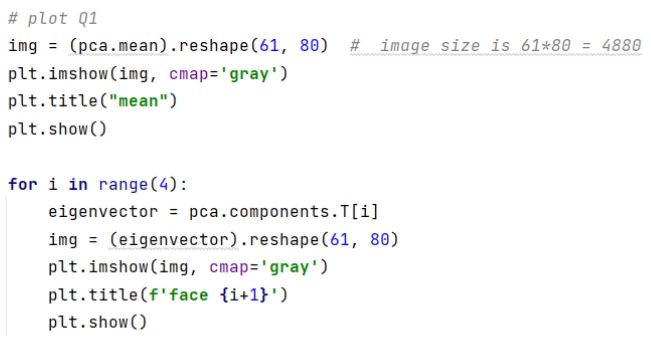
Code Part Problem Report

學號：t11902210 姓名：張一凡

（a）Plot the mean vector as an image as well as the top 4 eigenvectors, each as an image, by calling the given plot component routine. Each of those top eigenvectors is usually called an eigenface that physically means an informative “face ingredient.”

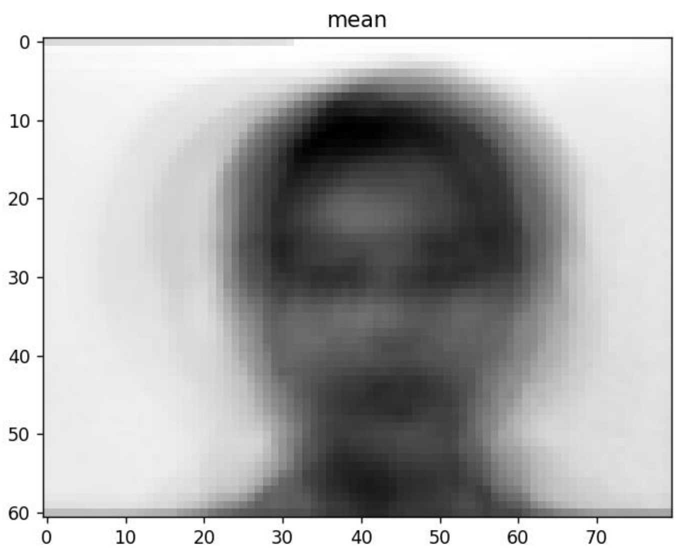
Answer：

First, what I am showing is the code that generates the average vector and the first four feature vectors. I use the imshow function to display the output, and because it is a grayscale image, I have added the parameter cmap='gray '. However, since the image set is trained by PCA to form a 4880-dimensional vector, there are certain requirements for the image's reshaping. Later, I used the prompts and load in the question_ The data function attempted to output the image specification and found that the image is 61 * 80 in size, which was successfully output. In addition, when outputting feature vectors, due to the convenience of performing transformations in the PCA definition, the specification is 4880 * 40. Therefore, I need to perform some transposing before outputting in order. Here is my code:



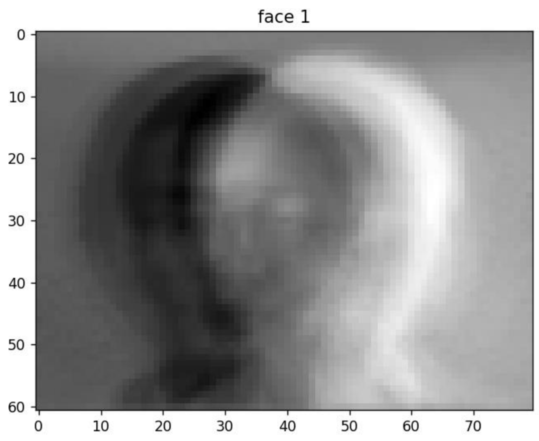
```python
# plot Q1
img = (pca.mean).reshape(61, 80)  #  image size is 61*80 = 4880
plt.imshow(img, cmap='gray')
plt.title("mean")
plt.show()

for i in range(4):
    eigenvector = pca.components.T[i]
    img = (eigenvector).reshape(61, 80)
    plt.imshow(img, cmap='gray')
    plt.title(f'face {i+1}')
    plt.show()
```

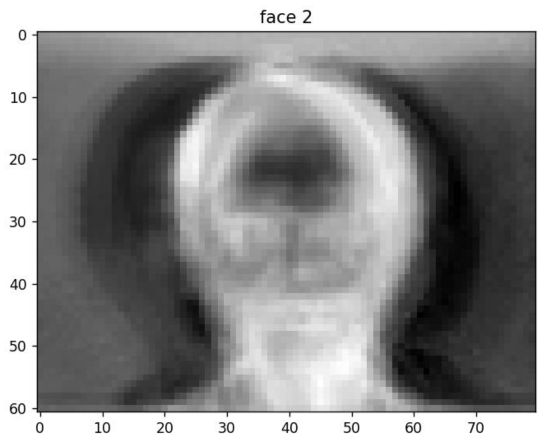
The first thing to display is the image drawn through the average vector, which displays the basic elements of the portrait, while also fusing the common features of multiple facial images as the average face.

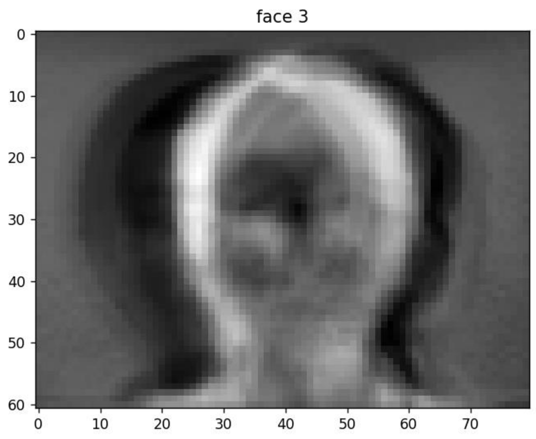


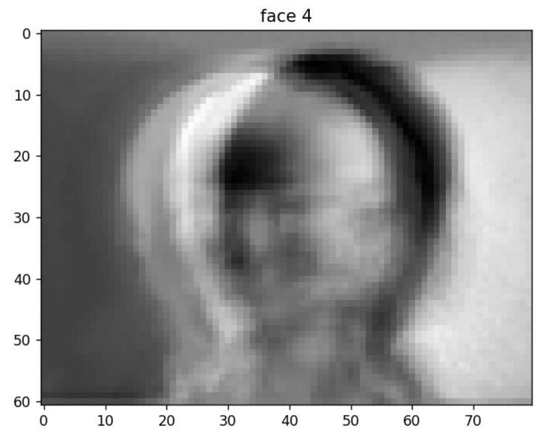
The following are the images drawn by the first four feature vectors that I have selected.

I have marked the result of which vector is displayed on each image. From the image display, it can be seen that different feature vectors correspond to different faces, which are somewhat different from the average face in the whiter part in the middle. This may be due to the difference between each vector and the principal component, resulting in the display of different features, such as lighting and facial expressions, Gender characteristics, etc. (personal analysis)





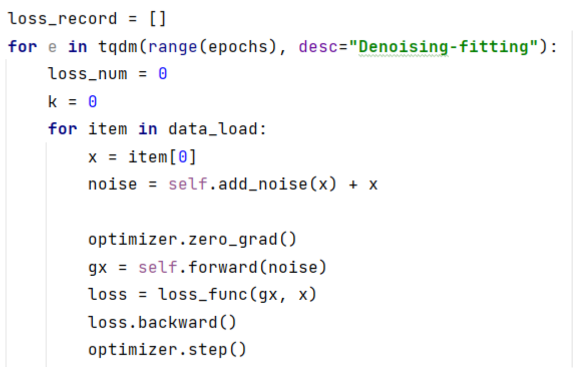




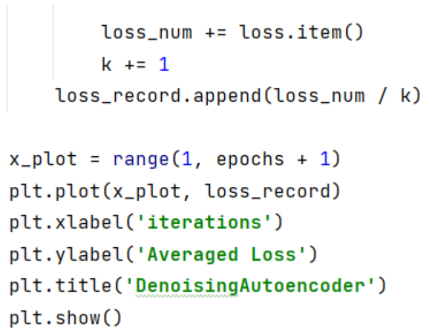
（b）Plot the training curve of Autoencoder and DenoisingAutoencoder

Answer：

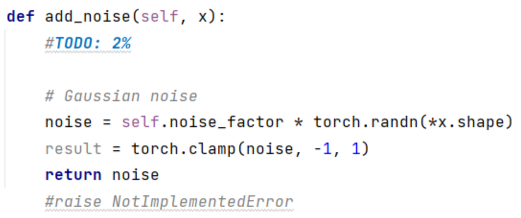
Firstly, I also demonstrate the process of calculating the loss rate according to the requirements of the question in Autoencoder and Denoising Autoencoder, and outputting the curve based on the number of iterations or epochs. I have set a variable loss in each epoch_ Num and divide it by the number to calculate the average loss for each iteration process and place it in a list loss_ In the record, finally output the curve through plot. In the DenoisingAutoencoder, I decided to use Gaussian noise after searching for data. Since the process of recording losses and generating curves is basically the same for both the Autoencoder and DenoisingAutoencoder, I chose to display the code of the DenoisingAutoencoder, as well as the code that generates noise.



```
loss_record = []
for e in tqdm(range(epochs), desc="Denoising-fitting"):
    loss_num = 0
    k = 0
    for item in data_load:
        x = item[0]
        noise = self.add_noise(x) + x

        optimizer.zero_grad()
        gx = self.forward(noise)
        loss = loss_func(gx, x)
        loss.backward()
        optimizer.step()
```



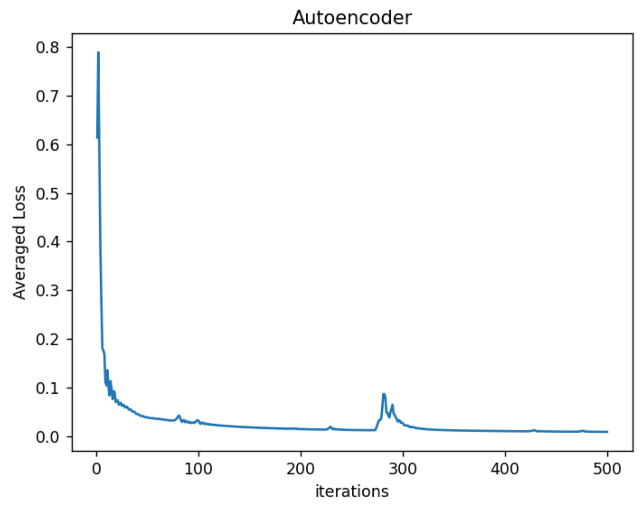
```
        loss_num += loss.item()
        k += 1
    loss_record.append(loss_num / k)

x_plot = range(1, epochs + 1)
plt.plot(x_plot, loss_record)
plt.xlabel('iterations')
plt.ylabel('Averaged Loss')
plt.title('DenoisingAutoencoder')
plt.show()
```



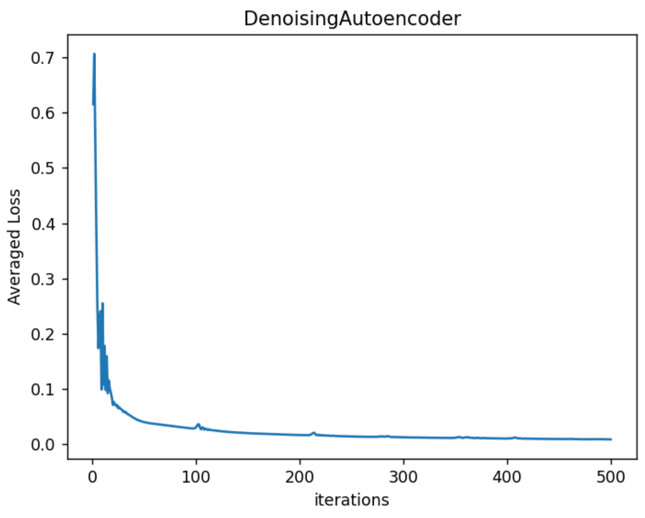
```
def add_noise(self, x):
    #TODO: 2%

    # Gaussian noise
    noise = self.noise_factor * torch.randn(*x.shape)
    result = torch.clamp(noise, -1, 1)
    return noise
    #raise NotImplementedError
```

Here is the training curve of Autoencoder and Denoising Autoencoder.

Autoencoder average loss curve



DenoisingAutoencoder average loss curve



From the results presented by the two curves, it can be seen that both Autoencoder and DenoisingAutoencoder can handle this problem well, with fast convergence speed. Therefore, they will gradually learn the feature representation of the input data, thereby gradually reducing the error. However, due to the addition of noise, DenoisingAutoencoder has greater advantages over Autoencoder. In the observation of this problem, it is mainly reflected in two aspects: 1. Although not obvious, from the subtle differences in the curve, it can be seen that the curve DenoisingAutoencoder exhibits faster convergence speed and ultimately lower error, indicating that adding noise will make the final loss rate smaller; 2. Although the training curve of DenoisingAutoencoder exhibits more fluctuations and fluctuations in the initial stage, it is extremely stable after convergence. On the contrary, Autoencoder exhibits significant fluctuations and instability in the later stage, indicating that DenoisingAutoencoder has stronger robustness and better performance.

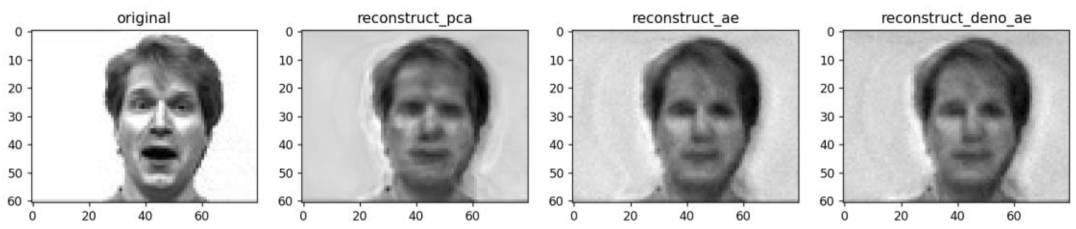
(c) Plot the original image and the images reconstructed with PCA, Autoencoder, and DenoisingAutoencoder side by side, ideally as large as possible. Then, list the mean squared error between the original image and each reconstructed image

Answer:

Firstly, I also presented the code to solve this problem. As the requirement in the question is to draw four images side by side, I called the subplots function to generate sub images and reshape each of the four images and display them on one image. The following is my implementation code.

```
# plot Q3
fig, axs = plt.subplots(1, 4, figsize=(20, 5))
img_original = (img_vec).reshape(61, 80)  #  image size is 61*80 = 4880
axs[0].imshow(img_original, cmap='gray')
axs[0].set_title("original")
img_pca = (img_reconstruct_pca).reshape(61, 80)  #  image size is 61*80 = 4880
axs[1].imshow(img_pca, cmap='gray')
axs[1].set_title("reconstruct_pca")
img_ae = (img_reconstruct_ae).reshape(61, 80)  #  image size is 61*80 = 4880
axs[2].imshow(img_ae, cmap='gray')
axs[2].set_title("reconstruct_ae")
img_deno_ae = (img_reconstruct_deno_ae).reshape(61, 80)  #  image size is 61*80 = 4880
axs[3].imshow(img_deno_ae, cmap='gray')
axs[3].set_title("reconstruct_deno_ae")
plt.show()
```

The following are the image results I generated, from left to right are the original image, reconstructed with PCA image, reconstructed with Autoencoder image, and reconstructed with DenoisingAutoencoder image.



Next, I will list the mean square error between the original image and each reconstructed image, which is the output result of the given code.

```
Reconstruction Loss with PCA: 0.010710469688056319
Reconstruction Loss with Autoencoder: 0.012720367232229133
Reconstruction Loss with DenoisingAutoencoder: 0.01331009228793101
```

Firstly, among the square error reconstruction errors, PAC has the lowest reconstruction error of 0.011, while AE and denoised AE have lower reconstruction errors of 0.0127 and 0.0133, respectively. This result may indicate that compared to PCA, the ability of automatic encoders and denoising automatic encoders to reconstruct the original image in this task is relatively weak. This may be because PCA preserves the information of the original data as much as possible during the reconstruction process, so it can perform well in terms of square error reconstruction error. At the same time, the automatic encoder and denoising automatic encoder are based on neural networks and can capture nonlinear structures in the data. Although they are not as good as PAC in this problem, they also perform well in terms of absolute results in the reconstructed image. Finally, when observing the results of image display, AE and denoising AE cannot visually generate images as close to the original image as possible compared to PCA, making the PCA method clearer. After searching for information, my conclusion is that they may be trying to learn more "smooth" or "average" data representations, which may cause the reconstructed image to appear a bit blurry, but this may indicate that AE and denoising AE are learning the internal structure of the data, rather than just remembering the input data. Overall, low reconstruction errors do not always mean better performance.

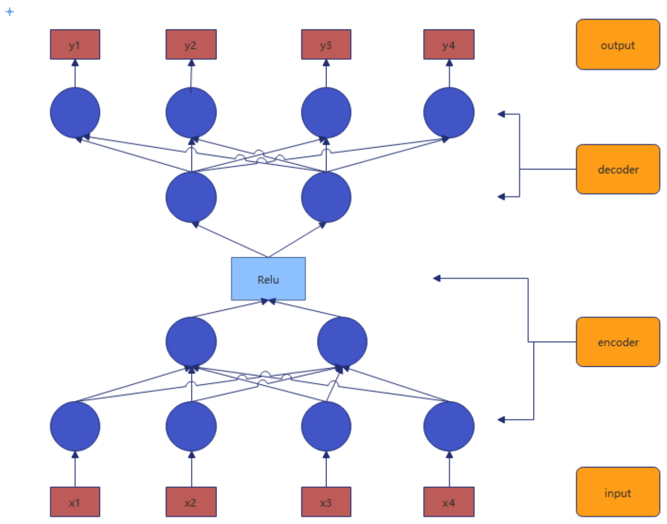
(d) Modify the architecture in Autoencoder in its constructor. Try at least two different network architectures for the denoising autoencoder. You can consider trying a deeper or shallower or fatter or thinner network. You can also consider adding convolutional layers and/or other activation functions. Draw the architecture that you have tried and discuss your findings, particularly in terms of the reconstruction error that the architecture can achieve after decent optimization.

Answer:

I have made three modifications to this issue and combined them with the original results to display and output. I will summarize my observations and analysis later on. At the same time, I also used concept maps as required to complete the structural diagrams of these self encoders. Due to space limitations, I will set the dimensions of all inputs and outputs to 4, which is convenient for comparison and observation. I will place concept maps of each variant and original structure at the end of each code for inspection and explanation.

Original design

```
self.encoder = nn.Sequential(
    nn.Linear(input_dim, encoding_dim),
    nn.Linear(encoding_dim, encoding_dim//2),
    nn.ReLU()
)
self.decoder = nn.Sequential(
    nn.Linear(encoding_dim//2, encoding_dim),
    nn.Linear(encoding_dim, input_dim),
)
```



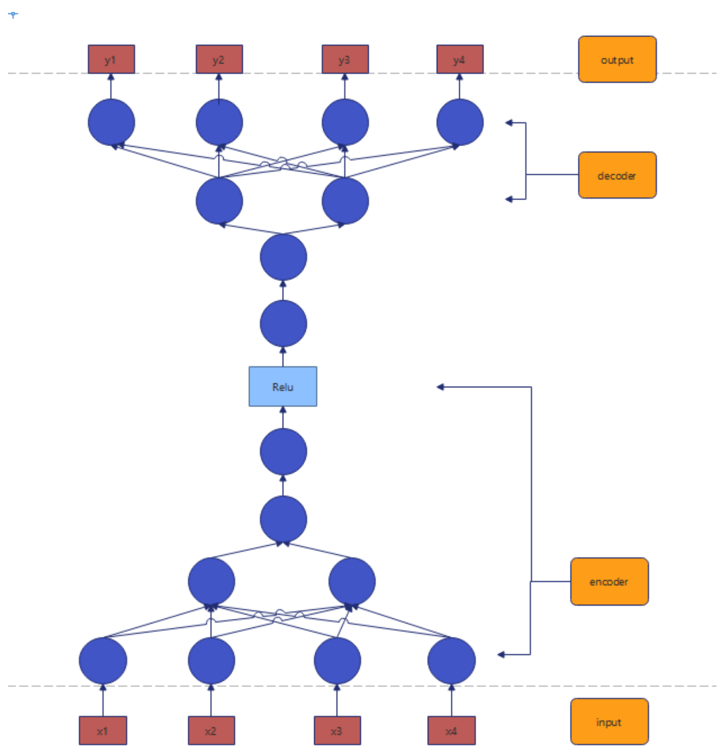
```
Acc from Autoencoder: 0.8666666666666667
Acc from DenoisingAutoencoder: 0.9333333333333333
```

```
Reconstruction Loss with Autoencoder: 0.015411890219807605
Reconstruction Loss with DenoisingAutoencoder: 0.012947307032105934
```

This is the accuracy and Reconstruction Loss of the original data, and each variant below will be compared with this and a conclusion will be drawn.

Deeper design

```
# deeper
self.encoder = nn.Sequential(
    nn.Linear(input_dim, encoding_dim),
    nn.Linear(encoding_dim, encoding_dim // 2),
    nn.Linear(encoding_dim // 2, encoding_dim // 3),
    nn.Linear(encoding_dim // 3, encoding_dim // 4),
    nn.ReLU()
)
self.decoder = nn.Sequential(
    nn.Linear(encoding_dim // 4, encoding_dim // 3),
    nn.Linear(encoding_dim // 3, encoding_dim // 2),
    nn.Linear(encoding_dim // 2, encoding_dim),
    nn.Linear(encoding_dim, input_dim),
)
```



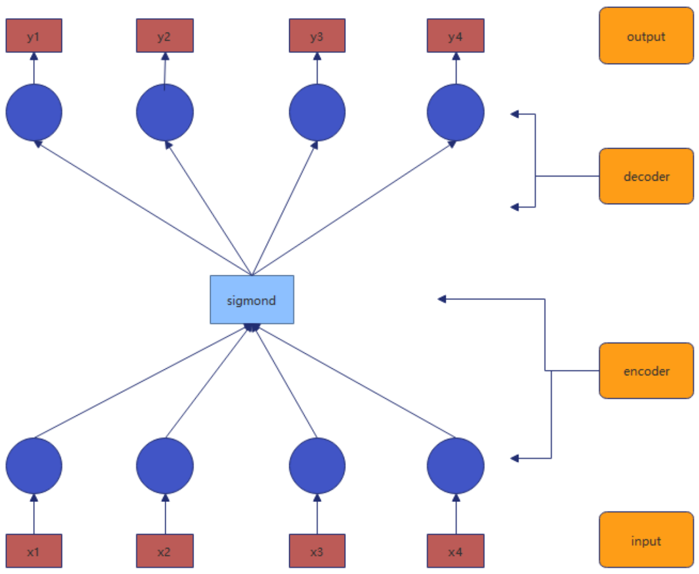
```
Acc from Autoencoder: 0.8666666666666667
Acc from DenoisingAutoencoder: 0.8666666666666667
```

```
Reconstruction Loss with Autoencoder: 0.014501936661465207
Reconstruction Loss with DenoisingAutoencoder: 0.014558046191633872
```

The first type is transformed into a deeper network, with an accuracy AE of 0.87 for both DenoiseAE and DenoiseAE, which is basically the same as the original accuracy AE, but DnoiseAE is slightly lower; In terms of Reconstruction Loss, both AE and DenoiseAE are approximately 0.0145. Compared with the original design, AE may be due to error issues, but the increase in DenoiseAE is more significant. I think the primary reason is the occurrence of overfitting. In the case of limited data, the deeper network may be overfitting on the training data, which makes its performance on the test data worse. At the same time, the deeper network may be more difficult to train when the optimizer is certain.

Shallower + sigmond design

```
# shallower + sigmond
self.encoder = nn.Sequential(
    nn.Linear(input_dim, encoding_dim),
    nn.Sigmoid()
)
self.decoder = nn.Sequential(
    nn.Linear(encoding_dim, input_dim),
)
```



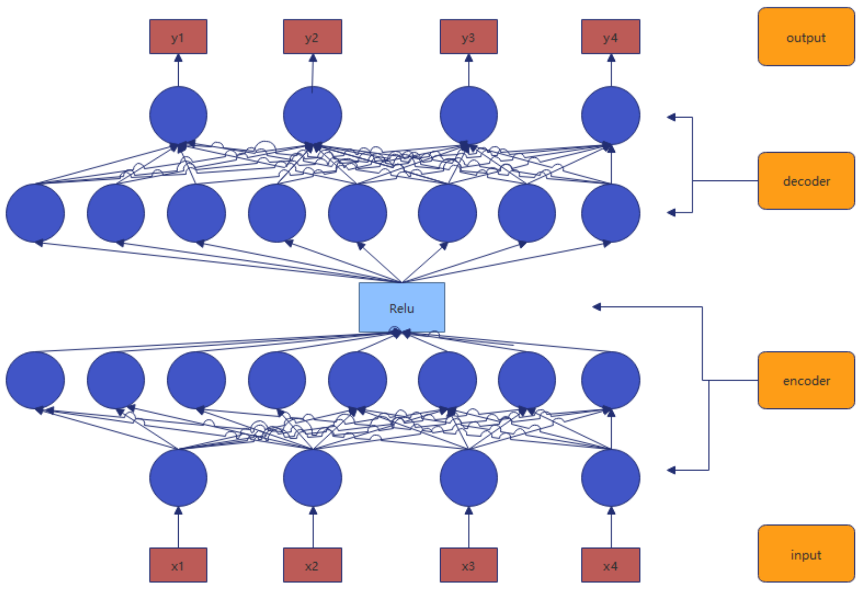
```
Acc from Autoencoder: 0.5333333333333333
Acc from DenoisingAutoencoder: 0.16666666666666666
```

```
Reconstruction Loss with Autoencoder: 0.028569871536348768
Reconstruction Loss with DenoisingAutoencoder: 0.03531376575089117
```

The second is a shallower network with sigmond activation function instead of relu, whose accuracy rate AE (0.53) and Denoise AE (0.17) are much lower than the original accuracy rate; In terms of Reconstruction Loss, AE and DenoiseAE were 0.029 and 0.035, respectively, indicating a significant increase compared to the original design. I think the reason is that underfitting occurs, and shallow networks may not have sufficient ability to learn complex data representations. This leads to significant errors when attempting to reconstruct input data, resulting in increased reconstruction losses and a significant decrease in accuracy; At the same time, the sigmoid activation function may limit the representation ability of the network. The output of the sigmoid function is between 0 and 1, which may not fully show all the characteristics of the data. The gradient of the original relu function in the positive part is 1, which will not have the problem of gradient disappearance. The result is more stable, reducing reconstruction loss and improving accuracy.

Fatter design

```
# fatter
self.encoder = nn.Sequential(
    nn.Linear(input_dim, encoding_dim),
    nn.Linear(encoding_dim, encoding_dim*2),
    nn.ReLU()
)
self.decoder = nn.Sequential(
    nn.Linear(encoding_dim*2, encoding_dim),
    nn.Linear(encoding_dim, input_dim),
)
```



```
Acc from Autoencoder: 0.9333333333333333
Acc from DenoisingAutoencoder: 0.9333333333333333
```

```
Reconstruction Loss with Autoencoder: 0.013918745855689495
Reconstruction Loss with DenoisingAutoencoder: 0.012682685163480024
```

The above are the three variations and original forms that I have shown that should be designed for this issue. The third type is transformed into a fatter network, with accuracy AE and DenoiseAE both reaching 0.93, which is relatively stable compared to the original accuracy; In terms of Reconstruction Loss, AE and DenoiseAE are 0.013 and 0.012 respectively, which are relatively stable and slightly improved compared to the original design. I think the reason is that it increases the width of the network with more hidden units, which can learn and store more information. Such a network may be more able to capture complex patterns in the data and generate lower errors when reconstructing inputs. This deeper network comparison with the first variant may be less likely to cause overfitting in the same situation, so it is more stable or even slightly improved compared with the original result.

The above are the three forms of variation and original design that I have presented for this issue, and the results and analysis are presented, with a focus on the Reconstruction Loss aspect. In fact, in the analysis, I believe that the impact and results of changing neural networks are similar to ordinary neural network models. However, in AE and DenoiseAE, it is mainly more convenient and advanced in image processing, which also shows me the accuracy of image processing and the relationship between reconstruction loss and layer number and layer width design.

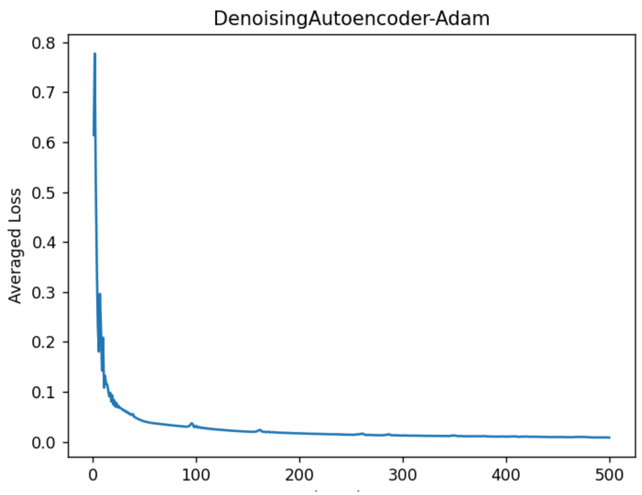
（e） Test at least 2 different optimizers, compare the training curve of DenoisingAutoencoder and discuss what you have found in terms of the convergence speed and overall performance of the model.

Answer:

An optimizer is an algorithm or method used in machine learning and deep learning to improve model performance and reduce prediction errors. The main objective of the optimizer is to minimize (or maximize) the loss function or objective function. In the neural network, the loss function measures the difference between the predicted value of the model and the actual value. The task of the optimizer is to find the parameters that can minimize the loss function (i.e. the weight and deviation of the model). I have selected two optimizers based on the requirements of the topic - Adam and SGD. Below, I will display their loss curves in DnoiseAE and explain them separately. Finally, I will make a comparative analysis. The generated image code is basically similar to Q1, so the image code part is not displayed.

```
# Adam
optimizer = optim.Adam(self.parameters(), lr=learning_rate)
# SGD
# optimizer = optim.SGD(self.parameters(), lr=learning_rate, momentum=0.7)
X_fit = torch.tensor(X).clone().detach().float()

data_pre = torch.utils.data.TensorDataset(X_fit)
```

Adam Optimizer

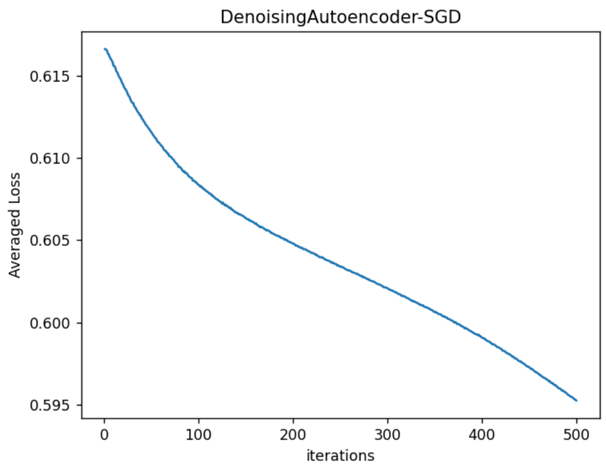


```
Acc from DenoisingAutoencoder: 0.9333333333333333
```

```
Reconstruction Loss with DenoisingAutoencoder: 0.013592173395249749
```

Adam (Adaptive Moment Estimation) is an optimization algorithm for deep learning models, particularly suitable for processing large-scale data and parameters. Adam combines the advantages of two other optimization algorithms: Adaptive Gradient Algorithm and Root Mean Square Propagation. Adam adjusts the learning rate separately for each parameter, which makes it particularly suitable for sparse data and non-uniform objective functions. At the same time, Adam also integrates the concept of momentum, which can help the optimizer skip local minima and accelerate convergence. Finally, Adam uses a deviation correction mechanism to obtain more accurate estimates faster from the initial stage of initialization and low gradient regions.

SGD Optimizer



```
Acc from DenoisingAutoencoder: 0.8666666666666667
```

```
Reconstruction Loss with DenoisingAutoencoder: 0.6826610005493821
```

SGD (Stochastic Gradient Descent) is a commonly used optimization algorithm in deep learning, used to train various types of models, especially neural networks. Random gradient descent only uses one training sample or a small batch of training samples during each update, which enables SGD to train faster and effectively handle large datasets. A common improvement method for SGD is to use momentum, which can help SGD accelerate in relevant directions and suppress oscillations, enabling it to converge to the optimal solution faster and more accurately.

The Adam optimizer has a significantly faster convergence speed in the results I have shown, with lower accuracy (0.93) and reconstruction loss (0.13). The final result and performance are both good. Compared to Adam, SGD has lower performance, only achieving an accuracy of 0.87 and a reconstruction loss of 0.68, while the convergence speed is slower. By comparing the results of Adam and SGD optimizers and searching for relevant information, I have made the following discussion: In terms of convergence speed, the Adam optimizer accelerates the learning process by calculating first-order moment estimation and second-order moment estimation of gradients. This adaptive learning rate adjustment method allows Adam to converge faster, so in the early stages of training, Adam usually performs better than SGD; In terms of reconstruction loss: The Adam optimizer can adjust the learning rate based on the gradient of each parameter, which can more effectively optimize complex functions, potentially resulting in lower reconstruction loss and higher accuracy; Overall performance: Although SGD can achieve better accuracy in certain situations, in many tasks, the Adam optimizer is usually more stable and does not require manual adjustment of learning rates. This makes Adam the preferred optimizer for many deep learning applications. But this does not mean that it is the best optimizer in all situations. Choosing the best optimizer often depends on the specific task, model architecture, and dataset. In some cases, such as when the dataset is very large or noisy, other optimizers such as SGD or RMSProp may perform better. In summary, this also provides us with advice and guidance on selecting an optimizer.