NCTU Machine Learning 2021Spring HW7

Kernel Eigenfaces and t-SNE

0610168 沈立淞

Kernel Eigenfaces Part

I. Code Explanations:

For the sake of Manifold assumption and convenience of visualization, we need a way to help us reduce data dimensionality and retain most of information at the same time. Here we implement PCA, kernel PCA, LDA and kernel LDA on Face database.

Parameters and data preprocessing

Figure1 shows the implementation of data loader and setting parameters.
KNN_K indicates the KNN's number of reference points.
I firstly resize the face dataset to 50X50, which has 2500 pixels, and then reshape the image to a 1X2500 numpy array, adding this to data numpy array.

```
train_size = 135
test_size = 30
total_size = train_size + test_size
KNN_K = 5
shape = (50,50)

def dataLoader():
    path_name = './Yale_Face_Database/Training/'
    files = os.listdir(path_name)
    train_data = []
    train_label = []
    for file in files:
        label = re.search(r'\d+',file)
        train_label.append(label.group())
        img = cv2.imread(path_name +'/'+file,-1)
        img = cv2.resize(img, shape)
        train_data.append(img.reshape((-1)))
    train_data = np.array(train_data)
    train_label = np.array(train_label)

    path_name = './Yale_Face_Database/Testing/'
    files = os.listdir(path_name)
    test_data = []
    test_label = []
    for file in files:
        label = re.search(r'\d+',file)
        test_label.append(label.group())
        img = cv2.imread(path_name +'/'+file,-1)
        img = cv2.resize(img, shape)
        test_data.append(img.reshape((-1)))
    test_data = np.array(test_data)
    test_label = np.array(test_label)

    return train_data, train_label, test_data, test_label
```

Fg.1 parameters and dataloader

PCA

To define dimensional reduction, our goal is to find a suitable projection matrix W to project data x to low dimension z, i.e. z = W*x. According to lecture slides, the principle component is covariance matrix's eigenvectors corresponding to maximum eigenvalue, and we can get the optimal solution W by combine most significant eigenvectors.

For eigenfaces, we simply reshape 25 most significant eigenvectors (i.e. W) of covariance matrix to a 50X50 image and then output the data.

For reconstruction, we re-project the new data to original dimension and add the original mean value:

$$\mathrm{X}_{reproject} = W^T * z + \mu$$

The following figure shows my PCA's procedure, visualization of eigenfaces and reconstructing original data face.

```
def PCA(train_data, train_label, test_data, test_label):
    print('PCA')
    mean_face = ((np.sum(train_data,axis = 0)/train_size).astype(int)).reshape((1,-1))
    Xc = train_data - mean_face
    COV_T = Xc @ Xc.T
    eigenVal, eigenVec = linalg.eigh(COV_T)
    idx = np.argsort(eigenVal)[::-1]
    eigenVal = eigenVal[idx]
    U = eigenVec[:,idx]
    W = (Xc.T @ U)[:,0:25]
    for i in range(25):
        W[:,i] = W[:,i]/linalg.norm(W[:,i])

    proj = Xc @ W
    reconstruct_face = proj @ W.T + mean_face
    visualization(W, reconstruct_face, 'PCA')


    new_train = []
    new_test = []
    for i in range(train_size):
        proj_coef = (train_data[i,:]- mean_face) @ W
        new_train.append(proj_coef)

    for i in range(test_size):
        proj_coef = (test_data[i,:]- mean_face) @ W
        new_test.append(proj_coef)

    new_train = np.array(new_train)
    new_test = np.array(new_test)
    knn(new_train, new_test, train_label, test_label)
```

Fg.2 PCA

```python
def visualization(eigenVec, reconstruct_face, method):
    #showing eigen and fisherfaces
    for i in range(25):
        plt.subplot(5, 5, i+1)
        plt.imshow(eigenVec[:,i].reshape(shape), cmap = 'gray')
        plt.savefig(method+'.png')

    #reconstruct faces
    idx = np.random.randint(train_size, size = 10)
    for i,id in enumerate(idx):
        plt.subplot(2, 5, i+1)
        plt.imshow(reconstruct_face[id].reshape(shape), cmap = 'gray')
        plt.savefig(method+'_'+str(idx)+'_'+'.png')
```

Fg.3 visualization of eigenfaces, fisherfaces and reconstructing original faces.

For face recognition, using **knn()** to do k-nearest neighbor algorithm. For each test data, the key idea to do recognition by referring to k-nearest Euclidean distance point and voting for test data's predict label.

```python
def knn(trains, tests,train_label, test_label):
    correct = 0
    for Did,data in enumerate(tests):
        print('ID: {}'.format(Did))
        distance = []
        for train in trains:
            dist = cdist(data.reshape((1,-1)), train.reshape((1,-1)))[0][0]
            distance.append(dist)
        idx = np.argsort(distance)
        neighbor_labels = train_label[idx]
        unique, pos = np.unique(neighbor_labels[0:KNN_K], return_inverse=True)
        counts = np.bincount(pos)
        maxpos = counts.argmax()
        predict_label = unique[maxpos]
        print('Predict label: {}'.format(predict_label))
        print('Actual label: {}'.format(test_label[Did]))
        print('')
        if(predict_label == test_label[Did]):
            correct += 1
    print('Accuracy: {}'.format(correct/len(tests)))
```

Fg.4 K-nn

LDA

To add consideration into labeled data, we use LDA to find projection matrix. LDA's thought is to maximize between-class scatter Sw:

$$\mathrm{Sw} = \sum_{j=1}^{k} S_j$$

$$S_j = \sum_{i \epsilon C_j} (Xi - \ m_j)(Xi - \ m_j)^T$$

$$m_j = \frac{1}{n_j} \sum_{i \epsilon C_j} Xi$$

And withing-class Sb:

$$\mathrm{Sb} = \sum_{j=1}^{k} S_{B_j} = \sum_{j=1}^{k} n_j (m_j - m)(m_j - m)^T$$

$$m = \frac{1}{n} \sum x$$

Then we can change the original problem into eigenvactor solving problem:

$$\mathrm{Sw}^{-1} * \mathrm{Sb} * \mathrm{w} = \lambda \mathrm{w}$$

In our homework, our datasize n is smaller than original dimension D, which will make Sw uninvertible. Thus here we use **np.linalg.pinv(Sw)** to compute pseudo inverse of Sw. After computing most significant eigenvectors, we than combine these vectors as projection matrix, and do what we have done in PCA:use W to show fisherfaces, reconstruct faces use W and doing recognition using KNN with projected data.

```
def LDA(train_data, train_label, test_data, test_label):
    print('LDA')
    mean_face = ((np.sum(train_data,axis = 0)/train_size).astype(int)).reshape((1,-1))
    Xc = train_data - mean_face
    mean = np.mean(train_data,axis=0)
    data, cluster_mean = clustering_by_label(train_data, train_label)

    #compute Sw, Sb
    Sw = np.zeros((shape[0]*shape[1],shape[0]*shape[1]))
    Sb = np.zeros((shape[0]*shape[1],shape[0]*shape[1]))
    for i in range(len(cluster_mean)):
        data[i] = data[i] - cluster_mean[i]
        for j in range(len(data[i])):
            mat = data[i][j].reshape((-1,1)) @ data[i][j].reshape((1,-1))
            Sw += mat
    cluster_mean = cluster_mean - mean
    for i in range(len(cluster_mean)):
        ni = len(data[i])
        mat = ni * cluster_mean[i].reshape((-1,1)) @ cluster_mean[i].reshape((1,-1))
        Sb += mat


    #solve eigenvector problem
    eigenVal, eigenVec = linalg.eig(np.linalg.pinv(Sw) @ Sb)
    idx = np.argsort(eigenVal)[::-1]
    W = eigenVec[:,idx]
    W = W[:,:25].real
    for i in range(25):
        W[:,i] = W[:,i]/linalg.norm(W[:,i])
    proj = Xc @ W
    reconstruct_face = proj @ W.T + mean_face
    visualization(W, reconstruct_face, 'LDA')

    new_train = []
    new_test = []
    for i in range(train_size):
        proj_coef = (train_data[i,:]- mean_face) @ W
        new_train.append(proj_coef)

    for i in range(test_size):
        proj_coef = (test_data[i,:]- mean_face) @ W
        new_test.append(proj_coef)

    new_train = np.array(new_train)
    new_test = np.array(new_test)
    knn(new_train, new_test, train_label, test_label)
```

Fg.5 LDA

Kernel PCA

Kernel PCA is a way to deal with nonlinear data. The idea is:

project data to feature space=> computing covariance matrix =>solving eigenvector problem => project feature data to lower dimension space.

Thanks to kernel tricks, we can compute the projected feature data without knowing the feature space.

First, use kernel function to compute kernel matrix **K** and doing centralization for data:

$$\mathrm{K}^{C} = K - \ 1_N K - K 1_N + \ 1_N K 1_N$$

$$1_N \ \text{is NxN matrix with every element } 1/\mathrm{N}$$

Let **a** be the significant eigenvectors of K^{C}, than we can use following formula to find projected feature data:

$$\mathrm{W}\Phi(\mathrm{Xnew}) = K(Xi, Xnew) * \mathbf{a}$$

And thus , again, the problem becomes to an eigenvector problem. We than use $\mathrm{W}\Phi(\mathrm{Xnew})$ to do recognition with knn.

```
def kernelPCA(train_data, train_label, test_data, test_label):
    print('kernelPCA')
    data = np.vstack((train_data, test_data))
    K = RBF_kernel(data)
    N1 = np.full((K.shape), 1/K.shape[0])
    Kc = K - N1 @ K - K @ N1 + N1 @ K @ N1
    eigenVal, eigenVec = linalg.eig(Kc)
    idx = np.argsort(eigenVal)[::-1]
    W = eigenVec[:,idx][:,:25].real
    for i in range(25):
        W[:,i] = W[:,i]/linalg.norm(W[:,i])
    proj = Kc.T @ W
    newtrain = proj[:train_size]
    newtest = proj[train_size:]
    knn(newtrain, newtest , train_label, test_label)
```

Fg.6 kernel PCA

As for kernel function, I define several kernels, including RBF kernel, linear kernel, polynomial kernel and tanh kernel.

```python
def RBF_kernel(X):   #compute kernel(gram) matrix, which use rational quadratic kernel as kernel function
    alpha = 0.00001
    kernel = np.zeros((len(X),len(X)))
    for i in range(len(X)):
        for j in range(len(X)):
            value = np.exp(-alpha * (np.sum((X[i,:]-X[j,:])**2)))
            kernel[i,j] = value
    return kernel


def poly_kernel(X):
    d = 2
    kernel = np.zeros((len(X),len(X)))
    for i in range(len(X[:,0])):
        for j in range(len(X)):
            value = (X[i].dot(X[j]))**d
            kernel[i,j] = value
    return kernel

def linear_kernel(X):
    kernel = X @ X.T
    return kernel

def tanh_kernel(X):
    kernel = np.zeros((len(X),len(X)))
    for i in range(len(X)):
        for j in range(len(X)):
            value = np.tanh(X[i].dot(X[j]))
            kernel[i,j] = value
    return kernel
```

Fg.7 kernel function

Kernel LDA

The same idea as Kernel PCA, we use kernel trick to compute the projected data by technically compute within and between scatter. We use the following formula to formulate eigenvector problem:

$$M = \sum_{j=1}^{c} l_j (\mathbf{M}_j - \mathbf{M}_*)(\mathbf{M}_j - \mathbf{M}_*)^{\mathrm{T}}$$

$$N = \sum_{j=1}^{c} \mathbf{K}_j (\mathbf{I} - \mathbf{1}_{l_j}) \mathbf{K}_j^{\mathrm{T}}.$$

$$(\mathbf{M}_*)_j = \frac{1}{l} \sum_{k=1}^{l} k(\mathbf{x}_j, \mathbf{x}_k).$$

where the i^{th} component of $\mathbf{K}_t$ is given by $k(\mathbf{x}_i, \mathbf{x}_t)$.

Using M,N to construct eigenvector problem:

$$N^{-1}Mw = \lambda w$$

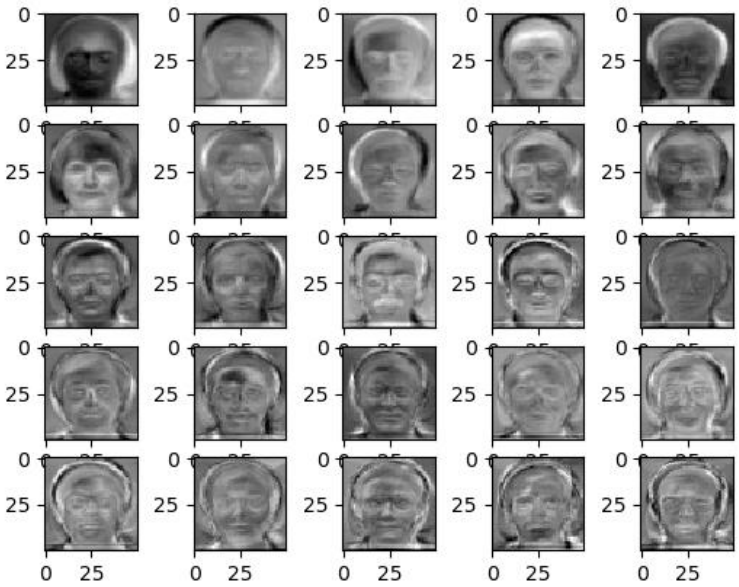
As same scenario in LDA, we need to compute N^{-1} by computing pseudo inverse of N.

```python
def kernelLDA(train_data, train_label, test_data, test_label):
    print('kernelLDA')
    data = np.vstack((train_data, test_data))
    data_label = np.hstack((train_label, test_label))
    K = RBF_kernel(data)
    mean = np.mean(K,axis=0)
    data, cluster_mean = clustering_by_label(train_data, train_label)
    M = np.zeros((total_size,total_size))
    N = np.zeros((total_size,total_size))
    clusters = np.unique(train_label)
    for cluster in clusters:
        row_id = np.where(data_label == cluster)[0]
        Ki = K[row_id, :]
        l = len(row_id)
        N1 = np.full((l,l), 1/l)
        N = N + Ki.T @ (np.identity(l) - N1) @ Ki
        cluster_mean = np.mean(Ki, axis=0)
        M = M + l * (cluster_mean - mean).T @ (cluster_mean - mean)
    eigenVal, eigenVec = linalg.eig(np.linalg.pinv(N) @ M)
    idx = np.argsort(eigenVal)[::-1]
    W = eigenVec[:,idx][:,:25].real
    for i in range(25):
        W[:,i] = W[:,i]/linalg.norm(W[:,i])
    proj = K.T @ eigenVec
    newtrain = proj[:train_size]
    newtest = proj[train_size:]
    knn(newtrain, newtest , train_label, test_label)
```

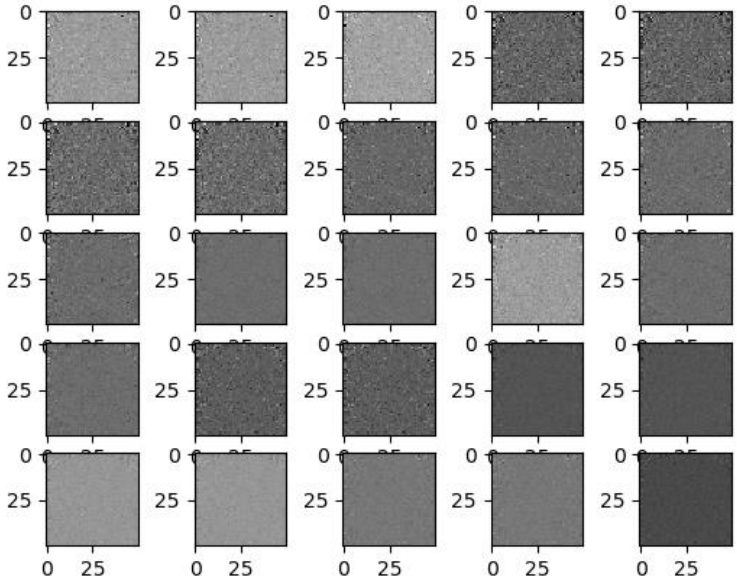
Fg.8 kernel LDA

II. Experimenting and results:

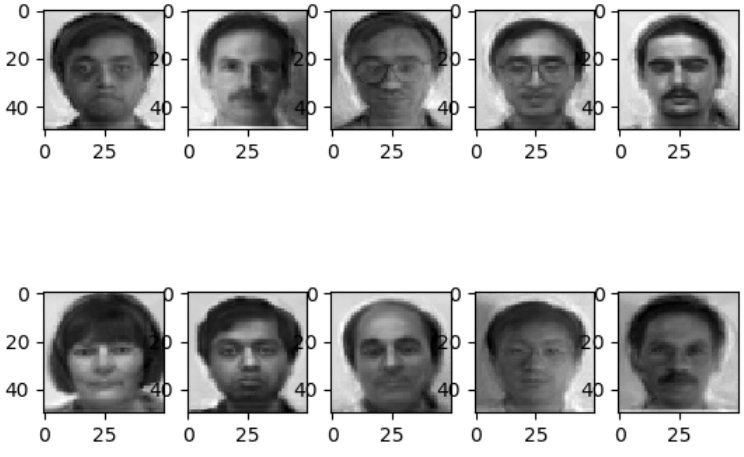
Eigenfaces, Fisherfaces and reconstruction



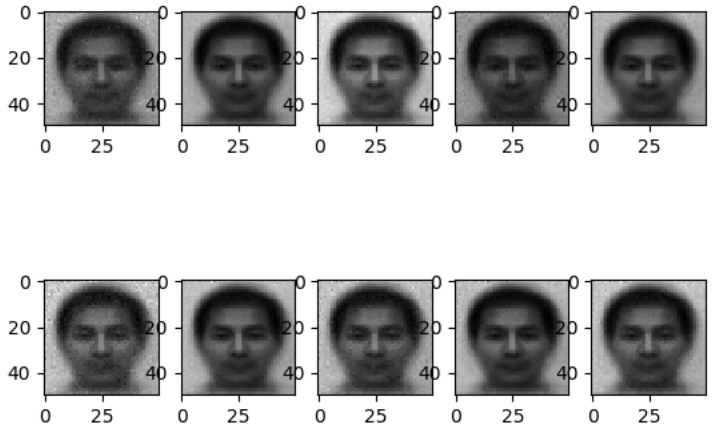
25 eigenfaces



25 fisherfaces



10 randomly picked reconstruction using eigenfaces



10 randomly picked reconstruction using fihserfaces

PCA and LDA Face recognition

PCA	LDA
K=1: Accuracy = 0.83	K=1: Accuracy = 0.66
K=3: Accuracy = 0.83	K=3: Accuracy = 0.6
K=5: Accuracy = 0.9	K=5: Accuracy = 0.73
K=7: Accuracy = 0.86	K=7: Accuracy = 0.66
K=9: Accuracy = 0.8	K=9: Accuracy = 0.6
K=11: Accuracy = 0.8	K=11: Accuracy = 0.66
K=13: Accuracy = 0.83	K=13: Accuracy = 0.53

Kernel PCA and Kernel LDA Face recognition

Kernel PCA	Kernel LDA
K=1: Accuracy = 0.8	K=1: Accuracy = 0.63
K=3: Accuracy = 0.8	K=3: Accuracy = 0.53
K=5: Accuracy = 0.83	K=5: Accuracy = 0.66
K=7: Accuracy = 0.83	K=7: Accuracy = 0.56

K=9: Accuracy = 0.86	K=9: Accuracy = 0.53
K=11: Accuracy = 0.83	K=11: Accuracy = 0.5
K=13: Accuracy = 0.83	K=13: Accuracy = 0.5

t-SNE Part

I. Code Explanations:

t-SNE and SSNE

SNE is used to convert the high-dimensional Euclidean distances into conditional probability that represent similarities, which is highly used in data visualization. The difference between symmetric SNE and t-SNE is the use of probability distribution. For symmetric SNE, it uses Gaussian distribution in both high and low dimension space. We use the following formula to represent the probability and gradient descent:

$$p_{ij} = \frac{\exp(-\frac{\|x_i - x_j\|^2}{2\sigma^2})}{\sum_{k \neq l} \exp(-\frac{\|x_l - x_k\|^2}{2\sigma^2})}$$

$$q_{ij} = \frac{\exp(-\|y_i - y_j\|^2)}{\sum_{k \neq l} \exp(-\|y_l - y_k\|^2)}$$

$$\frac{\partial C}{\partial y_i} = 2\sum_j (p_{ij} - q_{ij})(y_i - y_j)$$

However, it has crowding problem, which usually occurs when we output the data from high dimension to low dimension. And thus, we use t-SNE to overcome this phenomenon. For t-SNE, it also uses Gaussian distribution in high dimension space, but uses student-t distribution in low dimension space. The following code shows formula of t-SNE probability:

$$p_{ij} = \frac{\exp(-\frac{\|x_i - x_j\|^2}{2\sigma^2})}{\sum_{k \neq l} \exp(-\frac{\|x_l - x_k\|^2}{2\sigma^2})}$$

$$q_{ij} = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_{k \neq l}(1 + \|y_l - y_k\|^2)^{-1}}$$

$$\frac{\partial C}{\partial y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)\,(1 + \|y_i - y_j\|^2)^{-1}$$

The following figure shows implementation of symmetric SNE modified from original t-SNE code:

```
if (method == 'ssne'):                              #modified
    num = -2. * np.dot(Y, Y.T)
    num = (-1. *  np.add(np.add(num, sum_Y).T, sum_Y))
    num = np.exp(num)

num[range(n), range(n)] = 0.
Q = num / np.sum(num)
Q = np.maximum(Q, 1e-12)
```

Figure. Low dimension probability modified from t-SNE

```
PQ = P - Q
for i in range(n):
    if(method=='tsne'):
        dY[i, :] = np.sum(np.tile(PQ[:, i] * num[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)

    if(method=='ssne'):                                  #modified
        dY[i, :] = np.sum(np.tile(PQ[:, i] , (no_dims, 1)).T * (Y[i, :] - Y), 0)
```

Figure. Gradient descent modified from t-SNE

Visualization the embedding of t-SNE and SSNE

Simply use output Y to plot scatter using matplotlib:

```
def visualization(Y,method, iter ,labels):

    plt.figure()
    plt.scatter(Y[:, 0], Y[:, 1], 20, labels)
    plt.savefig('./result/'+method +'_'+ str(iter) +'_' + 'perplexity' + str(perplexity)+'.png')
```

Figure. Visualization the embedding of t-SNE and SSNE

Visualizing the distribution of pairwise similarities

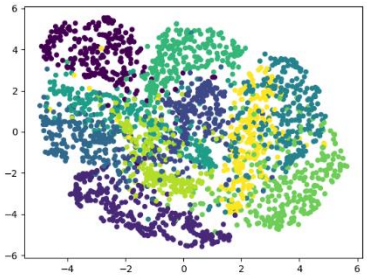
```
def visualization_pair(P, Q,method):
    plt.figure()
    plt.imshow(P, cmap='Spectral')
    plt.savefig('./result/'+method +'_highDim_' +'_' + 'perplexity' + str(perplexity)+'.png')


    plt.figure()
    plt.imshow(Q, cmap='Spectral')
    plt.savefig('./result/'+method +'_lowDim_' +'_' + 'perplexity' + str(perplexity)+'.png')
```

Figure. Visualization the distribution of pairwise similarities

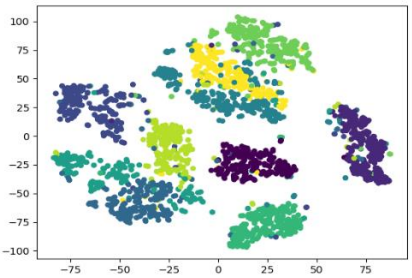
II. Experimenting and results:

Visualizing the Embedding of SSNE and TSNE

SSNE, perplexity=20

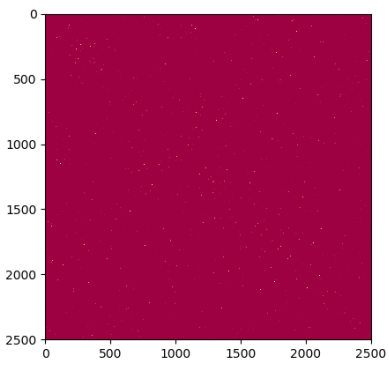


TSNE,perplexity=20

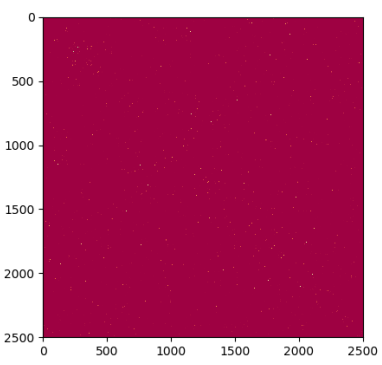


As we shall see, crowding problem in symmetric SNE is much more serious than t-SNE. In t-SNE, the scale in both X and Y are larger than symmetric SNE, which relieves the pressure of crowing problem in shown in symmetric SNE.

Visualizing the distribution of pairwise similarities

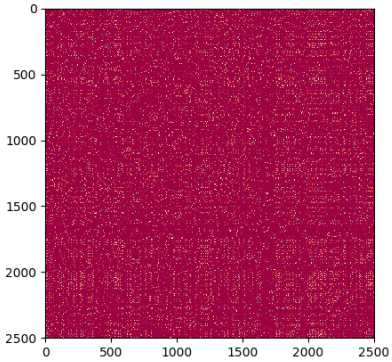


SSNE, perplexity=20



TSNE, perplexity=20

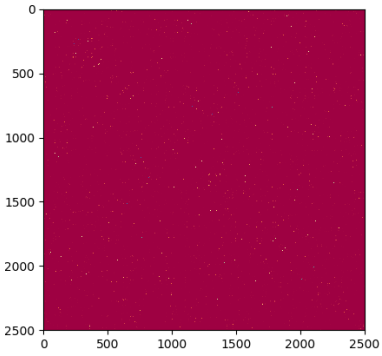
High Dimension



SSNE, perplexity=20

Low Dimension

High dimension

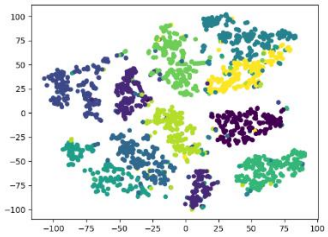


TSNE, perplexity=20

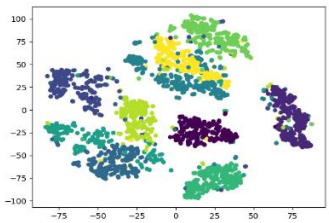
Low Dimension

Different Perplexity

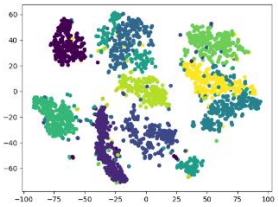
TSNE



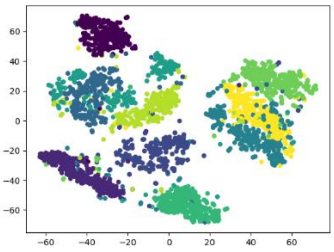
perplexity=10



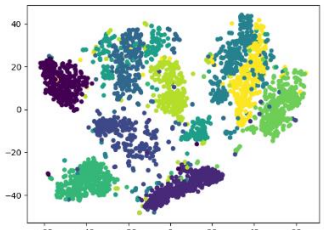
perplexity=20



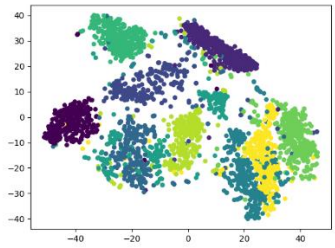
perplexity=30



perplexity=50



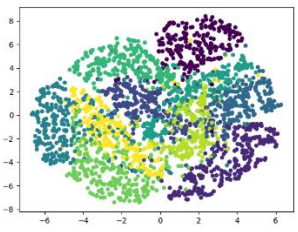
perplexity=70



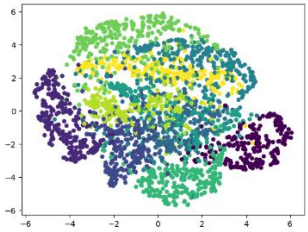
perplexity=100

As we shall see in t-SNE for small perplexity, crowding problem isn't obvious, and the scale is larger, while for larger perplexity, crowding problem is to be more serious, and the scale is smaller.

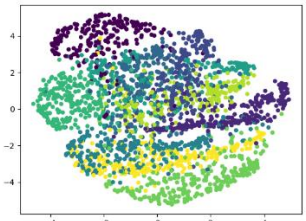
SSNE



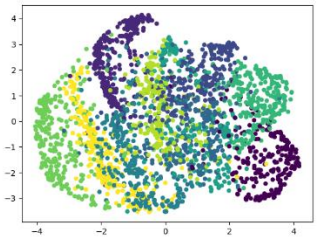
perplexity=10



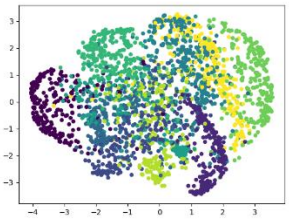
perplexity=20



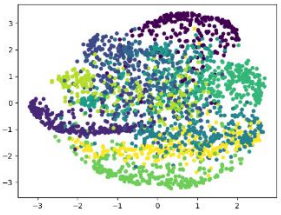
perplexity=30



perplexity=50



perplexity=70



perplexity=100

In symmetric SNE, I think there's no significant difference in different value of perplexity, and the crowding problem is remaining serious for all kinds of value of perplexity.