Machine Learning HW7

Code

Kernel Eigenfaces

Part1 – simple PCA and LDA

The gaol of PCA is to find an orthogonal projection W in which the data x after projection $z = Wx$ will have **maximum vatiance**, i.e. **minimum mean square error(MSE)**. And W is composed of k first largest eigenvectors of covariance matrix of x, so the below code is find all eigenvectors of x.

```
def simple_pca(n_image, images):
    ### compute variance
    image_transpose = images.T
    mean = np.mean(image_transpose, axis=1)
    mean = np.tile(mean.T, (n_image, 1)).T
    difference = image_transpose - mean
    covariance = difference.dot(difference.T) / n_image

    return covariance
```

Next, record our goal is find the W, here we take $k = 25$ and find the biggest 25 eigenvalues and eigenvectors respectively.

```
def find_eigenvector(matrix):
    eigenvalues, eigenvectors = np.linalg.eig(matrix)

    ### find the biggest 25 eigenvectors
    target_idx = np.argsort(eigenvalues)[::-1][:25]
    target_eigenvectors = eigenvectors[:, target_idx].real

    return target_eigenvectors
```

Last, visualize the result and use the k nearest neighbors to decide which the testing image should belomg to. Note that the classify is doing on the component space.

隱私權 - 條款

```python
    ### find the biggest 25 eigenvectors
    target_eigenvectors = find_eigenvector(matrix)

    ### transform eigenvectors to eigenface
    eigenface(target_eigenvectors, 0)

    ### randomly recontruct 10 eigenface
    construct_face(n_train, train_images, target_eigenvectors)

    ### classify
    classify(n_train, len(test_images), train_images, train_labels, test_images, test_labels, target_eigenvectors, k_neighbors)

    ### show the result
    plt.tight_layout()
    plt.show()
def decorrelate(n_image, images, eigenvectors):
    ### decorrelate original images to components space
    decorrlated_image = np.zeros((n_image, 25))

    for idx, image in enumerate(images):
        decorrlated_image[idx, :] = image.dot(eigenvectors)

    return decorrlated_image


def classify(n_train, n_test, train_images, train_labels, test_images, test_labels, eigenvectors, k_neighbors):
    decorrelate_train = decorrelate(n_train, train_images, eigenvectors)
    decorrelate_test = decorrelate(n_test, test_images, eigenvectors)

    error = 0

    distance = np.zeros(n_train)

    for test_idx, test_image in enumerate(decorrelate_test):
        for train_idx, train_image in enumerate(decorrelate_train):
            distance[train_idx] = np.linalg.norm(test_image - train_image)

        min_distance = np.argsort(distance)[:k_neighbors]

        predict = np.argmax(np.bincount(train_labels[min_distance]))
        if predict != test_labels[test_idx]:
            error += 1
    print(f'Error count: {error}\nError rate: {float(error) / n_test}')
```

The second method is LDA, like the PCA, we also want to find the matrix W, but use different way. Because it is supervised, we know all data points and their class, so it is possible to compute the distance between class and within the class center. If now we are dealing with multi-class cases $(C_1, C_2, \ldots, C_k)$

The distance between class scatter:

$$\textit{between-class scatter}:$$

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

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

The distance within class scatter:

$$\textit{within-class scatter}:\ S_W = \sum_{j=1}^{k} S_j, \text{ where } S_j = \sum_{i \in C_j}(x_i - \mathbf{m}_j)(x_i - \mathbf{m}_j)^\top$$

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

And use above two matrix to find W with the q largest eignvectors

$$\text{get first } q \text{ largest eigenvectors of } S_W^{-1}S_B \text{ as } W$$

Same as PCA, we first find the matrix and use it eigenvectors to compute W, follows above formula.

```
def simple_lda(num_of_each_class, images, labels):
    ### get overall mean
    overall_mean = np.mean(images, axis=0)

    ### mean of each class
    n_class = len(num_of_each_class)
    class_mean = np.zeros((n_class, 29 * 24))
    for label in range(n_class):
        class_mean[label, :] = np.mean(images[labels == label + 1], axis=0)

    ### get between class scatter
    scatter_b = np.zeros((29 * 24, 29 * 24), dtype=float)
    for idx, num in enumerate(num_of_each_class):
        difference = (class_mean[idx] - overall_mean).reshape((29 * 24, 1))
        scatter_b += num * difference.dot(difference.T)

    ### get within class scatter
    scatter_w = np.zeros((29 * 24, 29 * 24), dtype=float)
    for idx, mean in enumerate(class_mean):
        difference = images[labels == idx + 1] - mean
        scatter_w += difference.T.dot(difference)

    ### get Sw^(-1) * Sb
    matrix = np.linalg.pinv(scatter_w).dot(scatter_b)

    return matrix
```

Similarly, take $q = 25$, which means that we take the 25 largest eigenvectors to compose W, and visualize the result at the end. The code will be

```
### find the biggest 25 eigenvectors
target_eigenvectors = find_eigenvector(matrix)

### transform eigenvectors to eigenface
eigenface(target_eigenvectors, 0)

### randomly recontruct 10 eigenface
construct_face(n_train, train_images, target_eigenvectors)

### classify
classify(n_train, len(test_images), train_images, train_labels, test_images, test_labels, target_eigenvectors, k_neighbors)

### show the result
plt.tight_layout()
plt.show()
```

Part2 – Face Recognition

To reconstruct the face, we follow the formula

$xWW^\top$ **linear combination of principal components which tries to reconstruct original x**

```
def construct_face(n_image, images, eigenvectors):
    ### construct 10 eigenfaces randomly

    reconstruct_image = np.zeros((10, 29 * 24))

    choice = np.random.choice(n_image, 10)
    for idx in range(10):
        reconstruct_image[idx, :] = images[choice[idx], :].dot(eigenvectors).dot(eigenvectors.T)

    fig = plt.figure(2)
    fig.canvas.set_window_title(f'Reconstructed faces')
    for idx in range(10):
        ### original image
        plt.subplot(10, 2, idx * 2 + 1)
        plt.axis('off')
        plt.imshow(images[choice[idx], :].reshape(29, 24), cmap='gray')

        ### reconstructed image
        plt.subplot(10, 2, idx * 2 + 2)
        plt.axis('off')
        plt.imshow(reconstruct_image[idx, :].reshape(29, 24), cmap='gray')
```

Part3 – kernel PCA and LDA

The main process of using kernel method is similar to which we introduce previous. But the way of build the matrix we want to do eigendecomposition is different. For thr PCA, the matrix will become

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

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

where $K_{ij} = \Phi(x_i)\Phi(x_j)$

```
def kernal_pca(images, kernel_type, gamma):
    ### linear kernel:0 or RBF kernel:1
    if kernel_type == 'linear':
        ### linear kernel
        kernel = images.T.dot(images)
    elif kernel_type == 'rbf':
        ### RBF kernel
        kernel = np.exp(-gamma * cdist(images.T, images.T, 'sqeuclidean'))
    else:
        raise BaseException(f'Invalid kernel type. The kernel type should be linear or rbf')

    ### get centered kernel
    matrix_n = np.ones((29 * 24, 29 * 24), dtype=float) / (29 * 24)
    matrix = kernel - matrix_n.dot(kernel) - kernel.dot(matrix_n) + matrix_n.dot(kernel).dot(matrix_n)

    return matrix
```

Then, the kernel LDA use the below formula:

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

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

```
def kernel_lda(num_of_each_class, images, labels, kernel_type, gamma):
    n_class = len(num_of_each_class)
    n_image = len(images)

    if kernel_type == 'linear':
        ### Linear
        kernel_of_each_class = np.zeros((n_class, 29 * 24, 29 * 24))
        for idx in range(n_class):
            image = images[labels == idx + 1]
            kernel_of_each_class[idx] = image.T.dot(image)
        kernel_of_all = images.T.dot(images)
    elif kernel_type == 'rbf':
        ### RBF
        kernel_of_each_class = np.zeros((n_class, 29 * 24, 29 * 24))
        for idx in range(n_class):
            image = images[labels == idx + 1]
            kernel_of_each_class[idx] = np.exp(-gamma * cdist(image.T, image.T,
'sqeuclidean'))
        kernel_of_all = np.exp(-gamma * cdist(images.T, images.T, 'sqeuclidean'))
    else:
        raise BaseException(f'Invalid kernel type. The kernel type should be linear or
rbf')

    ### compute matrix_n
    matrix_n = np.zeros((29 * 24, 29 * 24))
    identity_matrix = np.eye(29 * 24)
    for idx, num in enumerate(num_of_each_class):
        matrix_n += kernel_of_each_class[idx].dot(identity_matrix - num *
identity_matrix).dot(kernel_of_each_class[idx].T)

    ### coompute matrix_m
    matrix_m_i = np.zeros((n_class, 29 * 24))
    for idx, kernel in enumerate(kernel_of_each_class):
        for row_idx, row in enumerate(kernel):
            matrix_m_i[idx, row_idx] = np.sum(row) / num_of_each_class[idx]

    matrix_m_star = np.zeros(29 * 24)
    for idx, row in enumerate(kernel_of_all):
        matrix_m_star[idx] = np.sum(row) / n_image

    matrix_m = np.zeros((29 * 24, 29 * 24))
    for idx, num in enumerate(num_of_each_class):
        difference = (matrix_m_i[idx] - matrix_m_star).reshape((29 * 24, 1))
        matrix_m += num * difference.dot(difference.T)

    ### get N^(-1) * M
    matrix = np.linalg.pinv(matrix_n).dot(matrix_m)

    return matrix
```

t-SNE

There are a little difference between t-SNE and s-SNE in their joint probability distribution in the low dimension:

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

t-SNE: $q_{ij} = \dfrac{(1+ \| y_i - y_j \|^2)^{-1}}{\sum_{k \neq l} (1+ \| y_i - y_j \|^2)^{-1}}$

Therefore, we need to modify this part to get the s-SNE in the sample code. The gradient computation also need to change because the distribution is changed.

```
# Compute pairwise affinities
sum_Y = np.sum(np.square(Y), 1)
num = -2. * np.dot(Y, Y.T)
if mode == 'tSNE':
    num = 1. / (1. + np.add(np.add(num, sum_Y).T, sum_Y))
elif mode == 'sSNE':
    num = np.exp(-1 * np.add(np.add(num, sum_Y).T, sum_Y))
else:
    raise BaseException(f'Invalid mode. The mode should be tSNE or sSNE(symmetricSNE)')
```

```
# Compute gradient
PQ = P - Q
if mode == 'tSNE':
    for i in range(n):
        dY[i, :] = np.sum(np.tile(PQ[:, i] * num[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
elif mode == 'sSNE':
    for i in range(n):
        dY[i, :] = np.sum(np.tile(PQ[:, i], (no_dims, 1)).T * (Y[i, :] - Y), 0)
else:
    raise BaseException(f'Invalid mode. The mode should be tSNE or sSNE(symmetricSNE)')
```

And add another code to record outputs for this part:

Make GIF

The below code record the process of every 10 iterations. Using these pictures to make the gif

```
def capture_current_state(Y, labels, mode, perplexity):
    plt.clf()
    plt.scatter(Y[:, 0], Y[:, 1], 20, labels)
    plt.title(f'{"t-SNE" if mode == "tSNE" else "symmetric-SNE"}, perplexity = {perplexity}')
    plt.tight_layout()
    canvas = plt.get_current_fig_manager().canvas
    canvas.draw()

    return Image.frombytes('RGB', canvas.get_width_height(), canvas.tostring_rgb())
```

(This picture show the other function call

`capture_current_state()`)

```
    ### save as gif
    filename = f'./output/{"t-SNE" if mode == "tSNE" else "symmetric-SNE"}_{perplexity}.gif'
    os.makedirs(os.path.dirname(filename), exist_ok=True)
    img[0].save(filename, save_all=True, append_images=img[1:], optimize=False, loop=0, duration=200)
```

At the end, show the graph of the probability distribution in high and low dimension respectively.

```
def draw_similarities(P, Q, labels):
    index = np.argsort(labels)
    plt.clf()
    plt.figure(1)

    ### plot P
    logP = np.log(P)
    sorted_P = logP[index][:, index]
    plt.subplot(121)
    img = plt.imshow(sorted_P, cmap='gray', vmin=np.min(logP), vmax=np.max(logP))
    plt.colorbar(img)
    plt.title(f'High dimensional space')

    ### plot Q
    logQ = np.log(Q)
    sorted_Q = logQ[index][:, index]
    plt.subplot(122)
    img = plt.imshow(sorted_Q, cmap='gray', vmin=np.min(logQ), vmax=np.max(logQ))
    plt.colorbar(img)
    plt.title(f'Low dimensional space')

    plt.tight_layout()
```

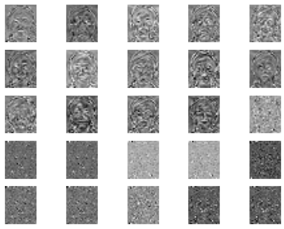
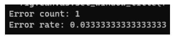
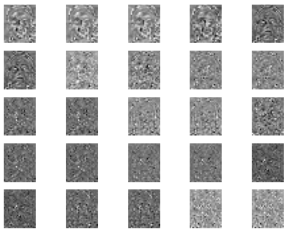
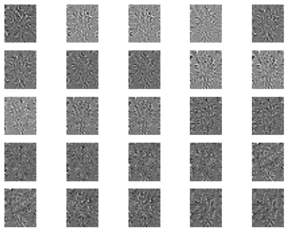
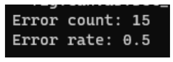
Experiment settings and results

Kernel Eigenfaces

PCA

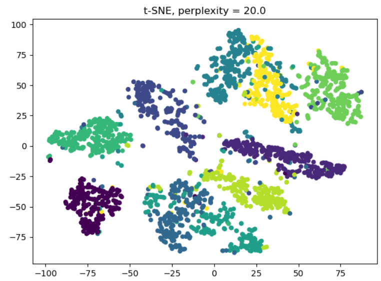
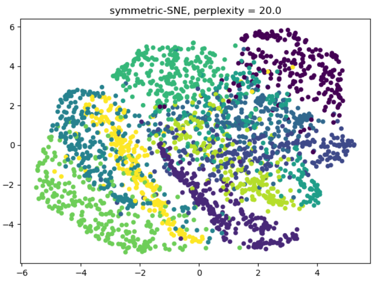
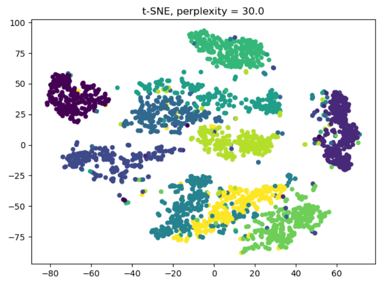
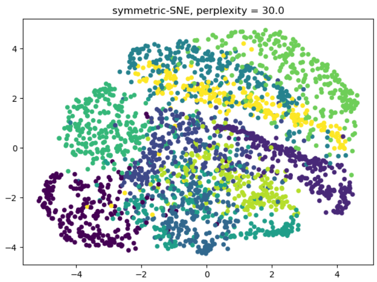
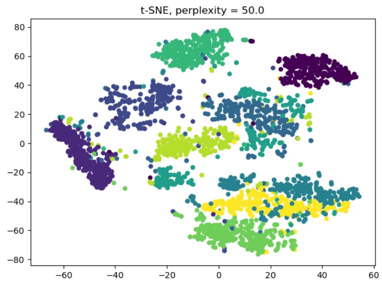
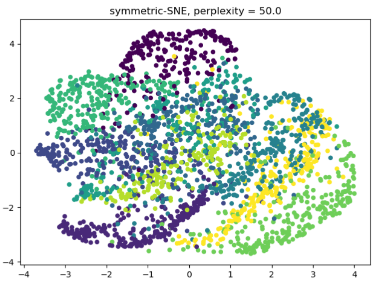
	Eigenfaces	Reconstructed faces	Error Rate
Simple			Error count: 4 Error rate: 0.13333333333333333
Linear kernel			Error count: 4 Error rate: 0.13333333333333333
RBF kernel			Error count: 4 Error rate: 0.13333333333333333

LDA

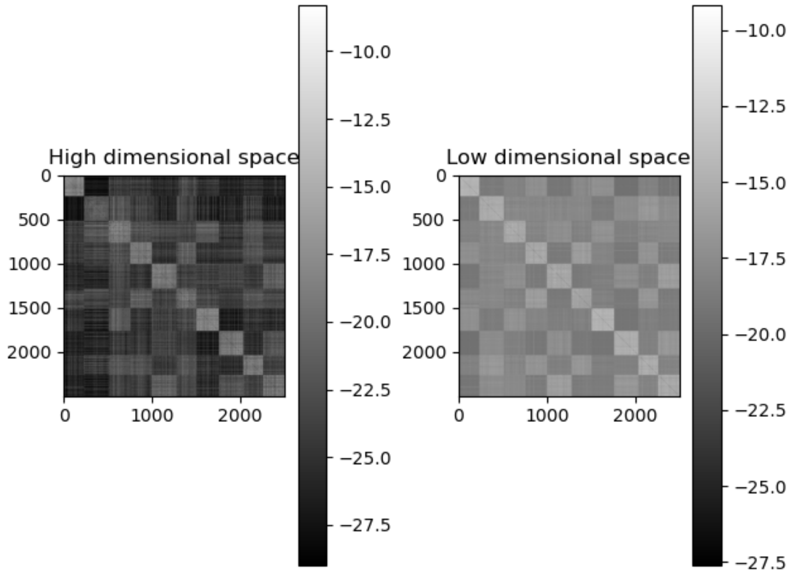
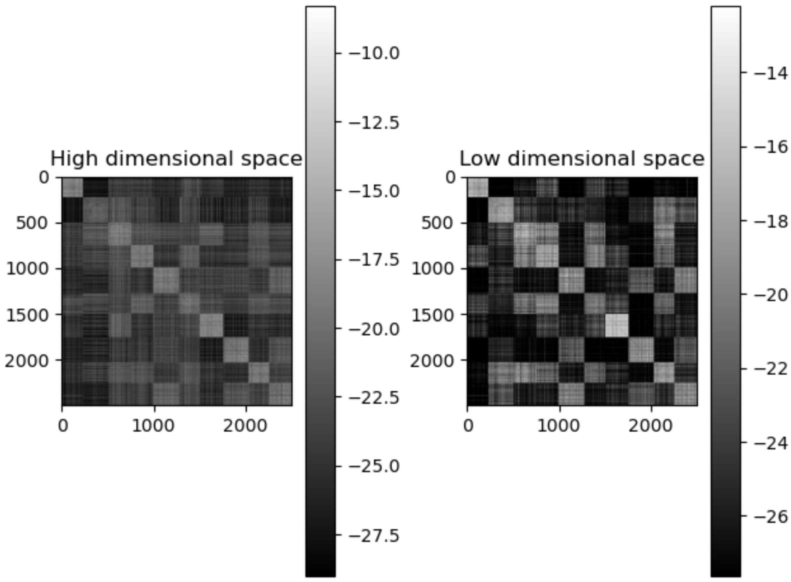
	Eigenfaces	Reconstructed faces	Error Rate
Simple			 Error count: 1 Error rate: 0.0333333333333333
Linear kernel			 Error count: 23 Error rate: 0.7666666666666667
RBF kernel			 Error count: 15 Error rate: 0.5

t-SNE

1. Embedding

Perplexity	t-SNE	s-SNE
20		
30		
50		

2. Pairwise Similarities

t-SNE	
s-SNE	

Observations and discussion

1. Simple LDA was a little better than simple PCA and kernel PCA. However, kernel LDA had the worst performance, hence the reconstructed face was more vague.
2. From graphs of tSNE and sSNE above, sSNE suffer the crowd problem, it is hard to distinguish different class without color. However, tSNE is easy, because it use the student t-distribution which is known as a longer tail than Gaussian distribution, it makes the point not need to be too far to get the low probability.
3. Perplexity can be seen as the number of influentail neighbors. The bigger perplexity, the more neighbors. Beside, the small perplexity means affected by few

neighbors, it may divide a class into many groups. And the Big perplexity have the clear global structure, but it may fuzzy the boundary between classes.