1.

eigen-decomposition $\Rightarrow U \Lambda U^T$ (Λ: eigenvalues, U: eigenvectors)

In SVD, $A = U \Sigma V^T$. The values of eigenvalue (or singular value) in Σ should be non-negative. ①

and diagonal element (eigenvalues) should satisfy "decreasing order". ②

ex) $\sigma_1 \geq \sigma_2 \geq \sigma_3 \geq \cdots \geq 0$

We should change Λ (from) to Σ and satisfy the above condition: ①, ②

We also need to modify U and V^T slightly. Then we get.

$$A = \begin{bmatrix} u_{11} & -u_{13} & u_{12} \\ u_{21} & -u_{23} & u_{22} \\ u_{31} & -u_{33} & u_{32} \end{bmatrix} \begin{bmatrix} 3 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{21} & u_{31} \\ u_{13} & u_{23} & u_{33} \\ u_{12} & u_{22} & u_{32} \end{bmatrix}$$

2.

H is a symmetric matrix then

$\text{svd}(H) = \underset{n \times n}{U} \Sigma V^T = U \Sigma U^T$ (because, it is symmetric)

$\underset{n \times n}{U}$

$$\max_{A \in R^{n \times k} : A^T A = I_k} \text{trace}(A^T H A)$$

$= \max \text{trace}(A^T U \Sigma U^T A)$

$= \max \text{trace}((U^T A)^T \Sigma U^T A)$

$= \max \text{trace}(Q^T \Sigma Q)$ where $Q = U^T A$ and $Q \in R^{n \times k} : Q^T Q = I_k$

This simplifies into

$= \max_{Q \in R^{n \times k} : Q^T Q = I} \text{trace}(Q^T \Sigma Q)$

Here we use the property of 'trace'

for example, trace(ABC) = trace(BCA)

We can rewrite the previous equation by the following

$= \max \text{trace}(\Sigma Q Q^T)$

trace(A) is just the summation of diagonal components.

$= \max\left(\sum_{i=1}^{n} \lambda_i b_i^T b_i\right)$ where "b" is the diagonal element of Q

Let $Q^T = A^T U$. Remember that $Q = U^T A$ previously. $A = Q \in R^{n \times k}$

$= \max\left(\sum_{i=1}^{n} \lambda_i a_i^T u_i u_i^T a_i\right)$

$= \max\left(\sum_{i=1}^{n} \lambda_i a_i^T a_i\right)$ because $u_i^T u_i = 1$ ($\because U$ is orthogonal and H is symmetric)

$$a_i^T a_i = \begin{cases} 0 & i > K \\ 1 & i \le K \end{cases}$$ and this is why $A^T A = I_k$!

we get rid of max here

$= \sum_{i=1}^{k} \lambda_i$ $\left(\because \max\left(\sum_{i=1}^{k} \lambda_i b_i^T b_i + \sum_{i=k+1}^{n} \lambda_i b_i^T b_i\right)\right)$

$\sum_{i=1}^{k} \lambda_i$ $= 0$

(3) when $1 \le i \le K$, $u_i = A q_i$, $q_i = q_1 \cdots q_k$ (q is a orthogonal vector)

(1) $\| A^T u_i \|_2 = 1$

(2) when $K < i \le n$

$\| A^T u_i \|_2 = 0$ ★ this is important

$\therefore A^* = [u_1, u_2, \cdots u_k] Q$ (and $Q = [q_1, q_2 \cdots \quad]^T \in R^{K \times K}$ Q is an arbitrary orthogonal matrix.)

(1), (2), (3) lead to the fact that

$\lambda_1 + \cdots + \lambda_k = \max_{A \in R^{n \times k},\, A^T A = I_k} \text{trace}(A^T H A)$

Result and Analysis

Jin Hyun Park, UIN: 633001823

3-a

- In `class Kernel_LR`, bias of `nn.Linear` is set to `False`.
- After hyperparameter tuning with different learning rate, epoch, batch size, and sigma, the best hyperparameters that I have found is the following
    - Learning rate = 0.01, epoch = 100, batch size = 32, and sigma = 1
- Test accuracy = 0.987 (98.7%)

3-b

- In `class RBF`, bias of `nn.Linear` is set to `False`
- After hyperparameter tuning with different hidden dimension size, learning rate, epoch, batch size, and sigma, the best hyperparameters that I have found is the following
    - Hidden dim size = 32, learning rate = 0.01, epoch = 100, batch size = 128, and sigma = 1
- Test accuracy = 0.985 (98.5%)

3-c

- In `class FFN`, bias of `nn.Linear` is set to `True`
- After hyperparameter tuning with different hidden dimension size, learning rate, epoch, and batch size, the best hyperparameters that I have is the following:
    - Hidden dim size = 32, learning rate = 0.01, epoch = 100, batch size = 128
- Test accuracy = 0.989 (98.9%)

Summary of Question 3

	Hidden dimension size	Test accuracy
Kernel LR	# of training samples	98.7%
RBF	32	98.5%
FFN	32	98.9%

- For comparison, the hidden layer dimension size is set to 32 for both RBF and FFN.
- RBF achieved similar test accuracy to Kernel LR (only 0.2% difference). This implies that RBF can perform similarly to Kernel LR with fewer parameters.
- FFN performs slightly better than RBF (only 0.4% difference). This implies that RBF can perform similarly to FFN with fewer parameters. This is possible because we used the K-means algorithm to find centroids.

4-b

# of principal component (p)	Reconstruction Error (RE)
32	129.37
64	85.82
128	45.63

4-d

# of hidden representation (d)	Reconstruction Error (RE)
32	132.70
64	86.63
128	46.60

4-e

p=d	RE of PCA	RE of AE
32	129.37	132.70
64	85.82	86.63
128	45.63	46.60

- The reconstruction error of PCA is slightly better than AE (Auto Encoder) but the differences are very small. This is because we did not use a non-linear activation function. With a linear activation function, PCA and AE basically do the same thing.

4-f

p=d	frobeniu_norm_error(G, W)	frobeniu_norm_error(G^T.G, W^T.W)
32	8.11	0.17
64	11.44	0.03
128	15.92	0.02

Note 1: It is represented as G.G^T and W.W^T instead of G^T.G and W^T.W in the following URL: [1] http://people.tamu.edu/~sji/classes/PCA.pdf.
Note 2: G^T.G and W^T.W are the same as G.G^T and W.W^T in my code implementation.

According to [1], a comparison of frobeniu_norm_error between G^T.G and W^T.W is a better approach than a comparison of frobeniu_norm_error between G and W. Unsurprisingly, it turned out that the errors were significantly reduced.

4-g

p=d	Shared weights	Non-shared weights
32	132.70	130.28
64	86.63	86.43
128	46.60	46.76

The result of shared weights and non-shared weights is very similar. The decoding part of the network essentially learns to mirror the encoder part for the non-shared weights model.

4-h

- After hyperparameter tuning with different epochs and batch sizes, the best hyperparameters that I have found is the following (d is set to 64).
    - Batch size = 64 and epoch = 3000
- I used the following network structure:
    - (Encoder) layer 1 → ReLU → (Encoder) layer 2 → [Hidden Representation] → (Decoder) layer 3 → ReLU → (Decoder) layer 4
    - For more details, please see the following two figures

```
W_encoder1 = torch.empty(self.n_features, 2 * self.d_hidden_rep)
self.w_encoder1 = nn.Parameter(nn.init.kaiming_normal_(W_encoder1))

W_encoder2 = torch.empty(2 * self.d_hidden_rep, self.d_hidden_rep)
self.w = nn.Parameter(nn.init.kaiming_normal_(W_encoder2))    # w == w_encoder2

W_decoder1 = torch.empty(self.d_hidden_rep, 2 * self.d_hidden_rep)
self.w_decoder1 = nn.Parameter(nn.init.kaiming_normal_(W_decoder1))

W_decoder2 = torch.empty(2 * self.d_hidden_rep, self.n_features)
self.w_decoder2 = nn.Parameter(nn.init.kaiming_normal_(W_decoder2))
```

```
encoder1 = torch.mm(self.w_encoder1.T, X)
encoder1 = torch.relu(encoder1)
encoder2 = torch.mm(self.w.T, encoder1)

decoder1 = torch.mm(self.w_decoder1.T, encoder2)
decoder1 = torch.relu(decoder1)
decoder2 = torch.mm(self.w_decoder2.T, decoder1)

return decoder2
```

	Reconstruction error (d=64)
PCA	85.82
Auto Encoder with shared weights	86.63
Auto Encoder with non-shared weights	86.43
Auto encoder with non-linearity and more layers	70.67

The Auto Encoder with non-linearity (= non-linear activation function) and more layers outperformed the others with respect to reconstruction error. This is because it (= the last method, 4-h) becomes **manifold learning** which is an approach to non-linear dimensionality reduction. I strongly believe that with more layers and trying out different combinations of hyperparameters (such as activation functions and batch size), it will be possible to get a lower reconstruction error.

5. Bonus.

K_{train} : training kernel

K_{test} : testing kernel

Let $A \in \mathbb{R}^{n*K}$. A = training data , K = the number of features

" $B \in \mathbb{R}^{m*K}$. B = testing data, K = "

$$K_{train} = A \cdot A^T \in \mathbb{R}^{n \times n}$$

$$svd(K_{train}) = U \Sigma V^T$$

↑ it is symmetric

$$= U \Sigma U^T$$

we can divide Σ because Σ is symmetric and p.s.d

$$= U\Sigma^{1/2} (U\Sigma^{1/2})^T$$

set $\Rightarrow A = U\Sigma^{1/2}$ --- ① all of diagonal element is > 0 because it is compact SVD

<NOTE>

" ① and ② implies that the training and the testing data can be recovered or obtained by the kernel."

In one line, $K_{train} = A \cdot A^T = U\Sigma^{1/2}(U\Sigma^{1/2})^T$

This implies that we can use U and Σ for training data. And, U and Σ can be obtained by svd of AA^T

For testing,

$$K_{test} = A \cdot B^T$$

$$= U\Sigma^{1/2} B^T$$

$$U^T K_{test} = \underbrace{U^T U}_{I} \Sigma^{1/2} B^T$$

$$= \Sigma^{1/2} \cdot B^T$$

$$B^T = (\Sigma^{1/2})^{-1} U^T K_{test}$$

$$B = K_{test}^T \cdot U \cdot (\Sigma^{-1/2})^T \quad \text{--- ②}$$

This implies that we can use U and Σ for testing data. And, U and Σ can be obtained by svd of AA^T

→ Therefore, we can use only the solver and data given above to train a logistic classifier and perform testing.