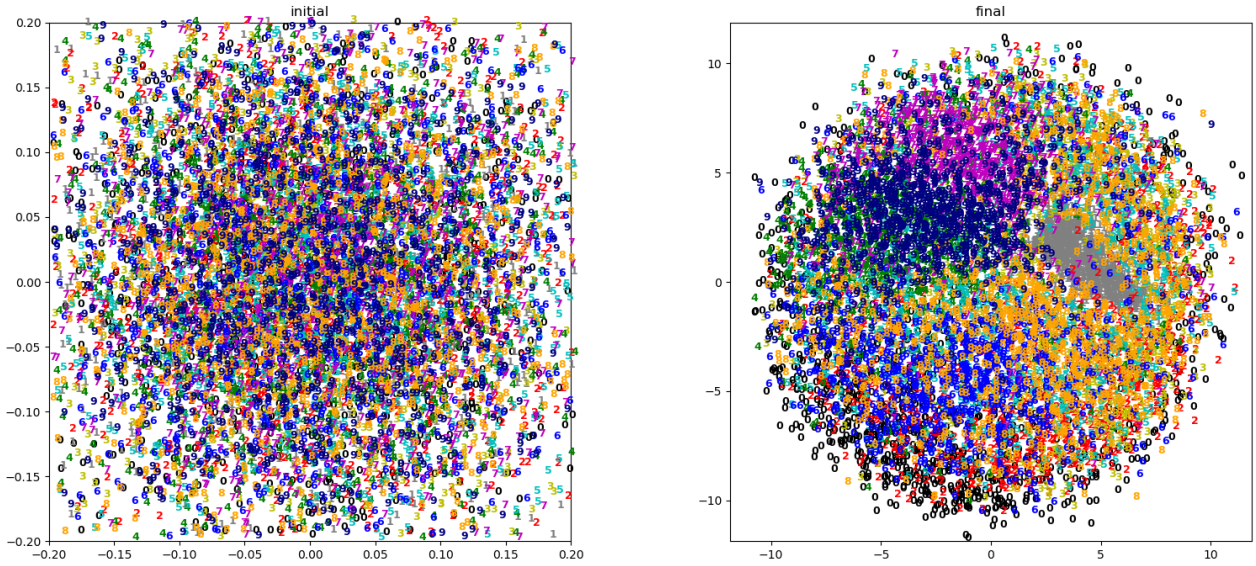

Fig. 1: MDS of mnist into 2-d space(different colors for different digits)
left: initial distribution right: final distribution

Hungarian algorithm[7] is a combinatorial optimization algorithm that solves the assignment problem in polynomial time. The intuitive procedure of Hungarian algorithm can be described as:

step 0:

- Initialize the assignment cost matrix D.

step 1:

- Update D by subtracting minimum of every row.

step 2:

- Update D by subtracting minimum of every column.

step 3:

- Cover all 0s with minimum number of rows and columns.

step 4:

- If the number of covered rows and columns is n, we are done. Go to step 5.
- If not, find the minimum value in the uncovered area. Add it to uncovered rows. Subtract it to covered columns. Go to step 4.

step 5:

- Derive the best assignment based on the covered rows and columns.

The hardest part is step 3. We can use various combination of rows and columns, but to find the way using the least rows and columns is not trivial, especially when the scale of the problem get larger. The detailed steps for the reference implementation are described below.

step 0:

- Initialize the assignment cost matrix D. D_{ij} means the cost of assigning i to j. D doesn't have to be a symmetric matrix. Most of the case it is not symmetric.

step 1:

- Update D by subtracting minimum of every row. Then there is at least one 0 in each row of D.
- Go to step 2.

step 2:

- Start from the first row first column, mark the 0 with **1** if its row and column is uncovered. After marking that 0 with **1**, making its row and column covered. So each row and each column will have at most one marked 0.
- Clear the covered state of row and column.
- Go to step 3.

step 3:

- Count how many columns have marked 0. Mark the columns with marked 0 covered.
  - If not all columns have marked 0, go to step 4. (At this stage, all the rows are uncovered. Only some columns are covered.)
  - If all columns have marked 0, then the marked 0 are the best assignment. Finish these steps.

step 4:

- Find uncovered 0 in D, i.e. 0 that is in both uncovered row and uncovered column. Store in coverd_D. coverd_D has 1 means an uncovered 0.
  - If there is no uncovered 0 in coverd_D, go to step 6.
  - If there is at least one uncovered 0 in coverd_D:

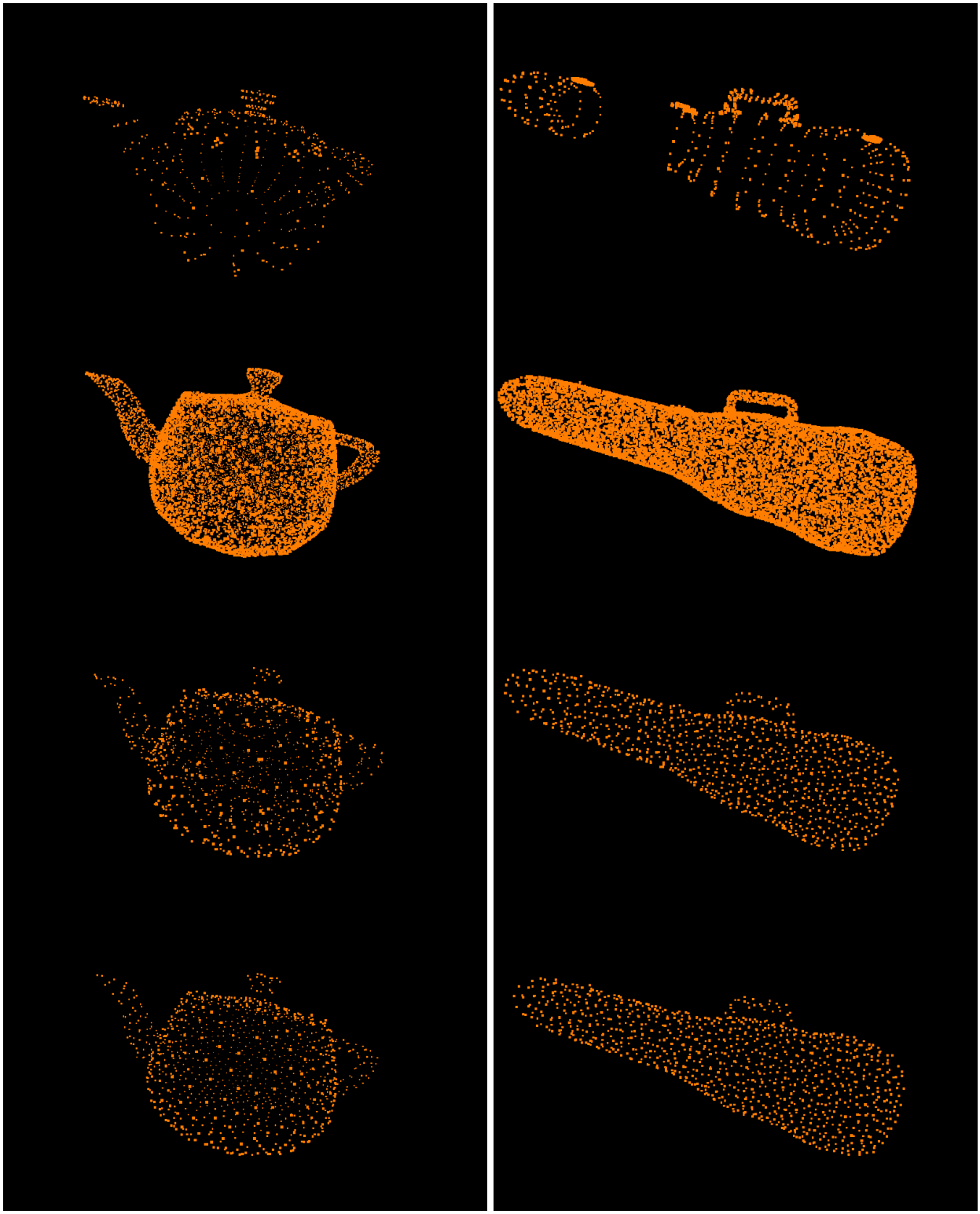

Fig. 2: Farthest Point Sampling result
row1:initial vertiecs row2:dense point(10000) row3:sparse point(1000) on-mesh row4:sparse point(1000) euclidean
left:teapot right:violin case

* Mark one of the uncovered 0 with **2**.
* Check if there is a **1** marked 0 in this **2** marked 0's row
  - If there is not a **1** marked 0 in this row: Store the place of this **2** marked 0 in Z0_r,Z0_c. Go to step 5.
  - If there is a **1** marked 0 in this row: Set this row to be covered. Set the **1** marked 0 column to be uncovered. Set that column of coverd_D to keep only 0 in uncovered rows. Set that row of coverd_D to be all 0. Check coverd_D to find uncovered 0 again.

step 5:

- Z0 is the uncovered **2** marked 0 in step 4. Z1 is the **1** marked zero in the column of Z0 (if any). Z2 is the **2** marked zero in the row of Z1 (there will always be one).
- Initialize the count to be 0, the path to be all 0 for matrix (2n)*2
- Add Z0 to the path so the current point in path is Z0.
- Find the first **1** marked 0 in the col defined by the current node in path. There is always at least one.
  - If there is not one, break.
  - If there is one, put that **1** marked 0 into the path and let it be the current node. Continue for next **1** marked 0.
- Find the first **2** marked 0 in the row defined by the current node in path.
- Convert paths. For each node in path, if it was **1** marked 0, delete that mark; if not, then make it **1** marked 0.
- Clear the marked on rows and columns.
- Clear the **2** marked 0.
- Go to step 3.

step 6:

- Find the smallest value of D in both uncovered row and column, i.e. the row and column that are not all zeros. That is, find the smallest not-0 value in D. Name it d.
- Add d to each covered row.
- Subtract d to each uncovered column.
- Go to step 4.

For the second part of the problem, we just formulate an optimization problem like question 1. Here we first generate 100 different radium from a uniform distribution. Each radium represents a point cloud sampled on the circle with that radium and we can sample it. So we have a series of point clouds now. Here I also initialize the point set X with truncated normal distribution. The only problem that needs to be take care is that here the loss is relatively small so we may want to use a relatively big learning rate, or enlarge the loss by timing a large scalar.

Fig. 3 shows the final optimization results. The gray points are those sampled points on circles with different radiums. The red points are the optimal x. It looks reasonable that it is nearly at the middle place of the series of circles.

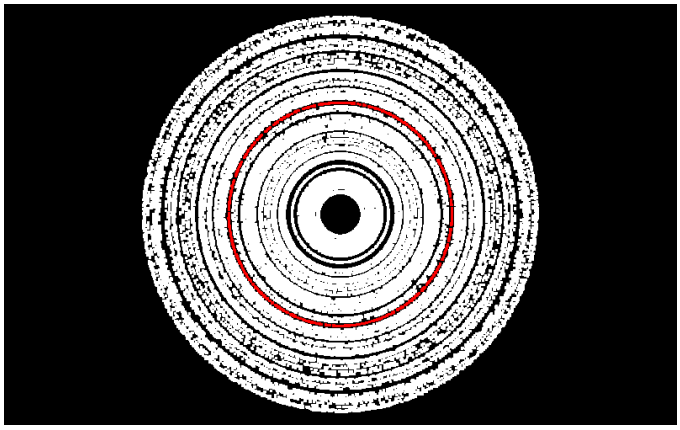

Fig. 3: red points to make Earth Mover's distance smallest

4. Denoising Autoencoder

For autoencoder we get quite a lot intuition from [9], especially about its symmetric structure. Also according to [10] we use *tf.image.resize_nearest_neighbor* and *tf.layers.conv2d* instead of *tf.nn.conv2d_transpose* to avoid checkerboard artifacts.

Given a $28 \times 28 \times 1$ image, we go through:

- Encoder
  - convolution layer with 32 kernels with size 3×3. Same padding. ReLU.
    $(28 \times 28 \times 32)$
  - maxpooling layer with pool size(2,2), strides(2,2). Same padding.
    $(14 \times 14 \times 32)$
  - convolution layer with 32 kernels with size 3×3. Same padding. ReLU.
    $(14 \times 14 \times 32)$
  - maxpooling layer with pool size(2,2), strides(2,2). Same padding.
    $(7 \times 7 \times 32)$
  - convolution layer with 16 kernels with size 3×3. Same padding. ReLU.
    $(7 \times 7 \times 16)$
  - maxpooling layer with pool size(2,2), strides(2,2). Same padding.
    $(4 \times 4 \times 16)$
- Decoder
  - Nearest neighbor resize layer. Size(7,7).
    $(7 \times 7 \times 16)$
  - convolution layer with 16 kernels with size 3×3. Same padding. ReLU.
    $(7 \times 7 \times 16)$
  - Nearest neighbor resize layer. Size(14,14).
    $(14 \times 14 \times 16)$
  - convolution layer with 32 kernels with size 3×3. Same padding. ReLU.
    $(14 \times 14 \times 32)$

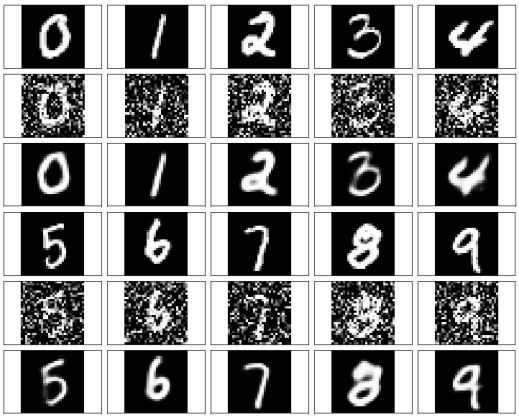

Fig. 4: Denoising Autoencoder
row 1/4:original row 2/5:noise row 3/6: recovered

- Nearest neighbor resize layer. Size(28,28).
  $(28 \times 28 \times 32)$
- convolution layer with 32 kernels with size 3×3. Same padding. ReLU.
  $(28 \times 28 \times 32)$
- convolution layer with 1 kernels with size 3×3. Same padding.
  $(28 \times 28 \times 1)$

Then we compute the sigmoid cross-entropy loss between the output of decoder and the input of encoder. Our object is to minimize this loss. Besides, when we want to use the parameters of this autoencoder to generated the recovered image we want to add one more sigmoid after the output of decoder in order to make sure it is always ranging from 0 to 1.

Fig. 4 visualizes the performance of denoising. It is relatively good. Here we set the noise variance to 0.5.

REFERENCES

[1] Visualizing MNIST: An Exploration of Dimensionality Reduction - colah's blog
[2] Sammon Embedding with Tensorflow - Everything about Data Analytics
[3] GitHub - qnzhou/PyMesh: Geometry Processing Library for Python
[4] GitHub - daavoo/pyntcloud: pyntcloud is a Python library for working with 3D point clouds.
[5] Osada, R., Funkhouser, T., Chazelle, B., & Dobkin, D. (2002). Shape distributions. ACM Transactions on Graphics (TOG), 21(4), 807-832.
[6] scikit-learn/linear_assignment_.py at master scikit-learn/scikit-learn GitHub
[7] Hungarian algorithm - Wikipedia
[8] Hungarian Algorithm for Assignment Problem — Set 1 (Introduction) - GeeksforGeeks
[9] deep-learning/Convolutional_Autoencoder_Solution.ipynb at master udacity/deep-learning GitHub
[10] Deconvolution and Checkerboard Artifacts