ECE/CS 559: Neural Networks Homework 5

Samuele Pasquale

October 2, 2024

1. Question 1:

   (a) Given the risk function with Given the risk function with $w \in \mathbb{R}^2$

   $$R(w) = 13w_1^2 - 10w_1w_2 + 4w_1 + 2w_2^2 - 2w_2 + 1 \tag{1}$$

   The optimality condition is:

   $$\nabla_w R = 0 \tag{2}$$

   Therefore, the gradient of the risk function can be calculated as:

   $$\nabla_w R = \begin{bmatrix} \frac{\partial R}{\partial w_1} \\ \frac{\partial R}{\partial w_2} \end{bmatrix} = \begin{bmatrix} 26w_1 - 10w_2 + 4 \\ -10w_1 + 4w_2 - 2 \end{bmatrix} \tag{3}$$

   Lastly, the optimality condition can be solved as:

   $$\begin{cases} 26w_1 - 10w_2 + 4 = 0 \\ -10w_1 + 4w_2 - 2 = 0 \end{cases} \quad \Rightarrow \quad \begin{cases} w_1 = 1 \\ w_2 = 3 \end{cases} \tag{4}$$

   (b) The Figure 1 shows the plot of the distance between the iterate and the optimal solution of the gradient descend over each iteration for different η's.

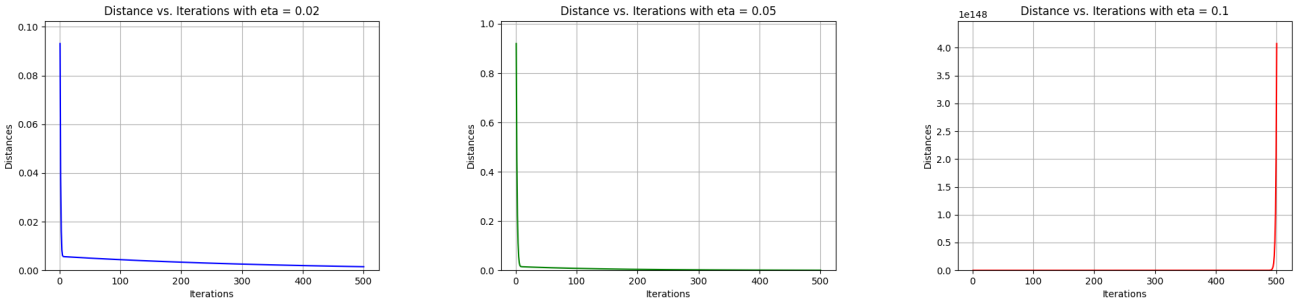

Figure 1: Distance vs. Iterations with $\eta = 0.02$, $\eta = 0.05$ e $\eta = 0.1$.

(c) As can be seen from the graphs in Figure 1, varying the learning rate η can either increase or decrease the distance from the optimal solution. The learning rate influences the size of the gradient update steps during the iterations. In particular, a small η allows for slower but more stable updates, while a larger η allows for faster updates but less stability. The graphs show both behaviors; indeed, by increasing η from 0.02 to 0.05, the value of η remains small enough to converge to a distance of 0 but more quickly (steeper graph). On the other hand, if η is increased too much (e.g., from 0.05 to 0.1), the distance diverges, highlighting the instability of choosing a η that is too large. The instability arises from the fact that, with updates being too large, the gradient descent toward a global or local minimum may be interrupted by a step that is too large, which would skip over the minimum, causing oscillations or, in the worst case, divergence.

2. Question 2:

(a) The sigmoid function φ and its derivative have been implemented in Python as follows:

```python
def sigmoid(x, a):
    """
    Function to calculate the sigmoid function with parameter a
    :param x: x
    :param a: a parameter
    :return:
    """
    # return the sigmoid function given x and a
    return 1 / (1 + np.exp(-a * x))

def sigmoid_derivative(x, a):
    """
    Function to calculate the derivative of the sigmoid function
    :param x:
    :param a: a parameter
    :return:
    """
    # compute the sigmoid function given x and a
    sig = sigmoid(x, a)
    # return the derivative
    return a * sig * (1 - sig)
```

(b) The dimensions of all weights, biases, and variables are:

$$\begin{aligned} W \in \mathbb{R}^{3\times 2} \qquad U \in \mathbb{R}^{1\times 3} \qquad v_z \in \mathbb{R}^{3\times 1} \\ b \in \mathbb{R}^{3\times 1} \qquad c \in \mathbb{R}^{1\times 1} \qquad z \in \mathbb{R}^{3\times 1} \\ x \in \mathbb{R}^{2\times 1} \qquad f \in \mathbb{R}^{1\times 1} \qquad v_f \in \mathbb{R}^{1\times 1} \end{aligned} \tag{5}$$

The forward equations from input x to the output f are:

$$x \quad \rightarrow \quad v_z = W \cdot x + b \quad \rightarrow \quad z = \varphi(v_z) \quad \rightarrow \quad v_f = U \cdot z + c \quad \rightarrow \quad f = \varphi(v_f)$$

The forward has been implemented in python as follows:

```python
# Forward (x -> vz -> z -> vf -> f)
x = [[x1_val], [x2_val]]
vz = v_z(W, x, b)
z_vect = z(vz, a)
vf = v_f(U, z_vect, c)
f_out = f(vf, a)
```

(c) The backward equations, given the loss function $l(y, f) = (f - y)^2$, are:

$$\nabla_f l = \frac{\partial l}{\partial f} = 2(f - y) \quad \rightarrow \quad \delta_f = \frac{\partial l}{\partial v_f} = \nabla_f l \cdot \varphi'(v_f)$$

$$\nabla_z l = U^T \cdot \delta_f \quad \rightarrow \quad \delta_z = \nabla_z l \cdot \varphi'(v_z)$$

$$\nabla_U l = \delta_f \cdot z^T \qquad \nabla_C l = \delta_f$$

$$\nabla_W l = \delta_z \cdot x^T \qquad \nabla_b l = \delta_z$$

The backward has been implemented in python as follows:

```
# Backward (grad_b_l <- grad_W_l <- delta_z <- grad_z_l <- grad_c_l <- grad_U_l
    <- delta_f <- grad_f_l)
grad_f_l  = gradient_f_l(f_out, y_val)
del_f = delta_f(grad_f_l, vf, a)
grad_u_l = np.array(del_f) @ np.array(z_vect).T
grad_c_l = del_f
grad_z_l = gradient_z_l(del_f, U)
del_z = delta_z(grad_z_l, vz, a)
grad_w_l = np.array(del_z) @ np.array(x).T
grad_b_l = del_z
```

(d) After implementing the training algorithm with forward and backward propagation, the MSE trend over the epochs is shown in Figure 2.

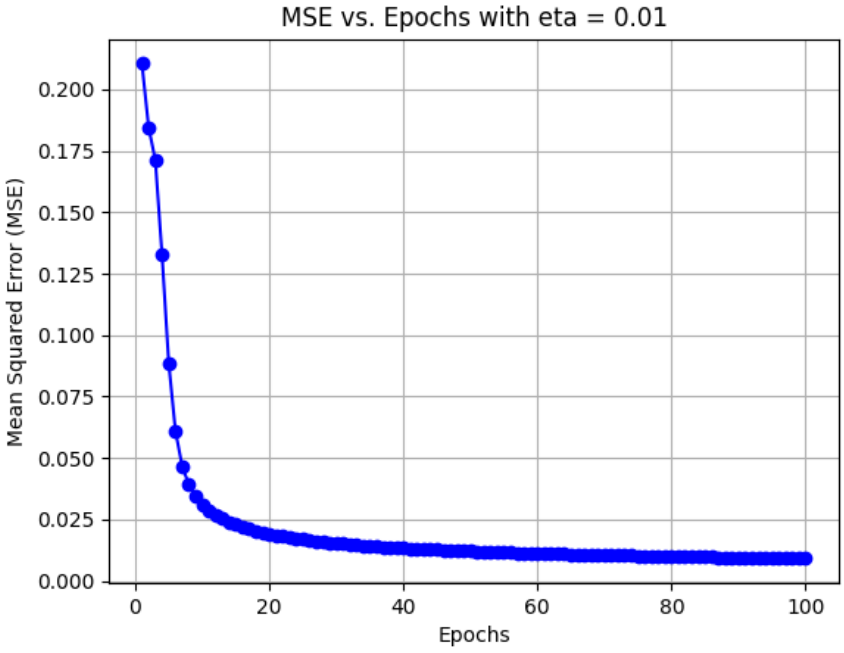

Figure 2: MSE trend over the epochs

(e) The plot of the decision boundary is shown in Figure 3.

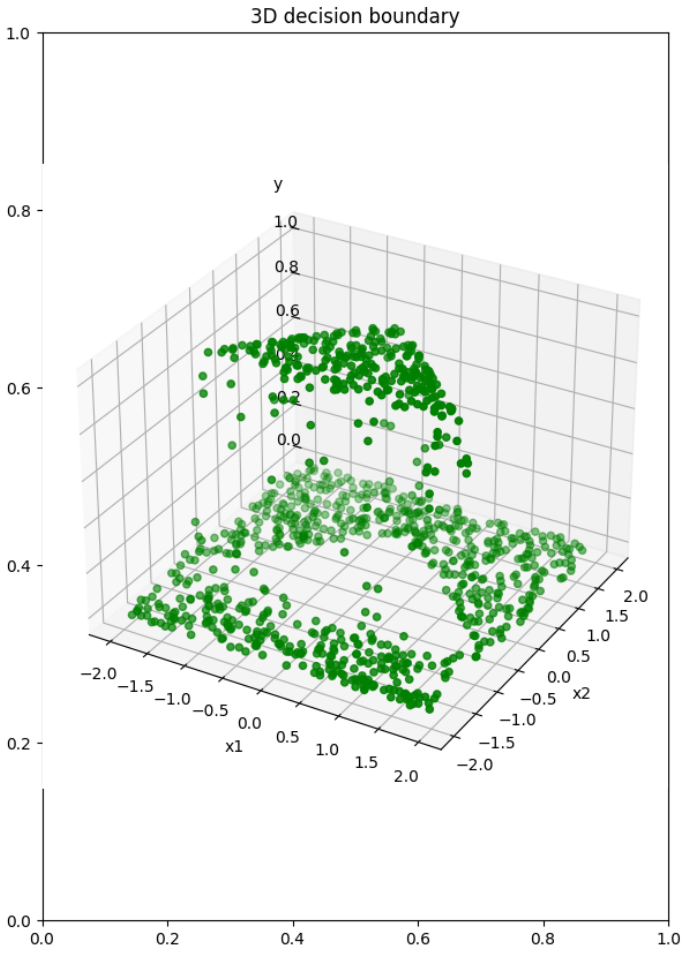

Figure 3: 3D decision boundary

(f) All the variants have been implemented. The results of each are shown below, followed by some final comments.

i. **Different η values**: As can be seen in Figure 4, by changing η, different behaviors are observed. In particular, with a smaller η, the algorithm is much more stable (MSE trend is more regular), while as η increases, the oscillations consequently increase. On the other hand, a larger η allows for faster convergence. As a matter of fact, with $\eta = 0.001$, 100 epochs are not enough to obtain a clear 3D decision boundary.

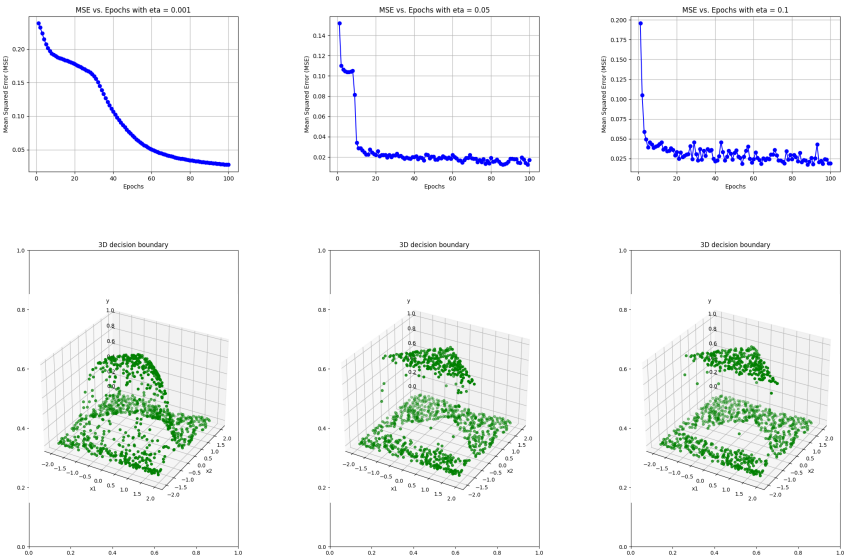

Figure 4: MSE and plot comparisons for different values of η.

ii. **Variable η based on the MSE value**: In this implementation, η is reduced by a factor of 0.95 when the MSE increases. With the example shown in Figure 5, it is highlighted that in this way, the oscillations obtained with an initial $\eta = 0.1$ are reduced, at the expense of slower convergence as η decreases. With 100 epochs, the result is satisfactory; however, with fewer epochs, the quality of the decision boundary could be worse, since the size of the gradient update steps gradually decreases, resulting in slower convergence.

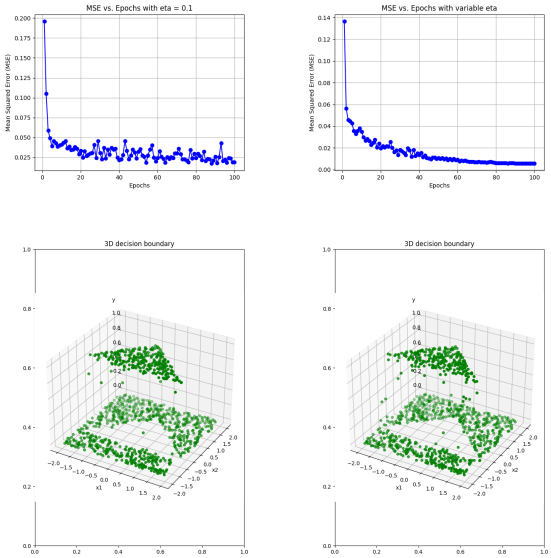

Figure 5: Comparison of MSE and plot results for variable η.

iii. **Different a parameters of the sigmoid function**: By modifying the parameter a of the sigmoid function, the slope of the function changes. The higher the value of a, the steeper

the slope. The slope affects the sensitivity of the neurons, leading to faster learning but with the risk of saturation near 0 or 1. From Figure 6, it can be observed that the quality of the decision boundary does not improve, and instability increases. As a result, the parameter a could be further reduced to increase stability, since the margin of epochs is still wide.

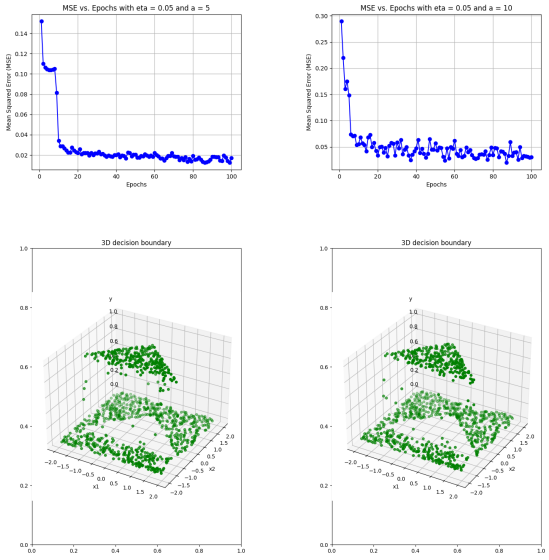

Figure 6: Comparison of MSE and plot results for different a's.

iv. **Different number of epochs**: By reducing the number of epochs, the number of passes over the dataset decreases, reducing the quality of the training. From Figure 7, it can be seen that this does not have a significant effect, particularly with $\eta = 0.01$ and parameter $a = 5$, but if, for example, η were smaller and/or a smaller, the number of epochs might need to be increased to achieve good training.

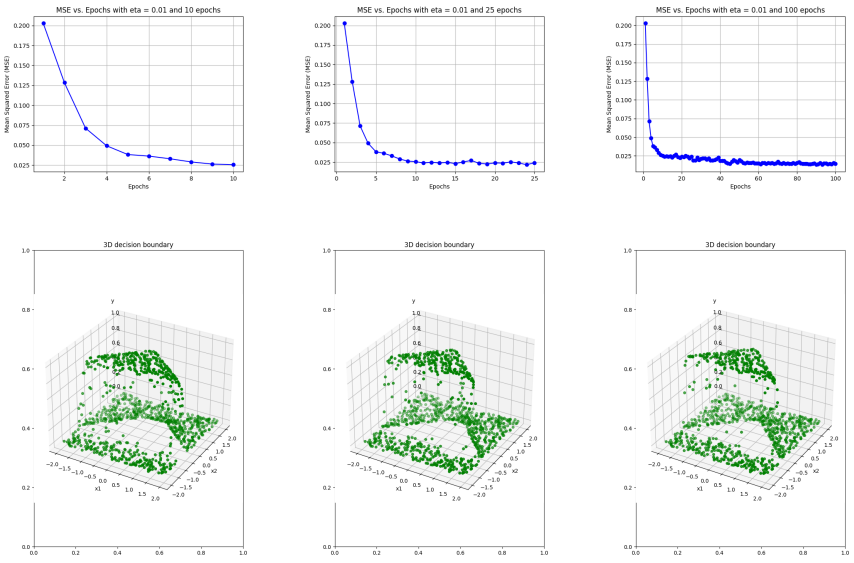

Figure 7: Comparison of MSE and plot results for different epochs.

v. **Proper gradient descent**: With Proper gradient descent, it can be observed that the number of updates is smaller, as these are made at the end of each epoch. As a result, a more stable MSE trend is obtained. On the other hand, by reducing the number of updates, 100 epochs are not sufficient to achieve an optimal decision boundary. Thus, even in this case, the stability causes a slowdown in the training of the neural network.

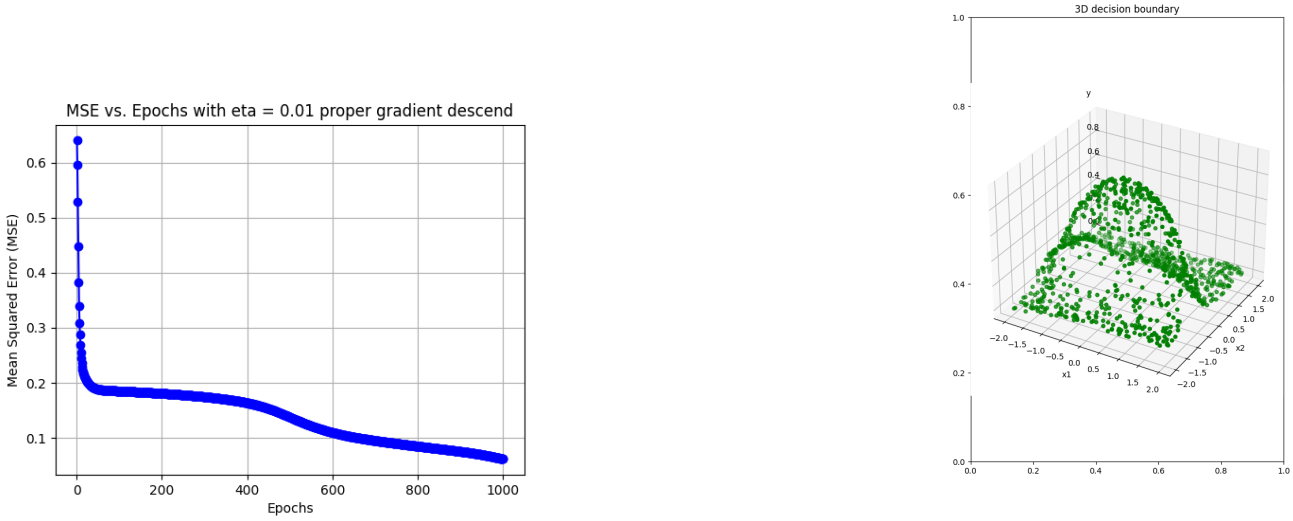

Figure 8: Comparison of MSE and plot results for proper gradient descent.

vi. **Minibatches**: In Figure 9, it can be seen how the effect of Proper gradient descent can be mitigated by reducing the batch size. In particular, the smaller the batch size, the more frequent the updates, and thus the better the decision boundary. The use of minibatches can therefore be a good trade-off between greater stability (inherited from Proper gradient

descent) and faster learning (by reducing the batch size).

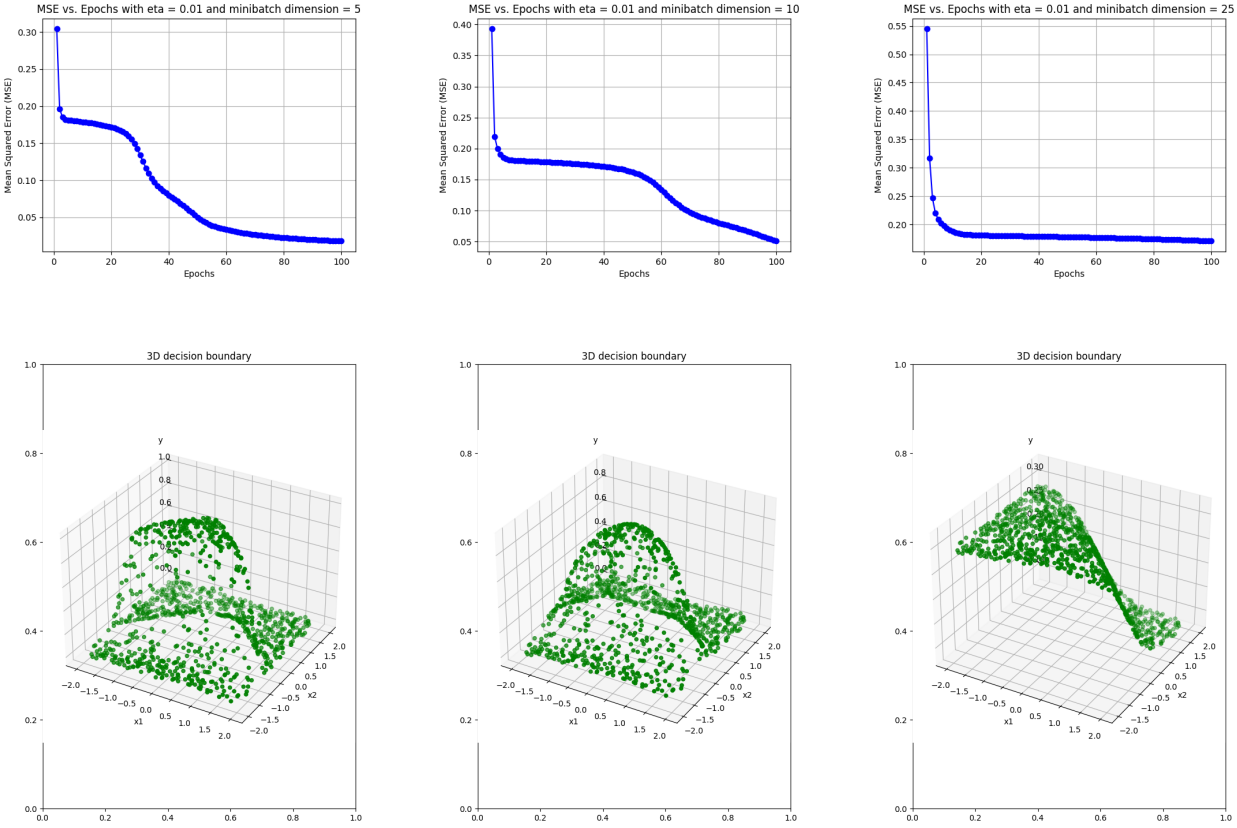

Figure 9: Comparison of MSE and plot results for different batch sizes.

vii. **Different architecture**: The architecture was changed by increasing the number of neurons in the hidden layer from 3 to 30. An increase in neurons in our case has a smaller impact. In particular, it allows for faster learning by the neural network. On the other hand, the difficulty of learning remains the same, so a real improvement is not visible with a simple problem like that of our dataset. In Figure 10, it can be observed that with only 8 batches, the quality of the training with more neurons is better. Therefore, the only advantage is a significant reduction in the number of epochs, which reflects in the training time of the neural network.

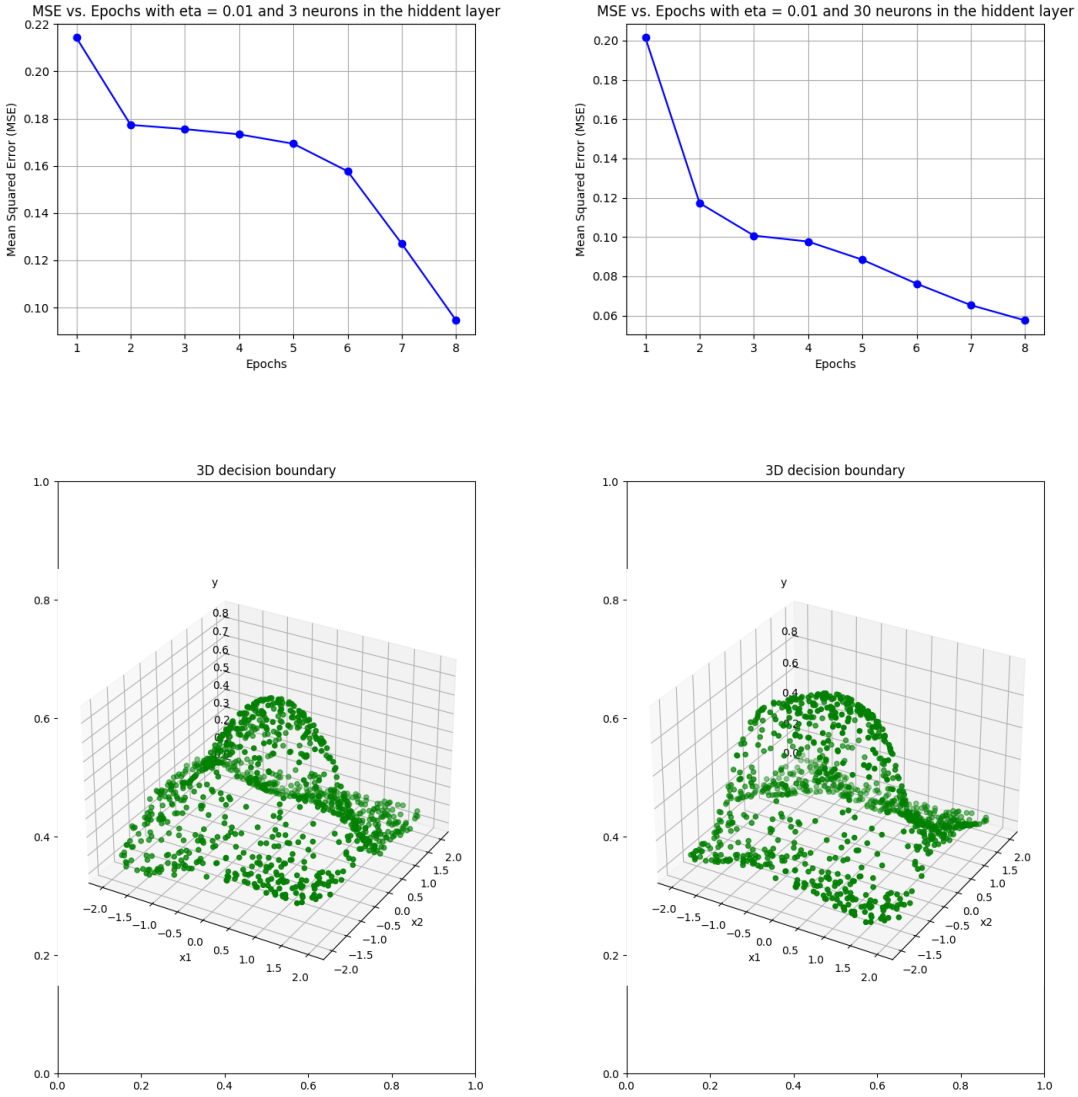

Figure 10: Comparison of MSE and plot results for different numbers of neurons.

viii. **Change in the standard deviation for the random generation of weights and biases**: By varying the standard deviation, random generations are more or less close to 0. With a smaller standard deviation (e.g., $\sigma = 0.1$), the gradients will initially be smaller, and thus less sensitive to inputs but more stable during learning. With a larger standard deviation (e.g., $\sigma = 0.5$), learning will be faster but more unstable. As can be seen from Figure 11, in the initial phase, the MSE descent is steeper. On the other hand, larger gradients and greater oscillations cause less convergence compared to a smaller standard deviation. As a result, the decision boundary is not satisfactory. To sum up, a smaller standard deviation allows the algorithm to be more stable and converge more, at the cost of requiring more epochs. A larger standard deviation, due to the oscillations and instability, can lead to unsatisfactory training.

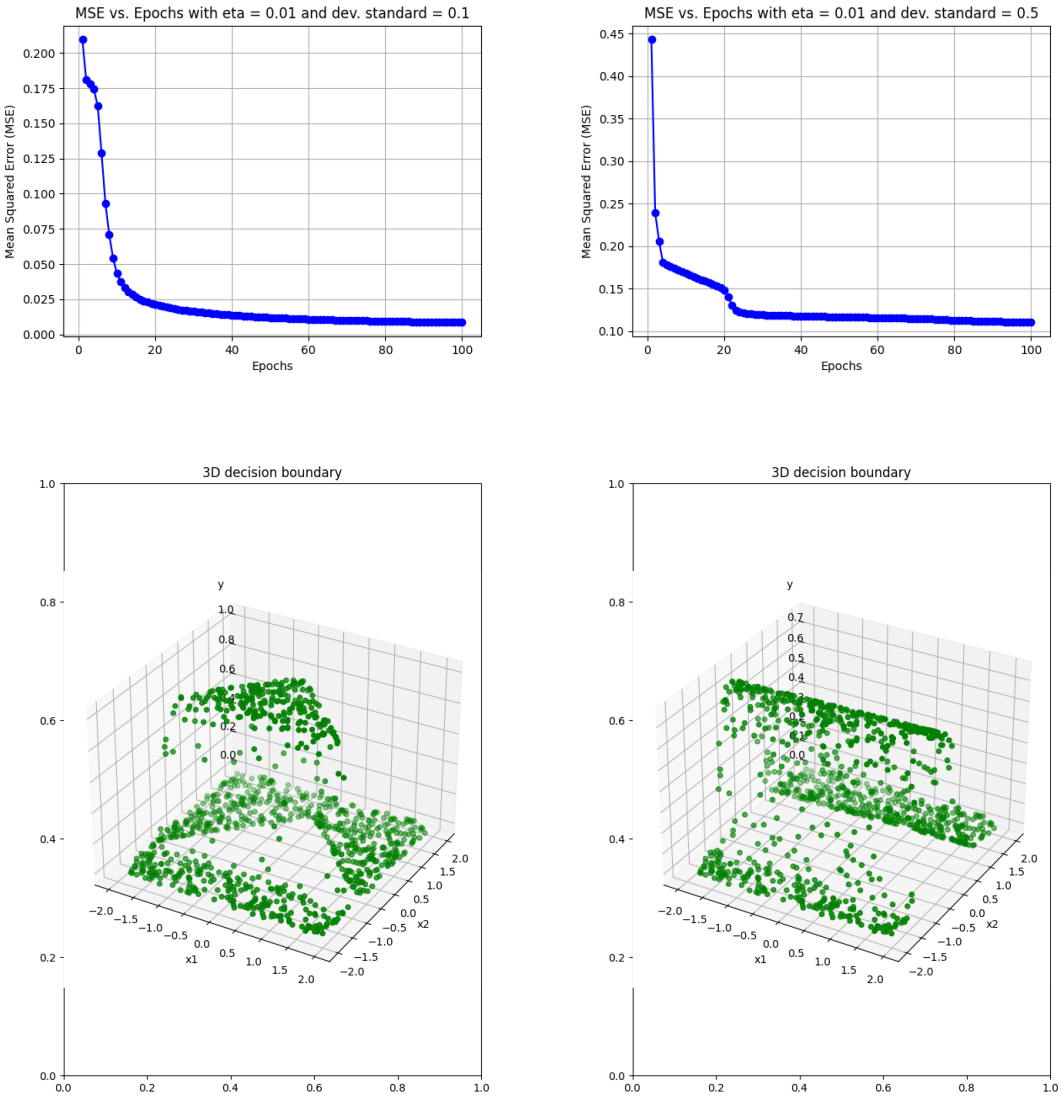

Figure 11: Comparison of MSE and plot results for different standard deviations.