a.code with detailed explanations

Kernel Eigenface

Part1

PCA.

1. Load data.

2. Use all data as input of “PCA” function to compute W matrix. “PCA” function will do following step:
   i. compute mean
   ii. compute covariance matrix
   iii. find eigenvector of the covariance matrix, normalize it and sort it, then use first 25 big eigenvector as ‘W’ matrix and return it.

3. Use ‘W’ matrix as a input of “draw” function to draw 25 eigenfaces.
   ‘W’ is a (m*25) matrix, it can seem as 25 eigenfaces image with m pixel.

4. Do reconstruct, compute x @ W @ W.T. (formula come from the lecture)
   For more accuracy, here I use (data - mu) as ‘x’, and add a mu in the end.

$$\underbrace{xW}_{z}\underline{W^\top}$$

5. Use “show_reconstruction” function to save a .png image.

```python
if __name__=="__main__":
    X, X_filename, X_label = readData('./Yale_Face_Database/Training')
    test, test_filename, test_label = readData('./Yale_Face_Database/Testing')


    data = np.vstack((X, test))
    filename = np.hstack((X_filename, test_filename))
    label = np.hstack((X_label, test_label))



    # PCA
    # Q1
    W = PCA(data)
    draw( 'pca_eigenface', W)

    mu = np.mean(data, axis=0)
    re = (data-mu) @ W @ W.T  + mu
    show_reconstruction(data,re,10,SHAPE[0],SHAPE[1])
```

```python
def PCA(X):


    # 找到mean
    mu = np.mean(X, axis=0)
    # 先標準化，mu變為0
    st_X = X - mu
    # 算covariance
    cov = st_X.T @ st_X
    # 找到cov的eigen
    eigen_val, eigen_vec = np.linalg.eigh(cov)

    # 排序後，取前 dims個 特徵向量回傳
    for i in range(eigen_vec.shape[1]):
        eigen_vec[:, i] = eigen_vec[:, i] / np.linalg.norm(eigen_vec[:, i])
    idx = np.argsort(eigen_val)[::-1]
    W = eigen_vec[:, idx]
    W = W[:, :dims].real


    return W
```

```python
def draw( title, W):

    folder = f"{title}_{time.time()}"
    os.mkdir(folder)
    os.mkdir(f'{folder}/{title}')

    plt.clf()
    for i in range(5):
        for j in range(5):
            idx = i * 5 + j
            plt.subplot(5, 5, idx + 1)
            plt.imshow(W[:, idx].reshape(SHAPE), cmap='gray')
            plt.axis('off')
    plt.savefig(f'./{folder}/{title}/{title}.png')

    for i in range(W.shape[1]):
        plt.clf()
        plt.title(f'{title}_{i + 1}')
        plt.imshow(W[:, i].reshape(SHAPE), cmap='gray')
        plt.savefig(f'./{folder}/{title}/{title}_{i + 1}.png')
```

```python
def show_reconstruction(X,X_recover,num,H,W):

    randint=np.random.choice(X.shape[0],num)
    for i in range(num):
        plt.subplot(2,num,i+1)
        plt.imshow(X[randint[i],:].reshape(H,W),cmap='gray')
        plt.subplot(2,num,i+1+num)
        plt.imshow(X_recover[randint[i],:].reshape(H,W),cmap='gray')
    plt.savefig(f'./reconstruct/reconstruction{str(SHAPE[0])}.png')
    plt.show()
```

LDA.

1. Use all data as input of “LDA” function to compute U matrix. “LDA” function will do following step:
    i. compute SW. (formula come from the lecture)

$$S_W = \sum_{j=1}^{k} S_j, \text{ where } S_j = \sum_{i \in \mathcal{C}_j} (x_i - \mathbf{m}_j)(x_i - \mathbf{m}_j)^\top$$

$$\text{and } \mathbf{m}_j = \frac{1}{n_j} \sum_{i \in \mathcal{C}_j} x_i$$

ii. compute SB. (formula come from the lecture)

$$S_B = \sum_{j=1}^{k} S_{B_j} = \sum_{j=1}^{k} n_j (\mathbf{m}_j - \mathbf{m})(\mathbf{m}_j - \mathbf{m})^\top$$

$$\text{where } \mathbf{m} = \frac{1}{n} \sum x$$

iii. compute eigenvector of (SW_inverse @ SB) , normalize it and sort it, then use first 25 big eigenvector as ‘U’ matrix and return it.

2. Do reconstruct, compute x @ U @ U.T. .
(formula come from the lecture)
For more accuracy, here I use (data - mu) as ‘x’, and add a mu in the end.

$$\underbrace{xW}_{z}\underline{W}^\top$$

3. Use “show_reconstruction” function to save a .png image. Same as above.

```
# LDA
# Q1


U = LDA(data, label)
print(U.shape)
draw( 'lda_fisherface', U)
mu = np.mean(data, axis=0)
re = (data-mu) @ U @ U.T  +mu
show_reconstruction(data,re,10,SHAPE[0],SHAPE[1])
```

```python
def LDA(X, label):
    (n, d) = X.shape
    label = np.asarray(label)

    c = np.unique(label)
    S_w = np.zeros((d, d), dtype=np.float64)
    S_b = np.zeros((d, d), dtype=np.float64)

    mu = np.mean(X, axis=0)

    # SW
    for i in c:
        X_i = X[np.where(label == i)[0], :]
        #mj
        mu_i = np.mean(X_i, axis=0)
        #(xi-mj)
        st_X_i = X_i-mu_i
        S_w += st_X_i.T @ st_X_i

    # SB
    for i in c:
        X_i = X[np.where(label == i)[0], :]
        #mj
        mu_i = np.mean(X_i, axis=0)
        #(m-mj)
        st_mu_i = mu_i - mu
        #ni
        ni = X_i.shape[0]
        S_b += ni * (st_mu_i.T @ st_mu_i)


    eigen_val, eigen_vec = np.linalg.eig(np.linalg.pinv(S_w) @ S_b)
    for i in range(eigen_vec.shape[1]):
        eigen_vec[:, i] = eigen_vec[:, i] / np.linalg.norm(eigen_vec[:, i])
    idx = np.argsort(eigen_val)[::-1]
    W = eigen_vec[:, idx][:, :dims].real
    return W
```

Part2

PCA.

1. Use all data as input of “PCA” function to compute W matrix.
2. Project train data & test data to ‘W’ matrix. Get “X_proj” & ”test_proj”.
3. Use “X_proj” & “test_proj” as inputs of “Recognition” function to do KNN. “Recognition” function will do following step:
    i. Compute each distance between testdata and traindata, record it, and then sort it.
    ii. Find nearest ‘k’ point as neighbor.
    iii. Get predict result by counting label of neighbor.
    iv. Verify that if the predict result is correct.

```python
# Q2
W = PCA(X)
print(W.shape)
X_proj = X @ W
test_proj = test @ W
print('PCA:')
Recognition(X_proj, X_label, test_proj,test_label)
```

```python
def Recognition(train, train_label, test, test_label ):
    test_num = test.shape[0]
    train_num = train.shape[0]

    all_dist = []
    for i in range(test_num):
        dist = []
        # 計算與每個點之間的距離
        for j in range(train_num):
            this_distance=np.sum((train[j] - test[i]) ** 2)
            dist.append((this_distance, train_label[j]))
        # 依據距離排序
        dist.sort(key=lambda l: l[0])
        #記錄此test data 與其他nebighbor 的關係記錄下來
        all_dist.append(dist)
    k=3
    error = 0
    for i in range(test_num):
        dist = all_dist[i]
        #找到最近的k個當neighbor
        neighbor=dist[:k]
        n_list=[]
        for j in range(k):
            n_list.append(neighbor[j][1])
        neighbor=np.array(n_list)

        # 統計neighbor 的 lable
        neighbor_label, count = np.unique(neighbor, return_counts=True)
        # 最多的當result
        most_n=np.argmax(count)
        predict = neighbor_label[most_n]
        #驗證答案
        if predict != test_label[i]:
            error += 1
    print(f'accuracy: {(1 - error / test_num):>.3f} ({test_num-error}/{test_num})')
    print("==========================================================")
```

LDA.

1. Use all data as input of “LDA” function to compute U matrix.
2. Project train data & test data to ‘U’ matrix. Get “X_proj” & ”test_proj”.
3. Use “X_proj” & “test_proj” as inputs of “Recognition” function to do KNN. (same as above)

```python
# Q2
U=LDA(X,X_label)
print(U.shape)
X_proj = X @ U
test_proj = test @ U
print('LDA:')
Recognition(X_proj, X_label, test_proj,test_label)
```


```python
def Recognition(train, train_label, test, test_label ):
    test_num = test.shape[0]
    train_num = train.shape[0]

    all_dist = []
    for i in range(test_num):
        dist = []
        # 計算與每個點之間的距離
        for j in range(train_num):
            this_distance=np.sum((train[j] - test[i]) ** 2)
            dist.append((this_distance, train_label[j]))
        # 依據距離排序
        dist.sort(key=lambda l: l[0])
        #記錄此test data 與其他nebighbor 的關係記錄下來
        all_dist.append(dist)
    k=3
    error = 0
    for i in range(test_num):
        dist = all_dist[i]
        #找到最近的k個當neighbor
        neighbor=dist[:k]
        n_list=[]
        for j in range(k):
            n_list.append(neighbor[j][1])
        neighbor=np.array(n_list)

        # 統計neighbor 的 lable
        neighbor_label, count = np.unique(neighbor, return_counts=True)
        # 最多的當result
        most_n=np.argmax(count)
        predict = neighbor_label[most_n]
        #驗證答案
        if predict != test_label[i]:
            error += 1
    print(f'accuracy: {(1 - error / test_num):>.3f} ({test_num-error}/{test_num})')
    print("==========================================================")
```


Part3

KernelPCA.

1. Use train data as input of “KernelPCA” function to compute W matrix. “KernelPCA” function will do following step:

   i. compute kernel, kernel type 1) linear 2)polynomial 3)RBF

   ii. Compute KC.(formula come from the lecture)

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

   $\mathbf{1}_N$ **is NxN matrix with every element $1/N$**

   iii. compute eigenvector of KC, normalize it and sort it, then use first 25 big eigenvector as ‘W’ matrix and return it.

3. Project train data & test data to ‘W’ matrix. Get “X_proj” & ”test_proj”.

3. Use “X_proj” & “test_proj” as inputs of “Recognition” function to do KNN. (same as above)

```
# Q3

kernel_type=2
W  = kernelPCA(X, kernel_type)
print(W.shape)
X_proj = X @ W
test_proj = test @ W

print('Kernel PCA:')
Recognition(X_proj, X_label, test_proj,test_label)
```

```
def kernelPCA(X, kernel_type):
    # 計算kernel
    if kernel_type == 1:
        kernel=X.T @ X
    elif kernel_type == 2:
        g=5
        c=10
        degree=2
        kernel = g * (X.T @ X) + c
        kernel = np.power(kernel,degree)
    else:
        g=1e-7
        kernel = -1 * g * scipy.spatial.distance.cdist(X.T, X.T, 'sqeuclidean')
        kernel = np.exp(kernel)

    gram_matrix = kernel
    # 計算它的斜方差
    n = gram_matrix.shape[0]
    one = np.ones((n, n)) / n
    k_cov = gram_matrix - one @ gram_matrix - gram_matrix @ one + one @ gram_matrix @ one

    # 把這個鞋坊差做eigen
    eigen_val, eigen_vec = np.linalg.eigh(k_cov)
    for i in range(eigen_vec.shape[1]):
        eigen_vec[:, i] = eigen_vec[:, i] / np.linalg.norm(eigen_vec[:, i])

    idx = np.argsort(eigen_val)[::-1]
    W = eigen_vec[:, idx]
    W = W[:, :dims].real


    return  W
```

KernelLDA.

1. Use train data as input of “KernelLDA” function to compute U matrix.

"KernelLDA" function will do following step:

i. compute kernel, kernel type 1) linear 2)polynomial 3)RBF

ii. compute M.

$$\mathbf{M} = (\mathbf{M}_2 - \mathbf{M}_1)(\mathbf{M}_2 - \mathbf{M}_1)$$

iii. compute N.

$$\mathbf{N} = \sum_{k=1,2} \mathbf{K}_k (\mathbf{I} - \mathbf{1}_{N_k}) \mathbf{K}_k^{\mathrm{T}}$$

iV. compute eigenvector of (N_inverse @ M), normalize it and sort it, then use first 25 big eigenvector as 'U' matrix and return it.

2. Project train data & test data to 'U' matrix. Get "X_proj" & "test_proj".
3. Use "X_proj" & "test_proj" as inputs of "Recognition" function to do KNN. (same as above)

```
# Q3

kernel_type=1
U  = kernelLDA(X, X_label, kernel_type)
print(U.shape)
X_proj = X @ U
test_proj = test @ U
print('Kernel LDA:')
Recognition(X_proj, X_label, test_proj,test_label)
```

```python
def kernelLDA(X, label, kernel_type):
    label = np.asarray(label)
    c = np.unique(label)

    # compute kernel
    if kernel_type == 1:
        kernel=X.T @ X
    # polynomial
    elif kernel_type == 2:
        g=0.1
        cof=100
        degree=3
        kernel = g * (X.T @ X) + cof
        kernel = np.power(kernel,degree)
    # RBF
    else:
        g=1
        kernel = -1 * g * scipy.spatial.distance.cdist(X.T, X.T, 'sqeuclidean')
        kernel = np.exp(kernel)

    n = kernel.shape[0]
    mu = np.mean(kernel, axis=0)
    N = np.zeros((n, n), dtype=np.float64)
    M = np.zeros((n, n), dtype=np.float64)

    # compute M
    for i in c:
        K_i = kernel[np.where(label == i)[0], :]
        l = K_i.shape[0]
        mu_i = np.mean(K_i, axis=0)
        M += l * ((mu_i - mu).T @ (mu_i - mu))
    # compute N
    for i in c:
        K_i = kernel[np.where(label == i)[0], :]
        l = K_i.shape[0]
        N += K_i.T @ (np.eye(l) - (np.ones((l, l), dtype=np.float64) / l)) @ K_i

    eigen_val, eigen_vec = np.linalg.eig(np.linalg.pinv(N) @ M)
    for i in range(eigen_vec.shape[1]):
        eigen_vec[:, i] = eigen_vec[:, i] / np.linalg.norm(eigen_vec[:, i])
    idx = np.argsort(eigen_val)[::-1]
    W = eigen_vec[:, idx][:, :dims].real

    return W
```

t-SNE

part1

There are two places that need to be modified.
First, how to compute Q. The following is the formula used to compute Q in symmetric SNE.(method: ’t’ is t-SNE, “s” is symmetric SNE)

$$q_{j|i} = \frac{\exp(- \mid\mid y_i - y_j \mid\mid^2)}{\sum_{k \neq i} \exp(- \mid\mid y_i - y_k \mid\mid^2)}$$

```python
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Compute pairwise affinities
if method=='t':
    sum_Y = np.sum(np.square(Y), 1)
    num = -2. * np.dot(Y, Y.T)
    num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y))
elif method=="s":
    sum_Y = np.sum(np.square(Y), 1)
    num = -2. * np.dot(Y, Y.T)
    num = np.exp(-1 * np.add(np.add(num, sum_Y).T, sum_Y))
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

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

Second, how to compute gradient. The following is the formula used to compute gradient in symmetric SNE.

$$\frac{\delta C}{\delta y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)$$

```python
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Compute gradient
PQ = P - Q
if method == "t":
    for i in range(n):
        dY[i, :] = np.sum(np.tile(PQ[:, i] * num[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
elif method == "s":
    for i in range(n):
        dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] - Y), axis=0)
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

part2

in for loop, each iterate will do gradient descent to improve y, and each 10 iterate will use “visualization” function to generate .png “visualization” will plot y on 2D space.

```python
# Compute current value of cost function
if (iter + 1) % 10 == 0:
    C = np.sum(P * np.log(P / Q))
    print("Iteration %d: error is %f" % (iter + 1, C))
    visualization(Y,P,Q,iter,perplexity)
```

```python
def visualization(Y, P, Q, iter, perplexity):
    pylab.clf()
    pylab.title('S-SNE_' + str(iter) + ' with perplexity : ' + str(perplexity))
    pylab.scatter(Y[:, 0], Y[:, 1], 20, labels)
    # pylab.show()
    pylab.savefig('./result/Q1/S-SNE' +str(time.time())+ '_' + str(iter) + '_' + str(perplexity) + '.png')
```

part3

Use formula come from the lecture to compute distribution of high-d and low-d pairwise similarity. Then plot it.

use **KL divergence** to measure the **distance b/w distributions of high-d and low-d pairwise similarities**

$$C = \sum_i KL(P_i \,||\, Q_i) = \sum_i \sum_j p_{j|i} \log \frac{p_{j|i}}{q_{j|i}}$$

```
def draw_distribution(P,Q):
    print("Q3")

    pylab.clf()
    pylab.title('distribution of pairwise similarities')

    PI,PJ=np.shape(P)
    ci=np.zeros((PI))
    for i in range(PI):
        for j in range(PJ):
            ci[i] += P[i][j] * np.log(P[i][j]/Q[i][j])
    plt.hist(ci.flatten(),bins=40,log=True)
    pylab.xlabel('Pairwise Similarities')
    pylab.ylabel('Amount')
    plt.show()

    pylab.subplot(2,1,1)
    pylab.title('tSNE high-dim')
    pylab.hist(P.flatten(),bins=40,log=True)
    pylab.subplot(2,1,2)
    pylab.title('tSNE low-dim')
    pylab.hist(Q.flatten(),bins=40,log=True)
    pylab.show()
```

part4

we can change perplexity here.

We can also change using_method here, ”t”=t-SNE,”s”=symmetric SNE.

```
if __name__ == "__main__":


    using_method ="t"
    perplexity = 20

    print("Run Y = tsne.tsne(X, no_dims, perplexity) to perform t-SNE on your dataset.")
    print("Running example on 2,500 MNIST digits...")
    X = np.loadtxt("mnist2500_X.txt")
    labels = np.loadtxt("mnist2500_labels.txt")
    Y = sne(X, 4, 50, perplexity,method=using_method)
    get_gif(pics_dir="./result/Q1")
```

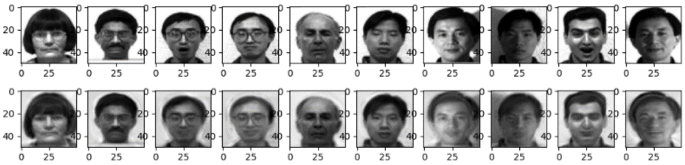
b.experiments settings and results

kernel eigenface

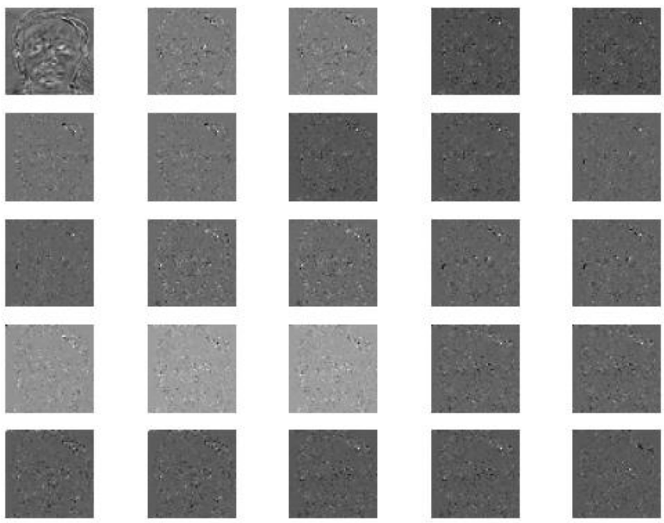
part1

eigenface

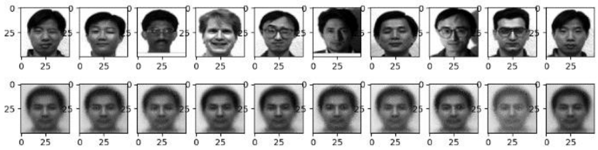
reconstruct by PCA. Upper row is original image, second row is reconstruction image.



Fisherfaces.



reconstruct by LDA Upper row is original image, second row is reconstruction image.



part2

k=3

PCA : accuracy: 0.833 (25/30)

LDA : accuracy: 0.600 (18/30)

part3

Compare part2 and part3 result, Observation:

Use Kernel method doesn't affect PCA performance much.
Use kernel method will improve LDA performance.

kernel PCA has good performance under any kernel.
Kernel LDA has best performance under Polynomial Kernel in my experiments.

k=3

Linear kernel:

Kernel PCA : accuracy: 0.867 (26/30)

Kernel LDA : accuracy: 0.600 (18/30)

Polynomail Kernel: gamma=5, c=10, degree=2

Kernel PCA : accuracy: 0.867 (26/30)

Kernel LDA : accuracy: 0.733 (22/30)

Polynomail Kernel: gamma=0.1, c=100, degree=3

Kernel PCA : accuracy: 0.833 (25/30)

Kernel LDA : accuracy: 0.667 (20/30)

RBF Kernel : g=1e-7

Kernel PCA : accuracy: 0.867 (26/30)

Kernel LDA : accuracy: 0.600 (18/30)

RBF Kernel :   g=1

Kernel PCA : accuracy: 0.867 (26/30)

Kernel LDA : accuracy: 0.600 (18/30)

k=5

Linear kernel:

Kernel PCA : accuracy: 0.867 (26/30)

Kernel LDA : accuracy: 0.633 (19/30)

Polynomail Kernel: gamma=5, c=10, degree=2

Kernel PCA : accuracy: 0.867 (26/30)

Kernel LDA : accuracy: 0.700 (21/30)

Polynomail Kernel: gamma=0.1, c=100, degree=3

Kernel PCA : accuracy: 0.867 (26/30)

Kernel LDA : accuracy: 0.700 (21/30)

RBF Kernel: g = 1e-7

Kernel PCA : accuracy: 0.867 (26/30)

Kernel LDA : accuracy: 0.600 (18/30)

RBF Kernel: g = 1

Kernel PCA : accuracy: 0.867 (26/30)

Kernel LDA : accuracy: 0.600 (18/30)

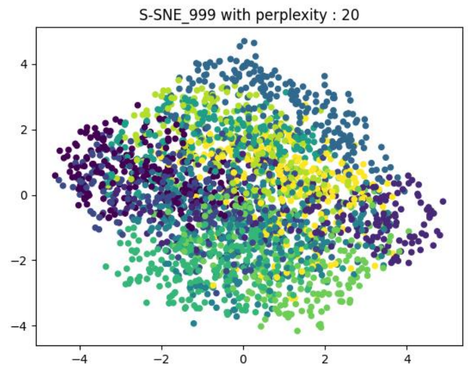
t-SNE

part1 & part2

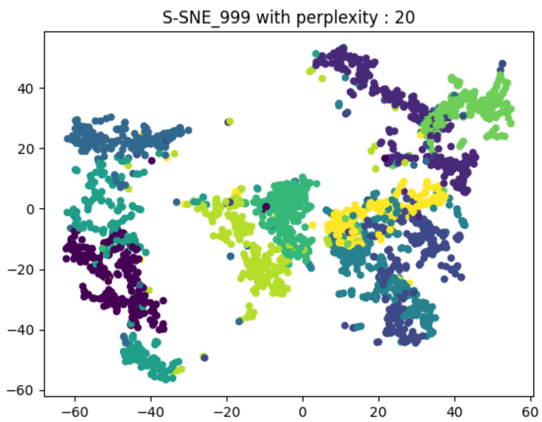
(more gif file is in zip folder)

crowded problem:

we can see that Symmetric SNE cannot clear separate each class.



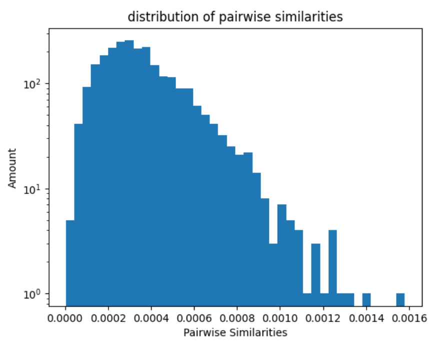
In contrast, t-SNE doesn't have crowed problem.

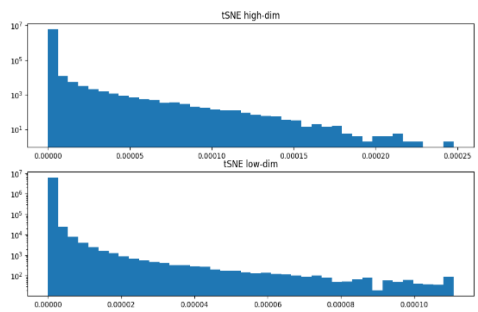


part3

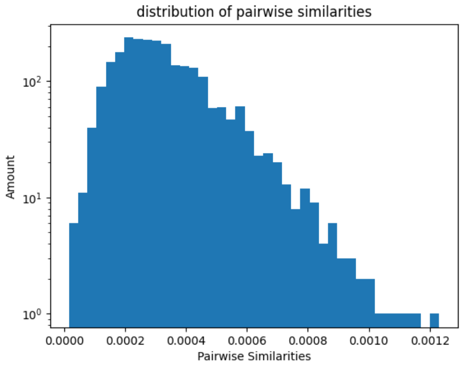
distribution of pair wise similarity.
Right image, after training, we can see P and Q distribution in high-dimension and low-dimension.
Left image, after training, is compute by lecture formula We can see that distribution is mainly on the left, it mean that the information relation between high-dimension and low-dimension is almost the same. $C = \sum_i KL(P_i \,||\, Q_i) = \sum_i \sum_j p_{j|i} \log \frac{p_{j|i}}{q_{j|i}}$

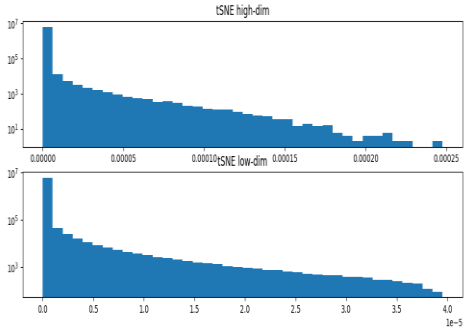
t-SNE





Symmetric SNE

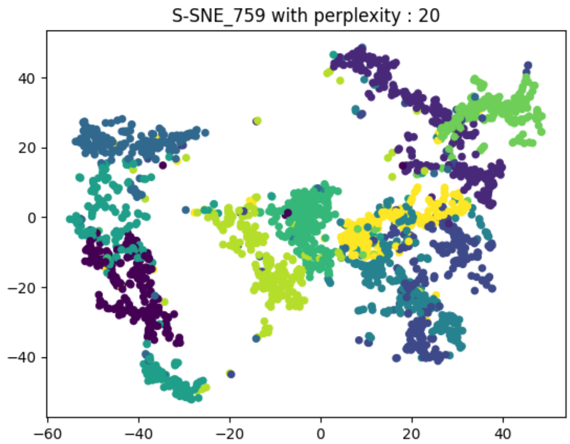




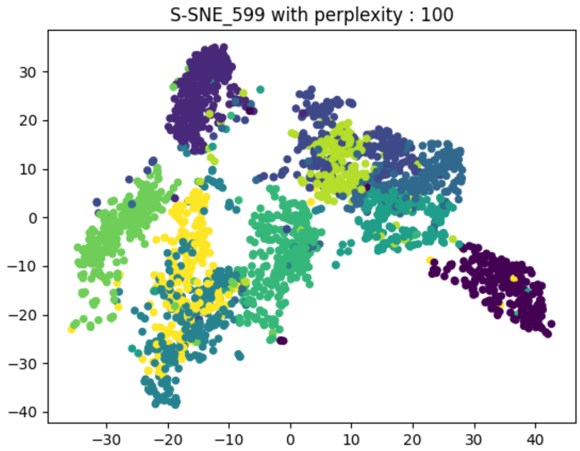
part4

Perplexity will influence the number of neighbors it will care. Larger perplexity will have the less sensitive to a small group of point. Therefore, In small perplexity, it sometimes will spilt one class to two group of point because of high sensitive, and larger perplexity will make same class point closer because of low sensitive.

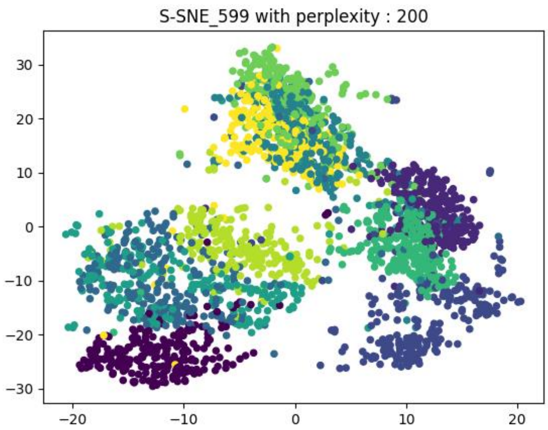
perplexity : 20



perplexity : 100

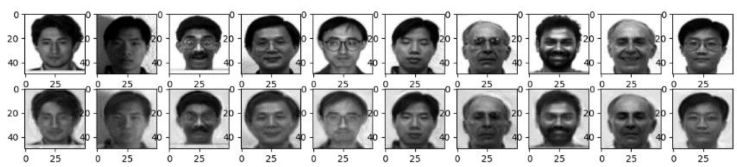


perplexity :200

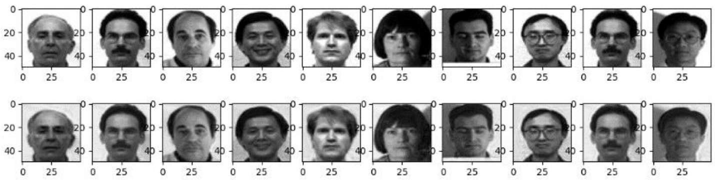


c. observations and discussion

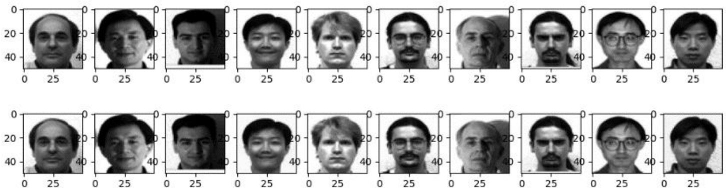
Here I want to discuss abount dimension in eigenface.
In this home work, we get 25 eigenface, which means that we reduce dimension. If original image is 2500 pixel, it mean we reduced the dimension from 2500 to 25.
Now I want to try other dimension, 40, 80, 150.
Dimension = 40



Dimension = 80



Dimension = 150



I found that we descended to dimension 150, it retained more feature than reduce to dimension 25, so it can reconstruct very clear picture. However, its speed become slow.

\