Yen Jui Chieh
B09902136

Homework #4
CSIE3005 Foundations of Artificial Intelligence

June 9, 2023

1 Hand-written Part

Problem 1

The first derivative of Swish is

$$\varphi'(s) = \theta(s) + s \cdot \theta'(s) = \frac{1}{1+e^{-s}} + s \cdot \frac{e^{-s}}{(1+e^{-s})^2} = \frac{1+e^{-s}+se^{-s}}{(1+e^{-s})^2}.$$

Problem 2

(A) Initially, we have $\mathbf{v}_0 = [1/3, 1/3, 1/3]^T$. Therefore,

$$\mathbf{v}_1 = \mathbf{P}\mathbf{v}_0 = \begin{bmatrix} 0 & 1 & 1/2 \\ 0 & 0 & 1/2 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1/3 \\ 1/3 \\ 1/3 \end{bmatrix} = \begin{bmatrix} 1/2 \\ 1/6 \\ 1/3 \end{bmatrix},$$

$$\mathbf{v}_2 = \mathbf{P}\mathbf{v}_1 = \begin{bmatrix} 0 & 1 & 1/2 \\ 0 & 0 & 1/2 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1/2 \\ 1/6 \\ 1/3 \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/6 \\ 1/2 \end{bmatrix},$$

$$\mathbf{v}_3 = \mathbf{P}\mathbf{v}_2 = \begin{bmatrix} 0 & 1 & 1/2 \\ 0 & 0 & 1/2 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1/3 \\ 1/6 \\ 1/2 \end{bmatrix} = \begin{bmatrix} 5/12 \\ 1/4 \\ 1/3 \end{bmatrix},$$

$$\mathbf{v}_4 = \mathbf{P}\mathbf{v}_3 = \begin{bmatrix} 0 & 1 & 1/2 \\ 0 & 0 & 1/2 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 5/12 \\ 1/4 \\ 1/3 \end{bmatrix} = \begin{bmatrix} 5/12 \\ 1/6 \\ 5/12 \end{bmatrix},$$

$$\mathbf{v}_5 = \mathbf{P}\mathbf{v}_4 = \begin{bmatrix} 0 & 1 & 1/2 \\ 0 & 0 & 1/2 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 5/12 \\ 1/6 \\ 5/12 \end{bmatrix} = \begin{bmatrix} 3/8 \\ 5/24 \\ 5/12 \end{bmatrix}.$$

(B) Let $\mathbf{v}^* = [a, b, c]^T$. Then

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1/2 \\ 0 & 0 & 1/2 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} \iff \begin{cases} a = b + c/2 \\ b = c/2 \\ c = a \end{cases},$$

which gives us $a = 2b = c$ after further simplification. Since $\mathbf{v}^*$ is normalized, we have $\mathbf{v}^* = [2/5, 1/5, 2/5]^T$.

Problem 3

- When $L = 1$, there is only one possible neuron network architecture $\left(d^{(0)}, d^{(1)}\right) = (10, 100)$. It's total number of weights is 1000.
- When $L = 2$, the total number of weights equals

$$10 \cdot d^{(1)} + d^{(1)} \cdot d^{(2)} = 10 \cdot d^{(1)} + d^{(1)} \cdot \left(100 - d^{(1)}\right) = -\left(d^{(1)} - 55^2\right) + 3025\,.$$

  With $0 < d^{(1)} < 100$ we know that

  - The minimum total number of weights is 109, achieved when $\left(d^{(0)}, d^{(1)}, d^{(2)}\right) = (10, 1, 99)$.
  - The maximum total number of weights is 3025, achieved when $\left(d^{(0)}, d^{(1)}, d^{(2)}\right) = (10, 55, 45)$.

- When $L = 3$, the total number of weights equals

$$\begin{aligned} 10 \cdot d^{(1)} + d^{(1)} \cdot d^{(2)} + d^{(2)} \cdot d^{(3)} &= 10 \cdot d^{(1)} + d^{(1)} \cdot d^{(2)} + d^{(2)} \cdot \left(100 - d^{(1)} - d^{(2)}\right) \\ &= 10 \cdot d^{(1)} + 100 \cdot d^{(2)} - \left(d^{(2)}\right)^2\,. \end{aligned} \tag{$\star$}$$

  Observe that

  - ($\star$) equals

$$10 \cdot d^{(1)} - \left(d^{(2)} - 50\right)^2 + 2500\,,$$

    and it's clear that $\left(d^{(0)}, d^{(1)}, d^{(2)}, d^{(3)}\right) = (10, 1, 1, 98)$ minimizes the total number of weights, which is 109.
  - when $d^{(2)}$ is fixed, in order to maximize ($\star$), $d^{(1)}$ should be as large as possible. Hence $d^{(1)} = 99 - d^{(2)}$. Then ($\star$) equals

$$-\left(d^{(2)} - 45\right)^2 + 3015\,,$$

    so $\left(d^{(0)}, d^{(1)}, d^{(2)}, d^{(3)}\right) = (10, 54, 45, 1)$ maximizes the total number of weights, which is 3015.

With the discussion above, we can finally conclude that

- The minimum total number of weights is 109, whose corresponding neuron network architecture is $\left(d^{(0)}, d^{(1)}, d^{(2)}, d^{(3)}\right) = (10, 1, 1, 98)$.
- The maximum total number of weights is 3025, whose corresponding neuron network architecture is $\left(d^{(0)}, d^{(1)}, d^{(2)}\right) = (10, 55, 45)$.

2 Programming Part

(a)

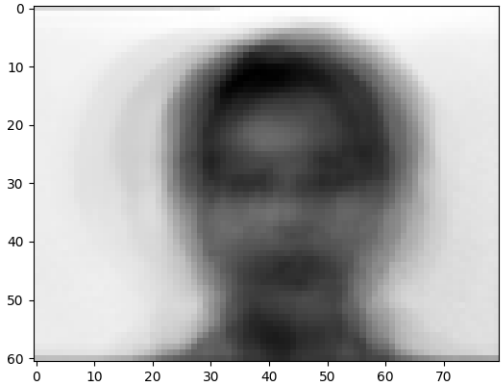
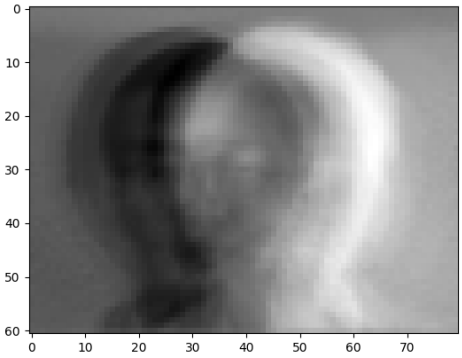
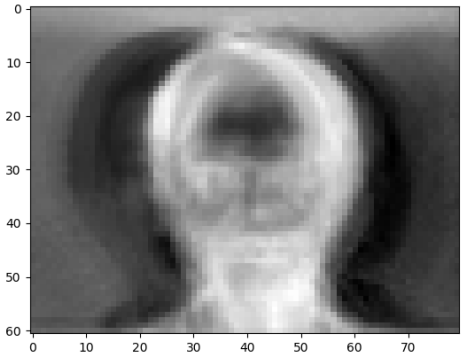


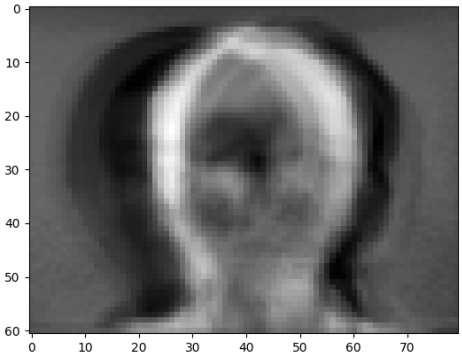
Figure 1: Mean vector.



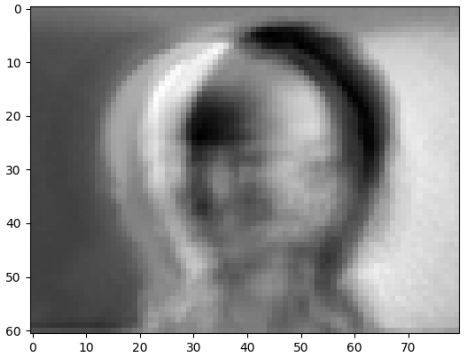
(a) 1st eigenvector.



(b) 2nd eigenvector.



(c) 3rd eigenvector.



(d) 4th eigenvector.

Figure 2: Top 4 eigenvectors.

(b)

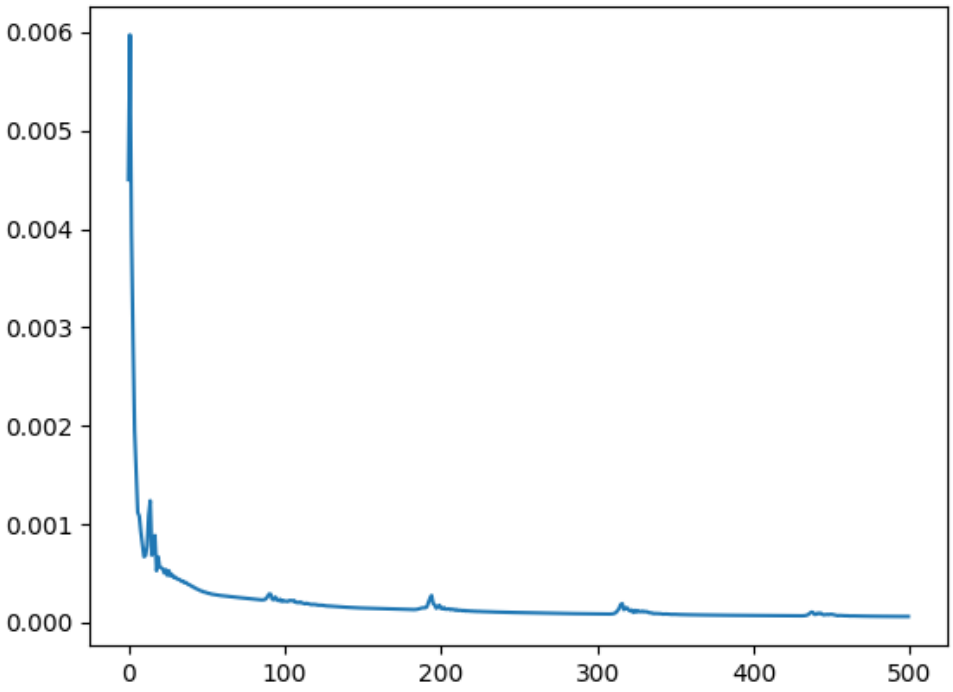


Figure 3: Training curve of `Autoencoder`.

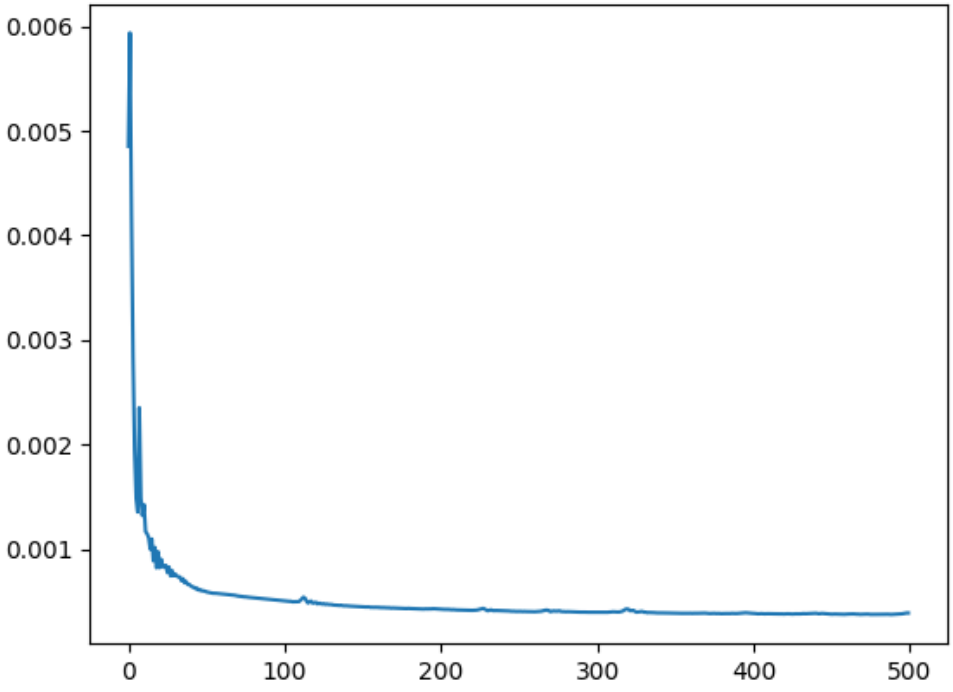


Figure 4: Training curve of `DenoisingAutoencoder`.

(c)

Figure 5: Original image.

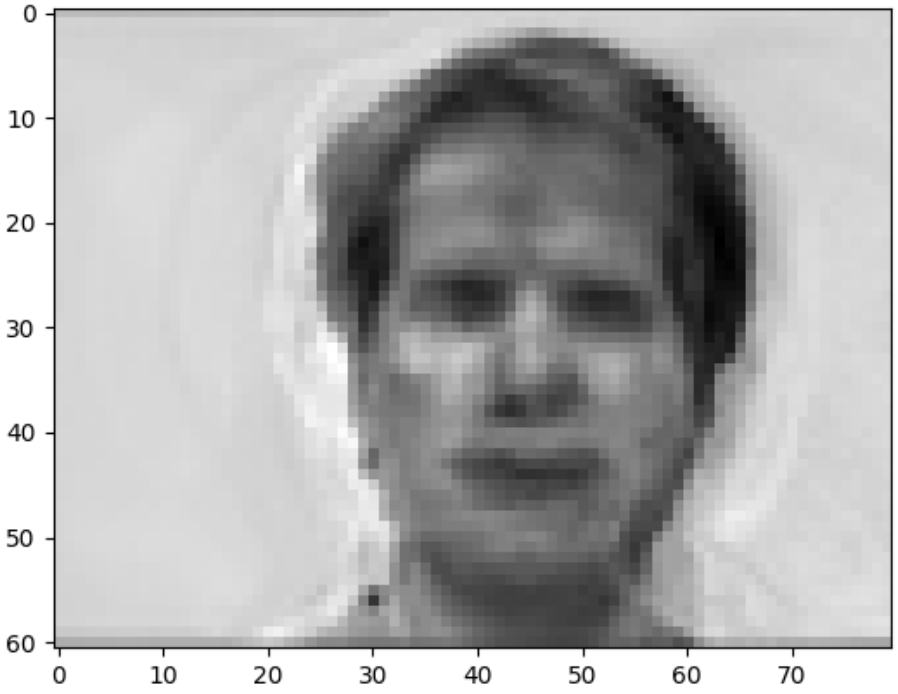


Figure 6: Reconstructed with `PCA` (MSE = 0.010710469688056315).

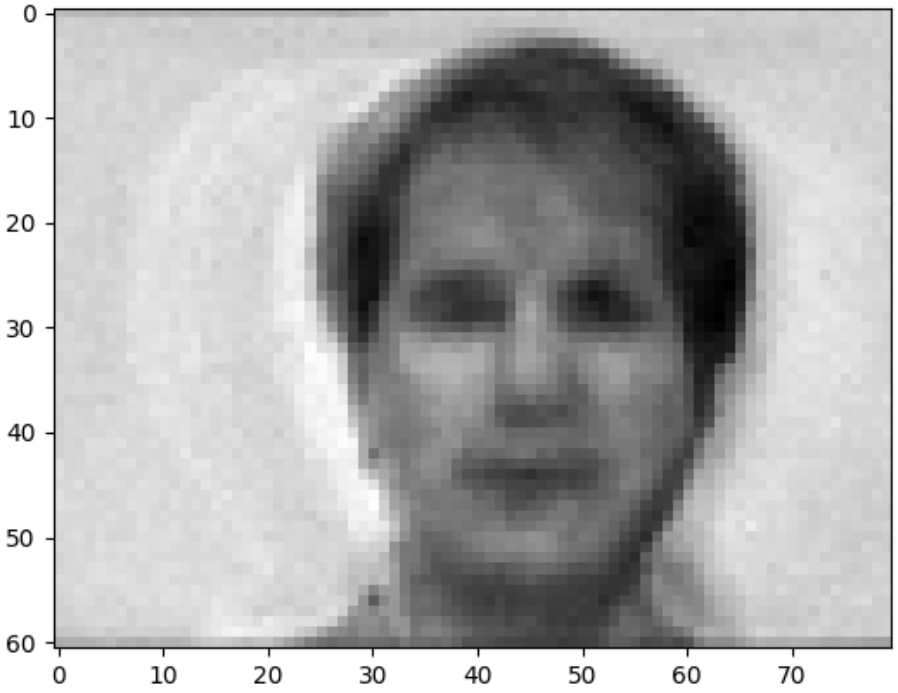


Figure 7: Reconstructed with `Autoencoder` (MSE = 0.013401435366304601).

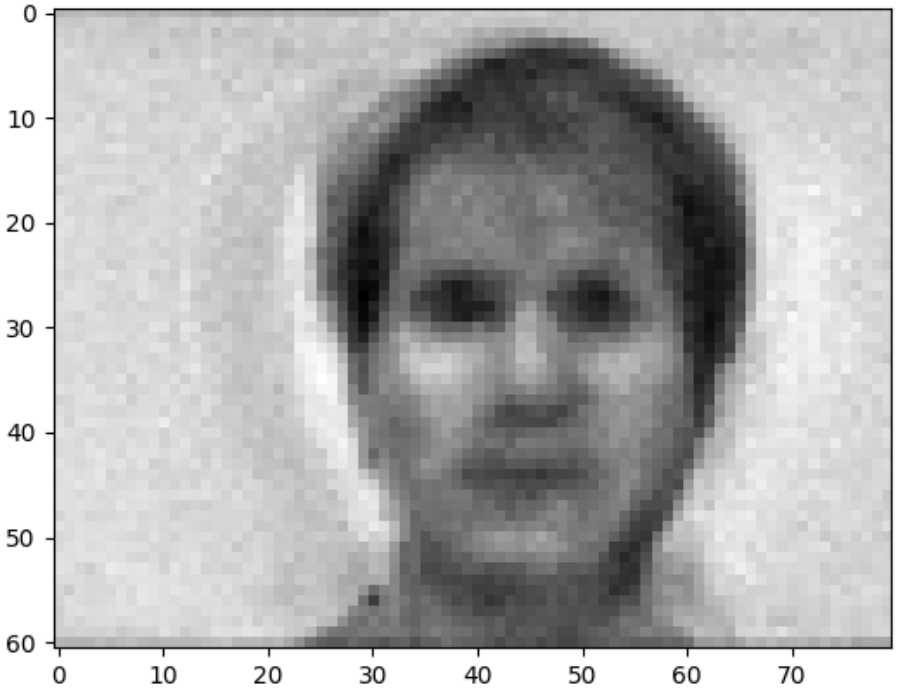


Figure 8: Reconstructed with `DenoisingAutoencoder` (MSE = 0.015951452182278912).

(d) I've tried two different network architectures for the denoising autoencoder, both with `Adam` optimizer. The first one is a deeper network.

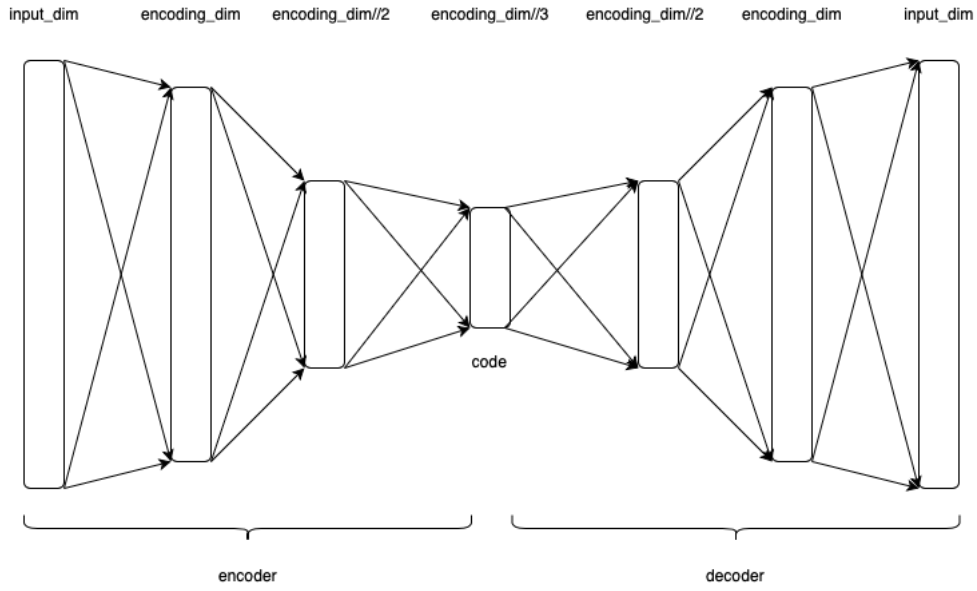


Figure 9: A deeper network architecture.

The network architecture above achieves a larger reconstruction loss of 0.0350080289782967. I assume this is because more layers in the encoder increases the difficulty of decent decoding. Its accuracy also drops to a shocking 0.13333333333333333. The second modified network architecture is a shallow one.

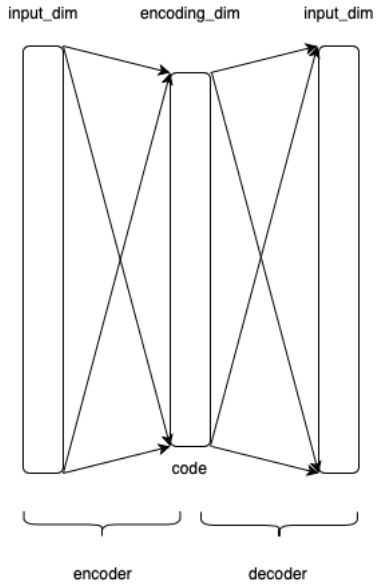
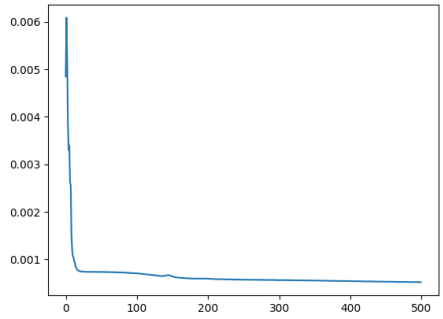


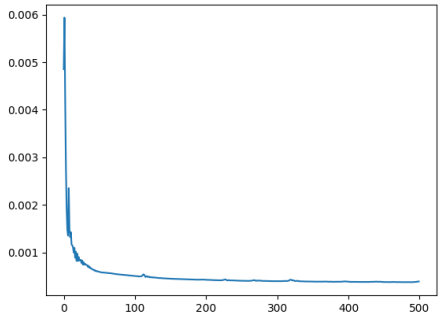
Figure 10: A shallower network architecture.

This one gives us a reconstruction loss of 0.02097132573524421, which is still higher than the original network, but lower than the former architecture. It's accuracy is 0.8666666666666667. I've never modified most other factors, such as the number and types of activation functions, and the optimizer choice. Therefore, I suppose that *the reconstruction performance will be worse if the neural network is either too deep or too shallow.*

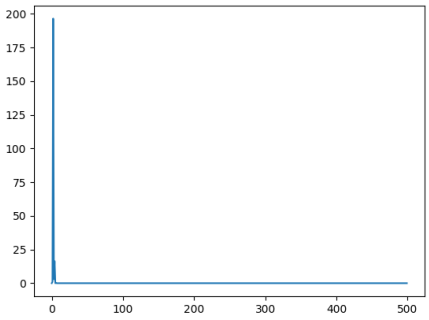
(e) I've tested on `DenoisingAutoencoder` a total of four different optimizers: `AdaGrad`, `Adam`, `RMSProp`, and `SGD`. The learning rate is fixed at 0.001 in all cases. Their training curves are plotted below.



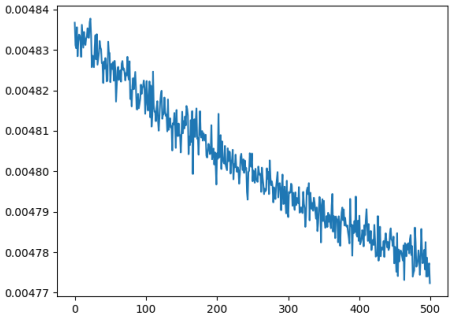
(a) `AdaGrad`.



(b) `Adam`.



(c) `RMSProp`.



(d) `SGD`.

Their overall performances are listed below.

	accuracy	reconstruction loss
`AdaGrad`	0.8333333333333334	0.0280905105901022
`Adam`	0.9333333333333333	0.015951452182278912
`RMSProp`	0.06666666666666667	0.1448340786671235
`SGD`	0.9333333333333333	0.6945623768957014

These are funny results. The accuracy of `RMSProp` and the convergence speed of `SGD` are so terrible that I start worrying about the correctness of my implementation. The reconstruction loss of `SGD` and `RMSProp` are a lot higher than the rest. However, `RMSProp` has the fastest convergence speed, and `SGD` ties with `Adam` in terms of accuracy. The overall performance of `Adam` is the best, so it's employed in my final submission.