

ML HW7 Report

309552007 袁鈺勳

A. Code

a. Kernel Eigenfaces

1. PCA

In PCA, it first finds projection matrix W and find 25 first eigenvectors with largest eigenvalues. Then, it uses these eigenvectors to display eigenfaces and reconstruct 10 randomly chosen faces. Last, the eigenvectors are used to classify test images.

```
if not mode:
    # Simple PCA
    info_log('== Simple PCA ==')
    matrix = simple_pca(num_of_training, training_images)
else:
    # Kernel PCA
    info_log(f'== {"RBF" if kernel_type else "Linear"} kernel PCA ==')
    matrix = kernel_pca(training_images, kernel_type, gamma)

# Find 25 first largest eigenvectors
target_eigenvectors = find_target_eigenvectors(matrix)

# Transform eigenvectors into eigenfaces
info_log('== Transform eigenvectors into eigenfaces ==')
transform_eigenvectors_into_faces(target_eigenvectors, 0)

# Randomly reconstruct 10 eigenfaces
info_log('== Reconstruct 10 faces ==')
construct_faces(num_of_training, training_images, target_eigenvectors)

# Classify
info_log('== Classify ==')
classify(num_of_training, len(testing_images), training_images, training_labels, testing_images, testing_labels,
        target_eigenvectors, k_neighbors)
```

Function “simple_pca” calculates the covariance of training images.

```
# Compute covariance
training_images_transposed = training_images.T
mean = np.mean(training_images_transposed, axis=1)
mean = np.tile(mean.T, (num_of_images, 1)).T
difference = training_images_transposed - mean
covariance = difference.dot(difference.T) / num_of_images

return covariance
```

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

Function “kernel_pca” calculates the gram matrix “kernel” and uses it to calculate the matrix K according to the function in the above

image.

```
# Compute kernel
if not kernel_type:
    # Linear
    kernel = training_images.T.dot(training_images)
else:
    # RBF
    kernel = np.exp(-gamma * cdist(training_images.T, training_images.T, 'sqeuclidean'))

# 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
```

2. LDA

In LDA, it first finds projection matrix W and find 25 first eigenvectors with largest eigenvalues. Then, it uses these eigenvectors to display fisherfaces and reconstruct 10 randomly chosen faces. Last, the eigenvectors are used to classify test images.

```
if not mode:
    # Simple LDA
    info_log('== Simple LDA ==')
    matrix = simple_lda(num_of_each_class, training_images, training_labels)
else:
    # Kernel LDA
    info_log(f'== {"RBF" if kernel_type else "Linear"} kernel LDA ==')
    matrix = kernel_lda(num_of_each_class, training_images, training_labels, kernel_type, gamma)

# Find 25 first largest eigenvectors
target_eigenvectors = find_target_eigenvectors(matrix)

# Transform eigenvectors into fisherfaces
info_log('== Transform eigenvectors into fisherfaces ==')
transform_eigenvectors_into_faces(target_eigenvectors, 1)

# Randomly reconstruct 10 eigenfaces
info_log('== Reconstruct 10 faces ==')
construct_faces(num_of_training, training_images, target_eigenvectors)

# Classify
info_log('== Classify ==')
classify(num_of_training, len(testing_images), training_images, training_labels, testing_images, testing_labels,
        target_eigenvectors, k_neighbors)
```

within-class scatter: $S_W = \sum_{j=1}^k S_j$, where $S_j = \sum_{i \in \mathcal{C}_j} (x_i - \mathbf{m}_j)(x_i - \mathbf{m}_j)^\top$
and $\mathbf{m}_j = \frac{1}{n_j} \sum_{i \in \mathcal{C}_j} x_i$

between-class scatter:

$$S_B = \sum_{j=1}^k S_{B_j} = \sum_{j=1}^k n_j (\mathbf{m}_j - \mathbf{m})(\mathbf{m}_j - \mathbf{m})^\top$$
$$\text{where } \mathbf{m} = \frac{1}{n} \sum x$$

Function “simple_lda” finds the between-class scatter and within-class scatter according to the functions in the above two images.

$$S_W^{-1} S_B \text{ as } W$$

Then, the function uses between-class scatter and within-class scatter to obtain the projection matrix W according to the definition in the above image.

```
# Get overall mean
info_log('=== Calculate overall mean ===')
overall_mean = np.mean(training_images, axis=0)

# Get mean of each class
info_log('=== Calculate mean of each class ===')
num_of_classes = len(num_of_each_class)
class_mean = np.zeros((num_of_classes, 29 * 24))
for label in range(num_of_classes):
    class_mean[label, :] = np.mean(training_images[training_labels == label + 1], axis=0)

# Get between-class scatter
info_log('=== Calculate 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
info_log('=== Calculate within-class scatter ===')
scatter_w = np.zeros((29 * 24, 29 * 24), dtype=float)
for idx, mean in enumerate(class_mean):
    difference = training_images[training_labels == idx + 1] - mean
    scatter_w += difference.T.dot(difference)

# Get Sw^(-1) * Sb
info_log('=== Calculate inv(within-class) * between-class ===')
matrix = np.linalg.pinv(scatter_w).dot(scatter_b)

return matrix
```

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

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

Function “kernel_lda” calculates the matrix N and M according to the functions in the above four images.

$$\mathbf{N}^{-1} \mathbf{M}$$

Then, the function calculates the projection matrix W according to the definition in the above image.

```

# Compute kernel
num_of_classes = len(num_of_each_class)
num_of_images = len(training_images)
if not kernel_type:
    # Linear
    kernel_of_each_class = np.zeros((num_of_classes, 29 * 24, 29 * 24))
    for idx in range(num_of_classes):
        images = training_images[training_labels == idx + 1]
        kernel_of_each_class[idx] = images.T.dot(images)
    kernel_of_all = training_images.T.dot(training_images)
else:
    # RBF
    kernel_of_each_class = np.zeros((num_of_classes, 29 * 24, 29 * 24))
    for idx in range(num_of_classes):
        images = training_images[training_labels == idx + 1]
        kernel_of_each_class[idx] = np.exp(-gamma * cdist(images.T, images.T, 'sqeuclidean'))
    kernel_of_all = np.exp(-gamma * cdist(training_images.T, training_images.T, 'sqeuclidean'))

# Compute N
info_log('=== Calculate 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)

# Compute M
info_log('=== Calculate matrix M ===')
matrix_m_i = np.zeros((num_of_classes, 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) / num_of_images
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
info_log('=== Calculate inv(N) * M ===')
matrix = np.linalg.pinv(matrix_n).dot(matrix_m)

return matrix

```

3. Eigenvectors

Function “find_target_eigenvectors” finds 25 first eigenvectors with largest eigenvalues. It only returns the real part of these eigenvectors.

```

# Compute eigenvalues and eigenvectors
info_log('=== Calculate eigenvalues and eigenvectors ===')
eigenvalues, eigenvectors = np.linalg.eig(matrix)

# Get 25 first largest eigenvectors
info_log('=== Get 25 first largest eigenvectors ===')
target_idx = np.argsort(eigenvalues)[::-1][:25]
target_eigenvectors = eigenvectors[:, target_idx].real

return target_eigenvectors

```

4. From Eigenvectors to Faces

Function “transform_eigenvectors_into_faces” reshapes the eigenvectors so as to display them as eigenfaces or fisherfaces.

```

faces = target_eigenvectors.T.reshape((25, 29, 24))
fig = plt.figure(1)
fig.canvas.set_window_title(f'{"Eigenfaces" if not pca_or_lda else "Fisherfaces"}')
for idx in range(25):
    plt.subplot(5, 5, idx + 1)
    plt.axis('off')
    plt.imshow(faces[idx, :, :], cmap='gray')

```

5. Face Reconstruction

$$x = W W^T x$$


Function “construct_faces” randomly chooses 10 images and reconstructs these faces according to the function in the above image.

```

reconstructed_images = np.zeros((10, 29 * 24))
choice = np.random.choice(num_of_images, 10)
for idx in range(10):
    reconstructed_images[idx, :] = training_images[choice[idx], :].dot(target_eigenvectors).dot(
        target_eigenvectors.T)
fig = plt.figure(2)
fig.canvas.set_window_title('Reconstructed faces')
for idx in range(10):
    # Original image
    plt.subplot(10, 2, idx * 2 + 1)
    plt.axis('off')
    plt.imshow(training_images[choice[idx], :].reshape((29, 24)), cmap='gray')

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

```

6. Classification

The training images and testing images are first decorrelated by eigenvectors in function “classify”. Then, the function uses k nearest neighbors to decide which the testing image should belong to.

```
decorrelated_training = decorrelate(num_of_training, training_images, target_eigenvectors)
decorrelated_testing = decorrelate(num_of_testing, testing_images, target_eigenvectors)
error = 0
distance = np.zeros(num_of_training)
for test_idx, test in enumerate(decorrelated_testing):
    for train_idx, train in enumerate(decorrelated_training):
        distance[train_idx] = np.linalg.norm(test - train)
    min_distances = np.argsort(distance)[:k_neighbors]
    predict = np.argmax(np.bincount(training_labels[min_distances]))
    if predict != testing_labels[test_idx]:
        error += 1
print(f'Error count: {error}\nError rate: {float(error) / num_of_testing}')
```

$$z = Wx$$

Function “decorrelate” projects the images on another space according to the function in the above image.

```
decorrelated_images = np.zeros((num_of_images, 25))
for idx, image in enumerate(images):
    decorrelated_images[idx, :] = image.dot(eigenvectors)

return decorrelated_images
```

b. t-SNE

1. Modification

The two image below are the functions of t-SNE which is the original version of the code.

$$q_{ij} = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_{k \neq l} (1 + \|y_i - y_k\|^2)^{-1}} \frac{\partial C}{\partial y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)(1 + \|y_i - y_j\|^2)^{-1}$$

The two images below are the functions of symmetric SNE.

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

Based on the functions above, we only need to modify q and gradient. The implementations are in the images below. Mode is controlled by “-m” argument from command line will decide which type of SNE to be used.

```

if not mode:
    # t-SNE
    num = 1. / (1. + np.add(np.add(num, sum_y).T, sum_y))
else:
    # symmetric SNE
    num = np.exp(-1. * np.add(np.add(num, sum_y).T, sum_y))

if not mode:
    # t-SNE
    for i in range(n):
        d_y[i, :] = np.sum(np.tile(p_q[:, i] * num[:, i], (no_dims, 1)).T * (solution_y[i, :] - solution_y), 0)
else:
    # symmetric SNE
    for i in range(n):
        d_y[i, :] = np.sum(np.tile(p_q[:, i], (no_dims, 1)).T * (solution_y[i, :] - solution_y), 0)

```

2. GIF

The code below will capture current scatter graph once every 10 iterations.

```
img.append(capture_current_state(solution_y, labels, mode, perplexity))
```

In function “capture_current_state”, a scatter graph will be generated and stored as an image. These images will be used to generate a gif file.

```

plt.clf()
plt.scatter(solution_y[:, 0], solution_y[:, 1], 20, labels)
plt.title(f'{"t-SNE" if not mode 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())

```

3. Pairwise Similarities

After SNE is finished, high-dimensional similarities and low-dimensional similarities will be displayed as images.

```
# Get sorted index
index = np.argsort(labels)
plt.clf()
plt.figure(1)

# Plot p
log_p = np.log(p)
sorted_p = log_p[index][:, index]
plt.subplot(121)
img = plt.imshow(sorted_p, cmap='gray', vmin=np.min(log_p), vmax=np.max(log_p))
plt.colorbar(img)
plt.title('High-dimensional space')





# Plot q
log_q = np.log(q)
sorted_q = log_q[index][:, index]
plt.subplot(122)
img = plt.imshow(sorted_q, cmap='gray', vmin=np.min(log_q), vmax=np.max(log_q))
plt.colorbar(img)
plt.title('Low-dimensional space')

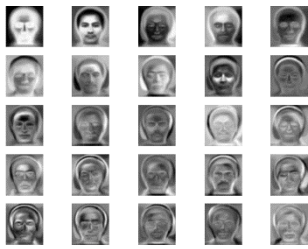

plt.tight_layout()
```

B. Results

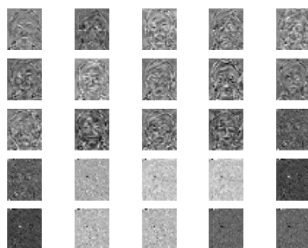

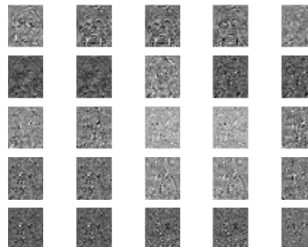

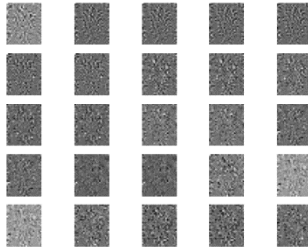

a. Kernel Eigenfaces

1. PCA

	Eigenfaces	Reconstructed faces	Error rate
Simple			Error count: 3 Error rate: 0.1
Linear kernel			Error count: 4 Error rate: 0.13333333333333333

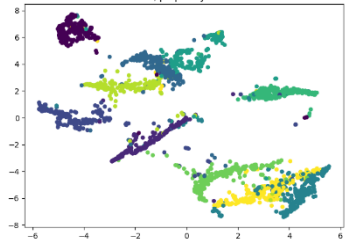
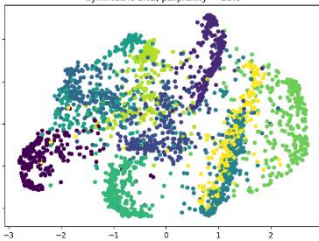
RBF kernel			Error count: 4 Error rate: 0.1333333333333333
---------------	---	--	--

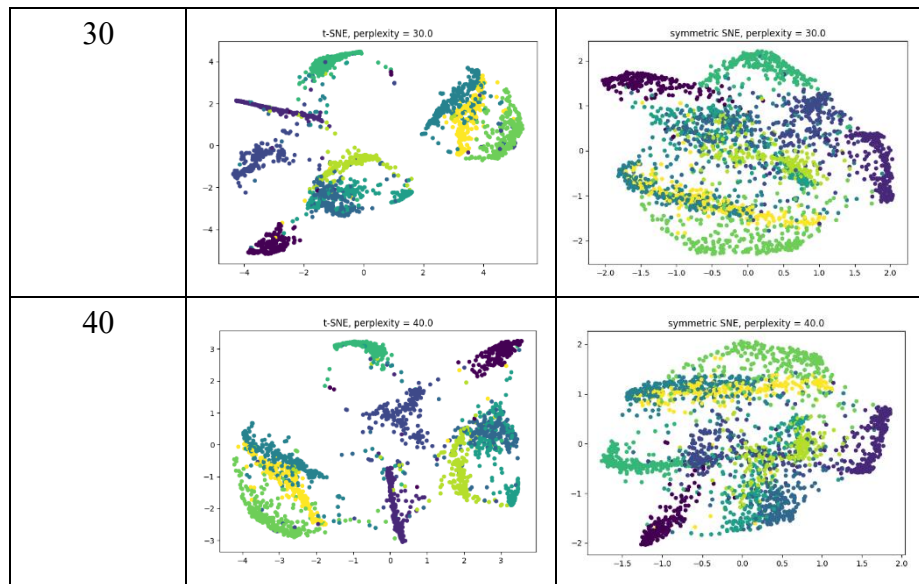
2. LDA

	Fisherfaces	Reconstructed faces	Error rate
Simple			Error count: 1 Error rate: 0.0333333333333333
Linear kernel			Error count: 21 Error rate: 0.7
RBF kernel			Error count: 13 Error rate: 0.4333333333333333

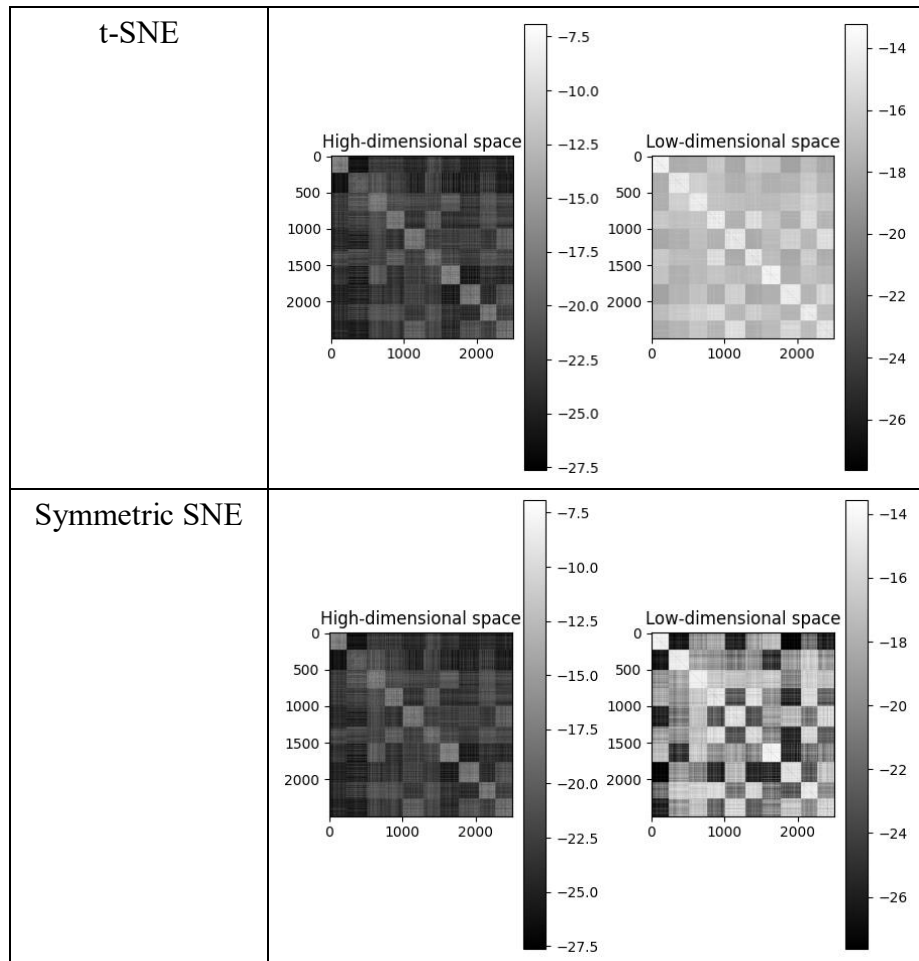
b. t-SNE

1. Embedding

Perplexity	t-SNE	Symmetric SNE
20		



2. Pairwise Similarities



C. Observation

- Reconstructed faces from LDA are not as clear as those from PCA.
- Performance of LDA is better than that of PCA based on only simple

version. The reason that the performance of kernel LDA is bad might be that my implementation of it is wrong.

- c. Eigenfaces depict the contour of the faces so as to extract the features of the images. Based on these features, we can classify the testing images.
- d. Reconstructed faces of PCA look like original faces.
- e. There is no huge difference of the performance between simple version and kernel version. Also, the performances of linear kernel and RBF kernel are similar.
- f. From the scatter graphs, symmetric SNE suffers from the crowded problem. It is hard to distinguish different class without color. Nevertheless, t-SNE uses t-distribution to alleviate crowded problem. Points far away from each other in high-dimensional space still have far distance from each other in low-dimensional space.
- g. The speed of convergence of symmetric SNE is faster than that of t-SNE via the results of gif files.
- h. Perplexity will affect the number of neighbors to be considered. The larger the perplexity is, the less sensitive the points is to smaller group. Because symmetric SNE suffers from crowded problem, the effect of perplexity is not clear. However, the effect of perplexity is clear in t-SNE as it gets larger.