0516312 陳皓婷

Kernel Eigenfaces/Fisherfaces

<Main function>:

```
if __name__ == "__main__":
    train_img, train_label = read_IMG("./Yale_Face_Database/Training",9)
    test_img, test_label = read_IMG("./Yale_Face_Database/Testing",2)
    k_knn = 3
    kernel_modes = [0,"linear","Polynomial","RBF"]
    for kernel_mode in kernel_modes:
        pca_transform, pca_reconstruct, pca_x= pca(train_img,train_label, test_img, test_label, k_knn, kernel_mode)
        lda_transform, lda_reconstruct= lda(pca_transform, pca_x,train_img,train_label, test_img, test_label, k_knn)
```

First, read data from the files.

```
def read_IMG(dirpath,subjects):
    ims = os.listdir(dirpath)
    num_ims = subjects*15
    pixels = [[] for i in range(num_ims)]
    label = [[] for i in range(num_ims)]
    i = 0
    for im in ims:
        pixels[i] = list(chain.from_iterable(imread(dirpath+'/'+im)))
        label[i] = int(i/subjects)
        i += 1
    pixels = np.array(pixels)
    label = np.array(label)
    return pixels, label
```

Then set the order to do PCA, LDA, Kernel PCA, Kernel LDA. And I set the k of knn-method to 3.

<PCA>:

```
def pca(x,y,t_x,t_y, k, kernel_mode):
    mean = np.mean(x, axis=0)
    delt = x-mean
    print("kernel_mode: ",kernel_mode)
    if kernel_mode == 0:
        S = np.cov(x, bias=True)
    else:
        S = kernel(x,kernel_mode)
        one_N = np.ones((x.shape[0],x.shape[0]))/x.shape[0]
        S = S - one_N@S - S@one_N + one_N@S@one_N
    eigenValues, eigenVectors = np.linalg.eig(S)
    ### sorted largest->smallest ###
    eigenValues = np.abs(eigenValues)
    idx = np.argsort(eigenValues)[::-1]
    eigenValues = eigenValues[idx]
    eigenVectors = eigenVectors[:,idx]
    # count important ones
    total_Val = np.sum(eigenValues)
    sumup = 0
    for i in range(len(idx)):
        sumup += eigenValues[i]
        if sumup/total_Val >= 0.97: # set threshold = 97%
            eigenValues = eigenValues[0:i]
            eigenVectors = eigenVectors[:,:i]
            break
```

First, compute the covariance to gather eigenvalues and eigenvector. (kernel_mode==0: simple PCA)

Then, sort the eigenvalues and get sorted eigenvectors. Setting threshold = 97% is able to get the most important(97%) eigenvectors.

```
    transform = delt.T@eigenVectors
    transform = np.real(transform)
    if kernel_mode == 0:
        showout_transform(transform)

    reconstruct = transform@(transform.T@delt.T) + mean.reshape(-1,1)
    if kernel_mode == 0:
        showout_reconstruct(x, reconstruct)

    t_z = transform.T@(t_x-mean).T   # projected faces (low_dim)
    z = transform.T@delt.T           # projected faces (low_dim)
    dist = np.zeros(x.shape[0])          # size = 135
    predicted_y = np.zeros(t_x.shape[0]) # size = 30

    for i in range(predicted_y.size):   # 0-30
        for j in range(dist.size):      # 0-135
            dist[j] = cdist(t_z[:,i].reshape(1,-1),z[:,j].reshape(1,-1),'sqeuclidean')
        nn = y[np.argsort(dist)[:k]] # closest k-labels
        sort_uniq_nn, sort_nn_appearcounts = np.unique(nn, return_counts=True)
        idx_pred = np.argmax(sort_nn_appearcounts)
        predicted_y[i] = sort_uniq_nn[idx_pred]

    error_rate=len(np.argwhere(t_y-predicted_y))/predicted_y.size
    print("predicted_y:",predicted_y)
    if kernel_mode == 0:
        print("pca accuarcy rate: ", 1.-error_rate)
    else:
        print("kpca accuarcy rate: ", 1.-error_rate)

    return transform, reconstruct, z
```

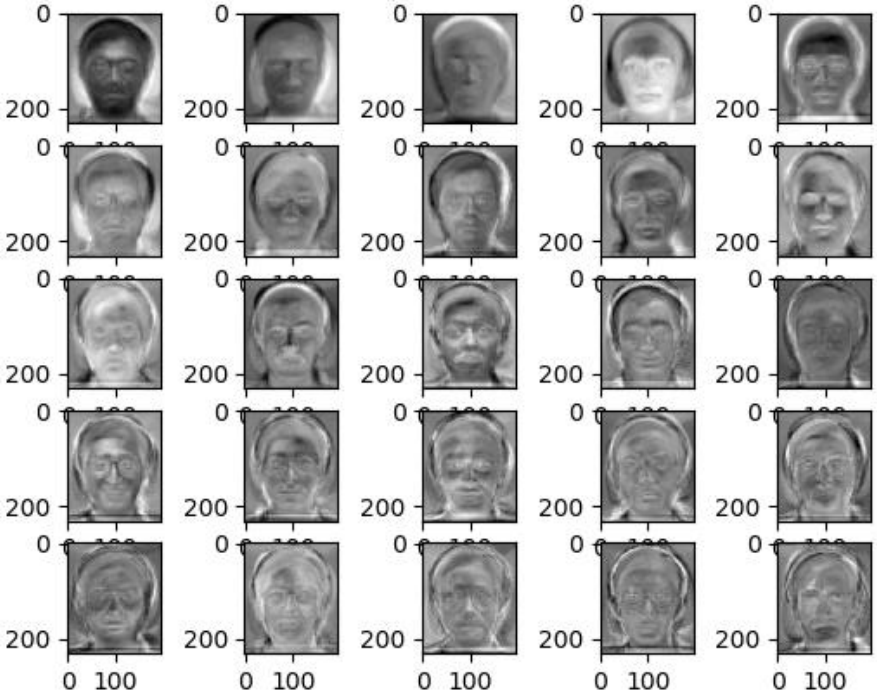
Third, get transform matrix by mixing difference from images to means and eigenvectors. Then we get the eigenfaces and show them by “showout_transform” function.

Fourth, do reconstruct by mixing transform matrix and difference from images to means and adding the mean in origin space. Then we get the reconstructed faces and show them by “showout_reconstruct” function.

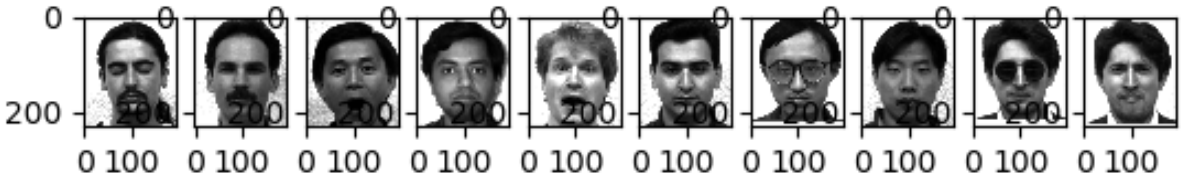
Fifth, count the performance of PCA: compare the distance of projected-testdata and projected-traindata, seek out the k-nearest-neighbors (knn), find the one who appears most times, pick that one as the predicted label and compute the accuracy rate.

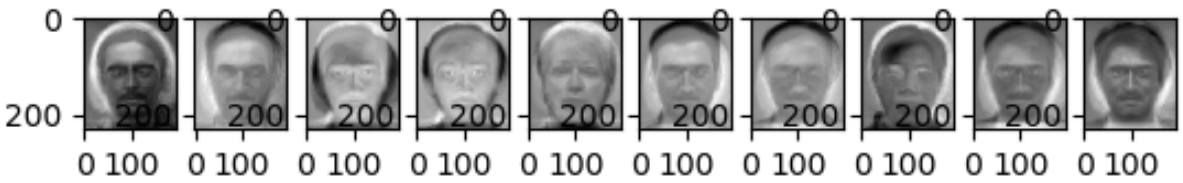
```
def showout_transform(transform):
    # eigenface
    for i in range(25):
        plt.subplot(5,5,i+1)
        plt.imshow(transform[:,i].reshape(231,195), cmap="gray")
    plt.show()

def showout_reconstruct(x, reconstruct):
    # random pick 10
    fig, axes = plt.subplots(2, 10)
    idx = np.random.choice(135, 10, replace=False)
    for i, random_idx in enumerate((idx)):
        axes[0, i].imshow(x[random_idx].reshape(231,195), cmap="gray")
        axes[1, i].imshow(reconstruct[:,random_idx].reshape(231,195), cmap="gray")
    plt.show()
```

First 25 eigenfaces:



Random pick 10 images to show the reconstruction:



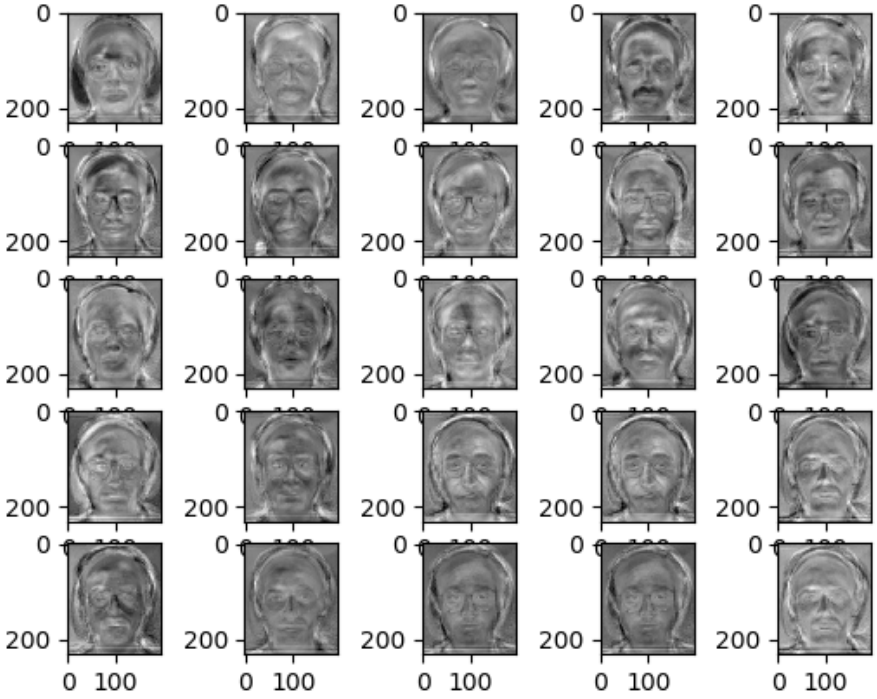


<LDA>:

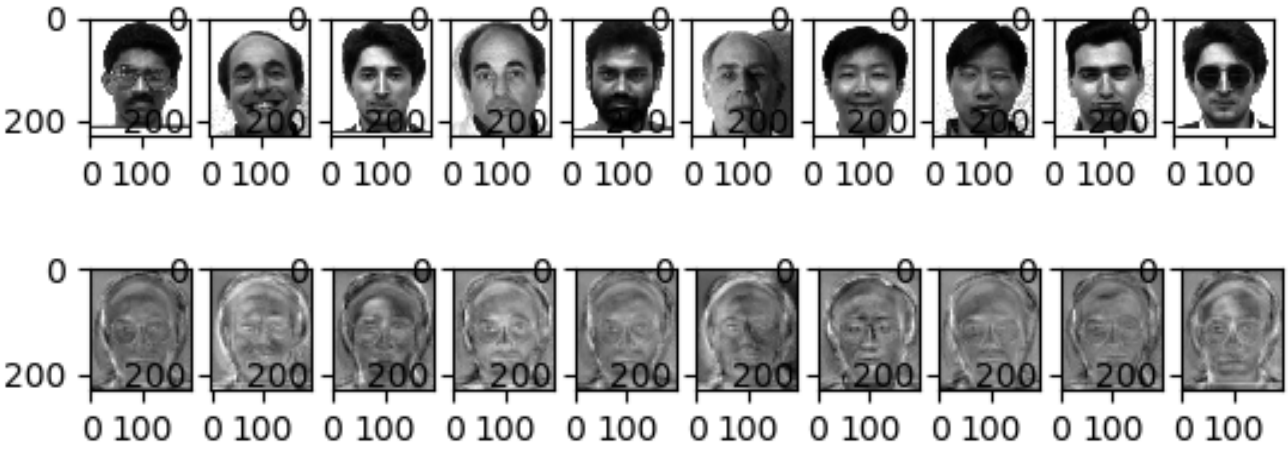
```
def lda(pca_transform,pca_x,x,y,t_x,t_y, k):
    # mean function
    allmean = np.mean(pca_x, axis=1) # (50,)

    # within class scatter
    within_class_S = np.zeros((pca_x.shape[0],pca_x.shape[0]))
    for i in range(15):
        within_class_S += np.cov(pca_x[:,i*9:i*9+9], bias=True)

    # in between class scatter
    in_btw_class_S = np.zeros((pca_x.shape[0],pca_x.shape[0]))
    for i in range(15):
        cl_mean = np.mean(pca_x.T[i*9:i*9+9,:], axis=0)
        in_btw_class_S += 9*(cl_mean-allmean).reshape(-1,1)@(cl_mean-allmean).reshape(1,-1)

    S = np.linalg.inv(within_class_S)@in_btw_class_S
    eigenValues, eigenVectors = np.linalg.eig(S)
    ### sorted largest->smallest ###
    eigenValues = np.abs(eigenValues)
    idx = np.argsort(eigenValues)[::-1]
    eigenValues = eigenValues[idx]
    eigenVectors = eigenVectors[:,idx]
    # count important ones
    total_Val = np.sum(eigenValues)
    sumup = 0
    for i in range(len(idx)):
        sumup += eigenValues[i]
        if sumup/total_Val >= 0.97: # set threshold = 99%
            if i<25:
                i=25
            eigenValues = eigenValues[0:i]
            eigenVectors = eigenVectors[:,:i]
            break

    transform = pca_transform@eigenVectors
    transform = np.real(transform)
    if kernel_mode == 0:
        showout_transform(transform)

    mean = np.mean(x, axis=0)
    delt = x-mean
    reconstruct = transform@(transform.T@delt.T) + mean.reshape(-1,1)
    if kernel_mode == 0:
        showout_reconstruct(x, reconstruct)

    t_z = transform.T@(t_x-mean).T   #  projected faces (low_dim)
    z = transform.T@delt.T           #  projected faces (low_dim)
    dist = np.zeros(x.shape[0])          # size = 135
    predicted_y = np.zeros(t_x.shape[0]) # size = 30

    for i in range(predicted_y.size):   # 0-30
        for j in range(dist.size):      # 0-135
            dist[j] = cdist(t_z[:,i].reshape(1,-1),z[:,j].reshape(1,-1),'sqeuclidean')
        nn = y[np.argsort(dist)[:k]] # closest k-labels
        sort_uniq_nn, sort_nn_appearcounts = np.unique(nn, return_counts=True)
        idx_pred = np.argmax(sort_nn_appearcounts)
        predicted_y[i] = sort_uniq_nn[idx_pred]

    error_rate=len(np.argwhere(t_y-predicted_y))/predicted_y.size
    print("predicted_y: ",predicted_y)
    if kernel_mode == 0:
        print("lda accuarcy rate: ", 1.-error_rate)
    else:
        print("klda accuarcy rate: ", 1.-error_rate)

    return transform, reconstruct
```

First, compute the within-class-scatter and the between-class-scatter from the pca_x (reduce-dimension-data).
Then, get the eigenvalues and eigenvectors based on mixing within-class-scatter and between-class-scatter.
Third, do the same things in PCA to get the most important eigenvectors.
Fourth, compute transform matrix to show the fisherfaces and compute reconstruction matrix to show the reconstructed faces.
Finally, compute the accuracy rate based on the distance of projected-testdata and projected-traindata and doing knn-method.

First 25 Fisherfaces:



Random pick 10 images to show the reconstruction:



<Kernel>:

```
def kernel(x,kernel_mode):
    if kernel_mode == "linear":
        return x@x.T
    elif kernel_mode == "Polynomial":
        g = 0.01
        c = 0
        d = 3
        return (g*x@x.T+c)**d
    elif kernel_mode == "RBF":
        g = 0.01
        return np.exp(-g*cdist(x,x,'sqeuclidean'))
```

In main function, we make the order to compute three kinds of kernel function – linear, polynomial and RBF.

```
kernel_modes = [0,"linear","Polynomial","RBF"]
for kernel_mode in kernel_modes:
    pca_transform, pca_reconstruct, pca_x= pca(train_img,train_label, test_img, test_label, k_knn, kernel_mode)
    lda_transform, lda_reconstruct= lda(pca_transform, pca_x,train_img,train_label, test_img, test_label, k_knn)
```

When we want kernel PCA, we replace the covariance with the output of kernel function subtracted some items. And In LDA, also use PCA-transformed-matrix maked from Kernel function to compute the result.

$$K^C = K - 1_N K - K 1_N + 1_N K 1_N$$

```
if kernel_mode == 0:
    S = np.cov(x, bias=True)
else:
    S = kernel(x,kernel_mode)
    one_N = np.ones((x.shape[0],x.shape[0]))/x.shape[0]
    S = S - one_N@S - S@one_N + one_N@S@one_N
```

Result:

```
kernel_mode:  0
predicted_y: [ 0.  1.  1.  1.  2.  2.  3.  0.  4.  4.  5.  5.  0.  6.  7.  7.  8.  8.
  9.  9. 10. 10.  3. 11. 12.  6. 13. 13. 14. 14.]
pca accuarcy rate:  0.8333333333333334
predicted_y:  [ 0.  2.  1.  1.  2.  2.  3.  0.  4.  4.  5.  5.  4.  6.  7.  7.  8.  8.
  9.  9. 10. 10.  3. 11. 12. 12. 13. 13. 14. 14.]
lda accuarcy rate:  0.8666666666666667
kernel_mode:  linear
predicted_y: [ 0.  1.  1.  1.  2.  2.  3.  0. 11.  4.  5.  5.  0.  6.  7.  7.  8.  8.
  9.  9. 10. 10.  3. 11. 12.  2. 13. 13. 14. 14.]
kpca accuarcy rate:  0.8
predicted_y:  [ 0.  4.  1.  1.  2.  2.  3.  0.  4.  4.  5.  5.  6.  6.  7.  7.  8.  8.
  9.  9. 10. 10.  3. 11. 12. 12. 13. 13. 14. 14.]
klda accuarcy rate:  0.9
kernel_mode:  Polynomial
predicted_y: [ 0.  1.  1.  1.  2.  2.  3.  0.  4.  4.  5.  5.  0.  6.  7.  7.  8.  8.
  9.  9. 10. 10.  3. 11. 12.  2. 13. 13. 14. 14.]
kpca accuarcy rate:  0.8333333333333334
predicted_y:  [ 0.  2.  1.  1.  2.  2.  3.  0.  4.  4.  5.  5.  2.  6.  7.  7.  8.  8.
  9.  9. 10. 10.  3. 11. 12. 12. 13. 13. 14. 14.]
klda accuarcy rate:  0.8666666666666667
kernel_mode:  RBF
predicted_y: [ 0.  1.  1.  1.  2.  2.  3.  0.  4.  4.  0.  5.  0.  6.  8.  7.  8.  8.
  9.  9. 10. 10.  3. 11. 12.  6. 13. 13. 14. 14.]
kpca accuarcy rate:  0.7666666666666666
predicted_y:  [ 0.  3.  1.  1.  2.  2.  3.  4. 12.  4.  5.  5.  4.  8. 13.  3. 12.  3.
  9.  5. 10. 10.  3. 11.  8.  4. 13. 13.  6.  2.]
klda accuarcy rate:  0.5
```

Discussion:

- From the result, we can observe that the performance in simple mode, linear-kernel , Polynomial-kernel is great. And the performance of RBF-kernel mode isn't so good.
- And compare to PCA, LDA performs better except in RBF kernel. I think that may be the problem of choosing proper parameters in RBF kernel.

t-SNE

from t-SNE to Symmetric SNE, I modify two parts:

Compute pairwise affinities

t-SNE: `num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y))`

s-SNE: `num = np.exp(-1. * np.add(np.add(num, sum_Y).T, sum_Y))`

Compute gradient

t-SNE: `dY[i, :] = np.sum(np.tile(PQ[:, i] * num[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)`

s-SNE: `dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)`

In t-SNE/s-SNE, I add some codes to save the images of every-10-iterations which are used to make GIF.

```
if iter%10 ==0:
    pylab.clf()
    pylab.xlim([-15,15])
    pylab.ylim([-15,15])
    pylab.scatter(Y[:, 0], Y[:, 1], 20, labels)
    pylab.savefig(os.path.join(output_dir,"ssne_{}.png".format(int(iter/10))))
```

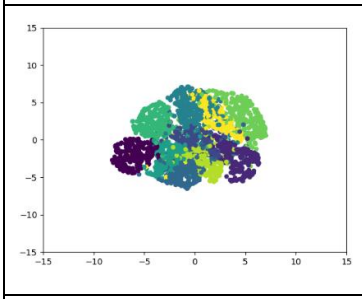
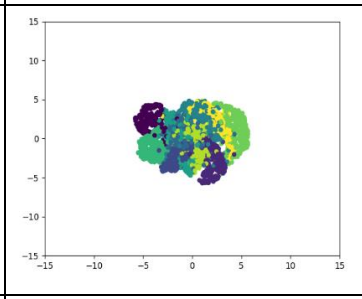
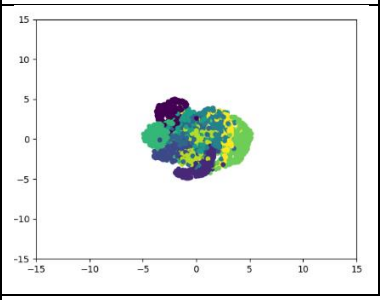
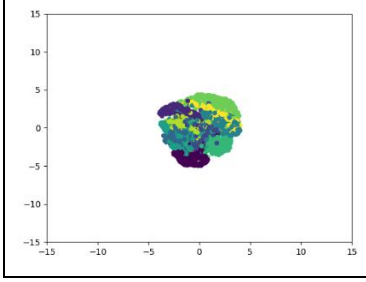
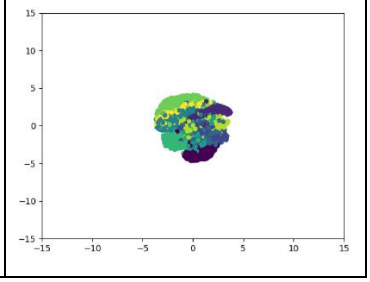
```
if iter%10 ==0:
    pylab.clf()
    pylab.xlim([-120,100])
    pylab.ylim([-100,120])
    pylab.scatter(Y[:, 0], Y[:, 1], 20, labels)
    pylab.savefig(os.path.join(output_dir,"tsne_{}.png".format(int(iter/10))))
```

After doing t-SNE/s-SNE, I use "printSimilarity" function to show the distribution of pairwise similarities in low and high dimensional space and "saveGIF" function to save the procedure.

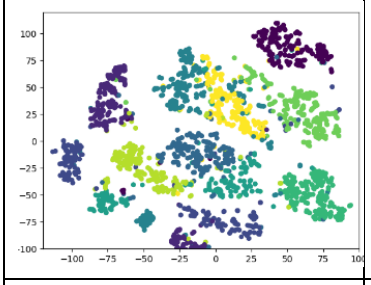
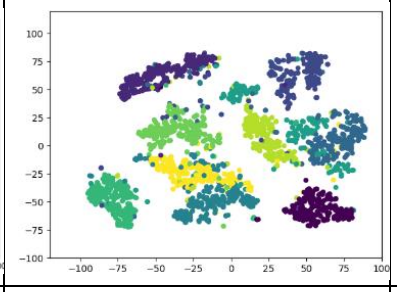
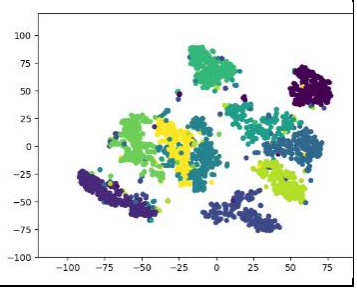
```
def printSimilarity(P,Q,output_dir):
    pylab.clf()
    plt.subplot(2,1,1)
    pylab.hist(P.flatten(),bins=40,log=True)
    plt.subplot(2,1,2)
    pylab.hist(Q.flatten(),bins=40,log=True)
    plt.savefig(output_dir+"_similarity.png")
```

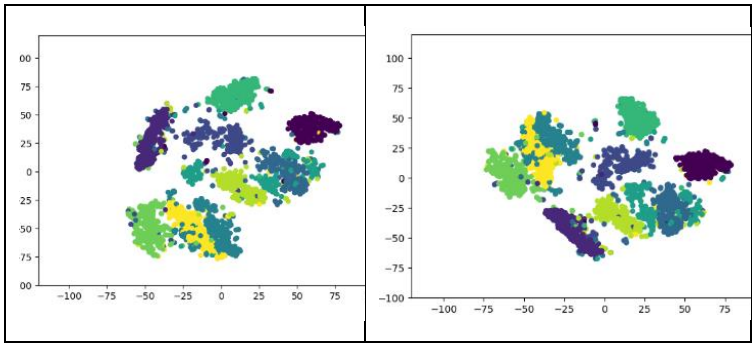
```python
def saveGIF(output_dir,mode):
    gifs = []
    if mode == 1:
        for i in range(100):
            gifs.append(Image.open(os.path.join(output_dir,"tsne_{}.png".format(i))))
    else:
        for i in range(100):
            gifs.append(Image.open(os.path.join(output_dir,"ssne_{}.png".format(i))))
    gifs[0].save(output_dir+'.gif', format='GIF',
                 append_images=gifs[1:],
                 save_all=True,
                 duration=300, loop=0)
```

Symmetric SNE after 1000 iterations.

Perplexity = 10	Perplexity = 20	Perplexity = 30
		
Perplexity = 40	**Perplexity = 50**	
		

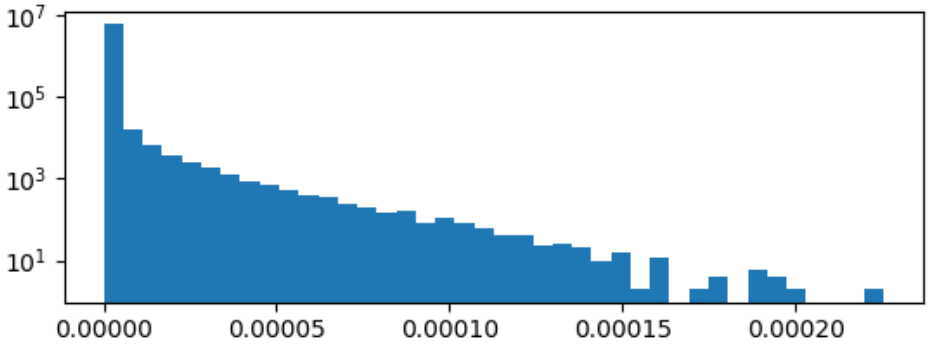
t-SNE after 1000 iterations.

Perplexity = 10	Perplexity = 20	Perplexity = 30
		
Perplexity = 40	**Perplexity = 50**	

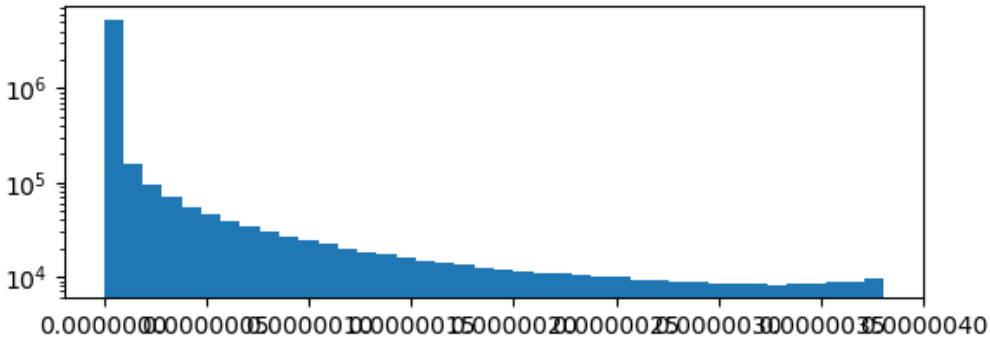


Symmetric SNE:

The pairwise similarity in high dimensional space (perplexity = 30)

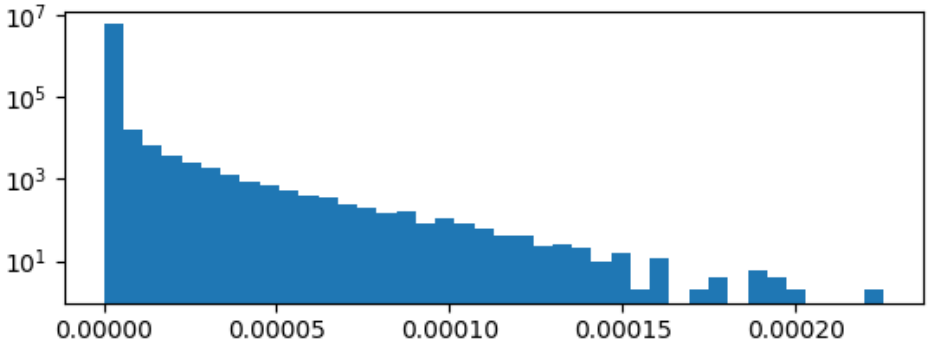


The pairwise similarity in low dimensional space (perplexity = 30)

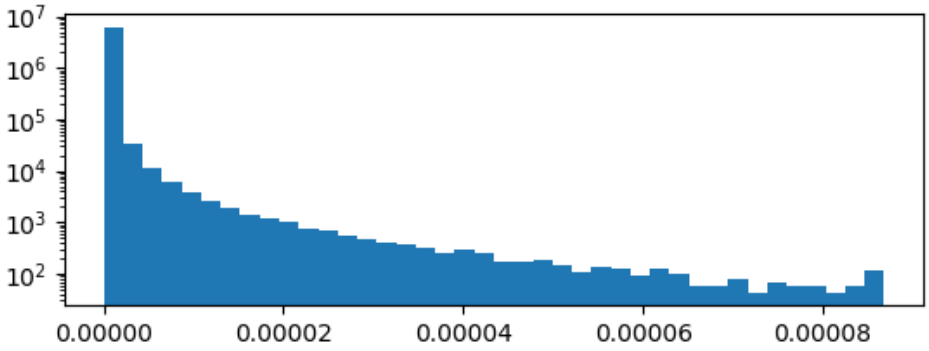


t-SNE:

The pairwise similarity in high dimensional space (perplexity = 30)



The pairwise similarity in low dimensional space (perplexity = 30)



Discussion:

- From the distribution of the pairwise similarity, we can see in low dimensional space, the range of distribution of t-SNE is much larger than Symmetric SNE So we can know the crowed problem in Symmetric SNE, and t-SNE don't have.
- From the scatter plot, we can observe that when perplexity becomes larger, the distribution becomes more concentrated, which makes Symmetric SNE more crowded.
- And, also, we can observe that in t-SNE, when perplexity becomes larger, the decision boundaries becomes more obvious.