Lab 1 - Back-propagation

1. Introduction

In this lab, I will implement a simple Multi-Layer Perceptron (MLP) model to tackle a binary classification problem. Our objective is to understand and execute the forward and backward pass processes, exploring how using two hidden layers can enhance the model's performance. I will also investigate various factors that influence the model's effectiveness, including:

- Trying Different Learning Rates: I will experiment with various learning rates to observe their impact on convergence and performance.
- Different Numbers of Hidden Units: By varying the number of hidden units in the layers, I aim to understand how model complexity affects classification results.
- Trying Without Activation Functions: I will explore the effects of removing activation functions to see how it influences the model's ability to learn complex patterns.

This experiment will provide insights into the fundamental structure and functioning of neural networks while allowing us to understand how different configurations affect the overall performance of the MLP.

2. Experiment setups

A. Sigmoid Functions

The Sigmoid function is a commonly used activation function, defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Its output range is (0, 1), making it suitable for binary classification tasks. The advantages of the Sigmoid function include:

- **Smoothness**: It is differentiable, allowing for backpropagation calculations.
- **Centering**: For inputs close to 0, the output is approximately 0.5, which stabilizes learning.

B. Neural Network

I will construct a neural network with two hidden layers, structured as follows:

- **Input Layer**: input size = 2
- **Fully connected Layer 1(fc1)**: Sigmoid activation function with 10 units
- **Fully connected Layer 2(fc2)**: Sigmoid activation function with 10 units
- **Output Layer**: A single neuron with a Sigmoid activation function, producing a predicted probability

C. Backpropagation

Backpropagation is a method used to compute gradients for the neural network, utilizing the chain rule for weight updates.

1. **Calculate Cost Function**: used cost function is Mean square error (MSE).

$$\mathrm{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \widehat{y_i})^2$$

2. **Compute Output Layer Gradients**: Differentiate the loss function with respect to the output.

$$\sigma'(z) = \sigma(z)\big(1 - \sigma(z)\big)$$

3. **Calculate Hidden Layer Gradients**: Use the chain rule to compute gradients for each layer.

For a single hidden layer, the gradients can be computed as:

$$gradient\ w^{\mathrm{L}} = \frac{\partial \mathrm{C}}{\partial \mathrm{w}^{\mathrm{L}}} = \frac{\partial \mathrm{z}^{\mathrm{L}}}{\partial \mathrm{w}^{\mathrm{L}}}\frac{\partial \mathrm{a}^{\mathrm{L}}}{\partial \mathrm{z}^{\mathrm{L}}}\frac{\partial \mathrm{C}}{\partial \mathrm{a}^{\mathrm{L}}} = \mathrm{a}^{\mathrm{L\text{-}1}}\delta^{\text{-}1}(\mathrm{z}^{\mathrm{L}})2(\mathrm{a}^{\mathrm{L}} - y)$$

$$gradient\ \mathrm{a}^{\mathrm{L}} = \frac{\partial \mathrm{C}}{\partial \mathrm{a}^{\mathrm{L\text{-}1}}} = \frac{\partial \mathrm{z}^{\mathrm{L}}}{\partial \mathrm{a}^{\mathrm{L\text{-}1}}}\frac{\partial \mathrm{a}^{\mathrm{L}}}{\partial \mathrm{z}^{\mathrm{L}}}\frac{\partial \mathrm{C}}{\partial \mathrm{a}^{\mathrm{L}}} = \mathrm{w}^{\mathrm{L}}\delta^{\text{-}1}(\mathrm{z}^{\mathrm{L}})2(\mathrm{a}^{\mathrm{L}} - y)$$

$$gradient\ \mathrm{b}^{\mathrm{L}} = \frac{\partial \mathrm{C}}{\partial \mathrm{w}^{\mathrm{L}}} = \frac{\partial \mathrm{z}^{\mathrm{L}}}{\partial \mathrm{b}^{\mathrm{L}}}\frac{\partial \mathrm{a}^{\mathrm{L}}}{\partial \mathrm{z}^{\mathrm{L}}}\frac{\partial \mathrm{C}}{\partial \mathrm{a}^{\mathrm{L}}} = \delta^{\text{-}1}(\mathrm{z}^{\mathrm{L}})2(\mathrm{a}^{\mathrm{L}} - y)$$

where z is the input of the activation function at the hidden layer, L is current layer, C is cost function, w is the weights of hidden layer, b is basis, a is the output after activation function

4. **Update Weights**: Update the weights for each layer based on the computed gradients and learning rate.

Here is SGD optimizer as an example,

$$w_{\square}^{L,\mathrm{new}} = w_{\square}^{L,\mathrm{old}} - \eta\frac{\partial \mathrm{C}}{\partial w_{\square}^{L}}$$

$$b_{\square}^{L,\mathrm{new}} = b_{\square}^{L,\mathrm{old}} - \eta\frac{\partial \mathrm{C}}{\partial b_{\square}^{L}}$$

D. Experiment Setup

In this experiment, I will explore the performance of our model by varying different hyperparameters and configurations. The following are the key aspects of our experimental setup:

a) Learning Rates

I will try different learning rates to see how they affect the convergence and performance of the model:

- [,0.001,0.01,0.1]

b) Optimizers

I will use different optimizer to update the model weights:

- SGD
- Momentum
- Adagrad
- Adam

c) Activation Functions

The experiment with different activation functions in the hidden layers:

- Sigmoid
- ReLU
- tanh
- without activation function

d) Hidden Layer Sizes

I will vary the number of hidden units to explore how the model's capacity affects its performance:

- [10,20,30]

e) Functions

The model was tested on different types of data generation functions to evaluate:

- generate_linear
- generate_XOR_easy

3. Results

Table 1. Experient parameter

Optimizer	Activation Function	Hidden size	Learning Rate
SGD	sigmoid	10	0.1

```
Learning Rate: 0.1, Hidden Size:10 Optimizer: SGD, Activation: sigmoid
epoch 0 loss : 0.34767138787125973
epoch 10000 loss : 0.013500445065910973
epoch 20000 loss : 0.008195428646830726
epoch 30000 loss : 0.006406685199266038
epoch 40000 loss : 0.005541475924103098
epoch 50000 loss : 0.005009944123432791
epoch 60000 loss : 0.004622349893779954
epoch 70000 loss : 0.004300723703807536
epoch 80000 loss : 0.004005526457058652
epoch 90000 loss : 0.003713165728402391
Iter 1 | Ground truth: [0] | Prediction: [0.00643077] |
Iter 2 | Ground truth: [0] | Prediction: [1.58001431e-05] |
Iter 3 | Ground truth: [1] | Prediction: [0.68121349] |
Iter 4 | Ground truth: [0] | Prediction: [1.61524122e-05] |
Iter 5 | Ground truth: [0] | Prediction: [1.12266358e-05] |
Iter 6 | Ground truth: [0] | Prediction: [1.05936929e-05] |
Iter 7 | Ground truth: [0] | Prediction: [1.3607682e-05] |
Iter 8 | Ground truth: [1] | Prediction: [0.99999832] |
Iter 9 | Ground truth: [1] | Prediction: [0.9999981] |
Iter 10 | Ground truth: [1] | Prediction: [0.99676208] |
Iter 11 | Ground truth: [0] | Prediction: [8.54552044e-06] |
Iter 12 | Ground truth: [1] | Prediction: [0.99999548] |
Iter 13 | Ground truth: [0] | Prediction: [1.76254952e-05] |
Iter 14 | Ground truth: [0] | Prediction: [3.11957501e-05] |
...
Iter 98 | Ground truth: [1] | Prediction: [0.99999851] |
Iter 99 | Ground truth: [0] | Prediction: [8.07101658e-06] |
Iter 100 | Ground truth: [0] | Prediction: [7.99274914e-06] |
Accuracy: 100.00%
```

Figure 1. indicate training loss and accuracy for generate_linear()

```
Learning Rate: 0.1, Hidden Size:10 Optimizer: SGD, Activation: sigmoid
epoch 0 loss : 0.2630092739967975
epoch 10000 loss : 0.024550752768769075
epoch 20000 loss : 0.003754835520409852
epoch 30000 loss : 0.001542449149859047
epoch 40000 loss : 0.0009085491907666594
epoch 50000 loss : 0.0006273723889722081
epoch 60000 loss : 0.0004727908568970940 3
epoch 70000 loss : 0.0003763727683520 1996
epoch 80000 loss : 0.000311038934172679
epoch 90000 loss : 0.0002641038641796 3737
Iter 1 | Ground truth: [0] | Prediction: [0.00393233] |
Iter 2 | Ground truth: [1] | Prediction: [0.99413327] |
Iter 3 | Ground truth: [0] | Prediction: [0.00330556] |
Iter 4 | Ground truth: [1] | Prediction: [0.99459643] |
Iter 5 | Ground truth: [0] | Prediction: [0.0047796] |
Iter 6 | Ground truth: [1] | Prediction: [0.99492665] |
Iter 7 | Ground truth: [0] | Prediction: [0.01088894] |
Iter 8 | Ground truth: [1] | Prediction: [0.99415989] |
Iter 9 | Ground truth: [0] | Prediction: [0.02275222] |
Iter 10 | Ground truth: [1] | Prediction: [0.96660307] |
Iter 11 | Ground truth: [0] | Prediction: [0.02845373] |
Iter 12 | Ground truth: [0] | Prediction: [0.02282395] |
Iter 13 | Ground truth: [1] | Prediction: [0.96499137] |
Iter 14 | Ground truth: [0] | Prediction: [0.01458366] |
...
Iter 19 | Ground truth: [1] | Prediction: [0.99979306] |
Iter 20 | Ground truth: [0] | Prediction: [0.0032481] |
Iter 21 | Ground truth: [1] | Prediction: [0.99982229] |
Accuracy: 100.00%
```

Figure 3. indicate training loss and accuracy for generate_XOR_easy()

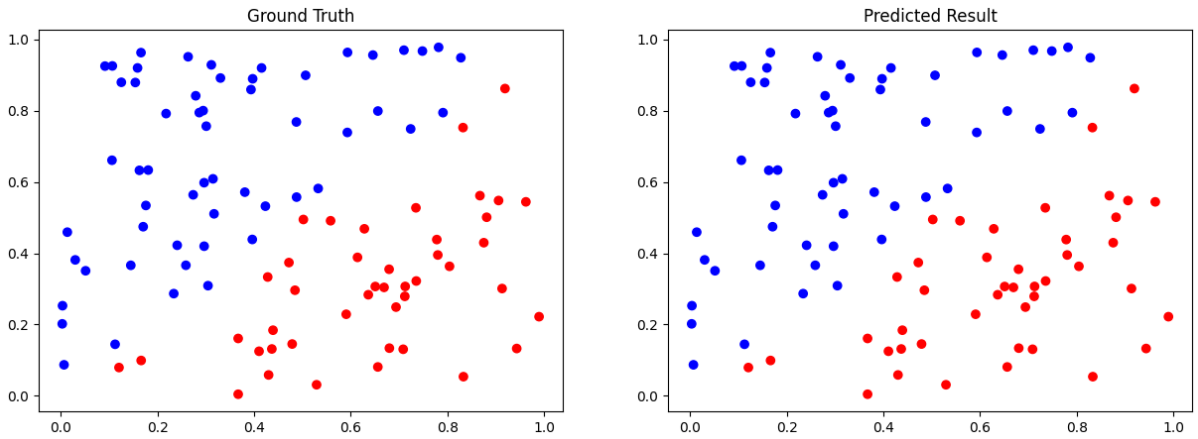

Figure 4. Comparison with ground truth and predicted result for linear data

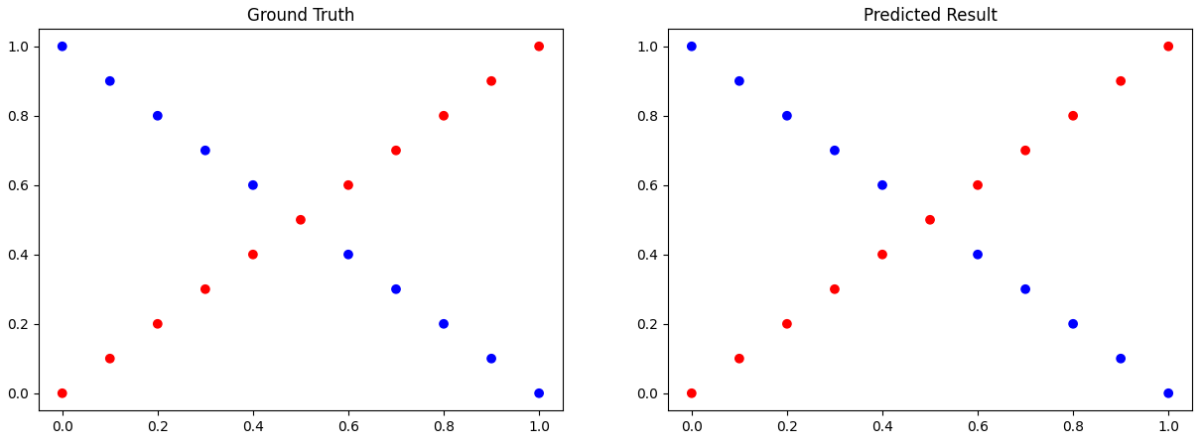

Figure 5. Comparison with ground truth and predicted result for XOR data

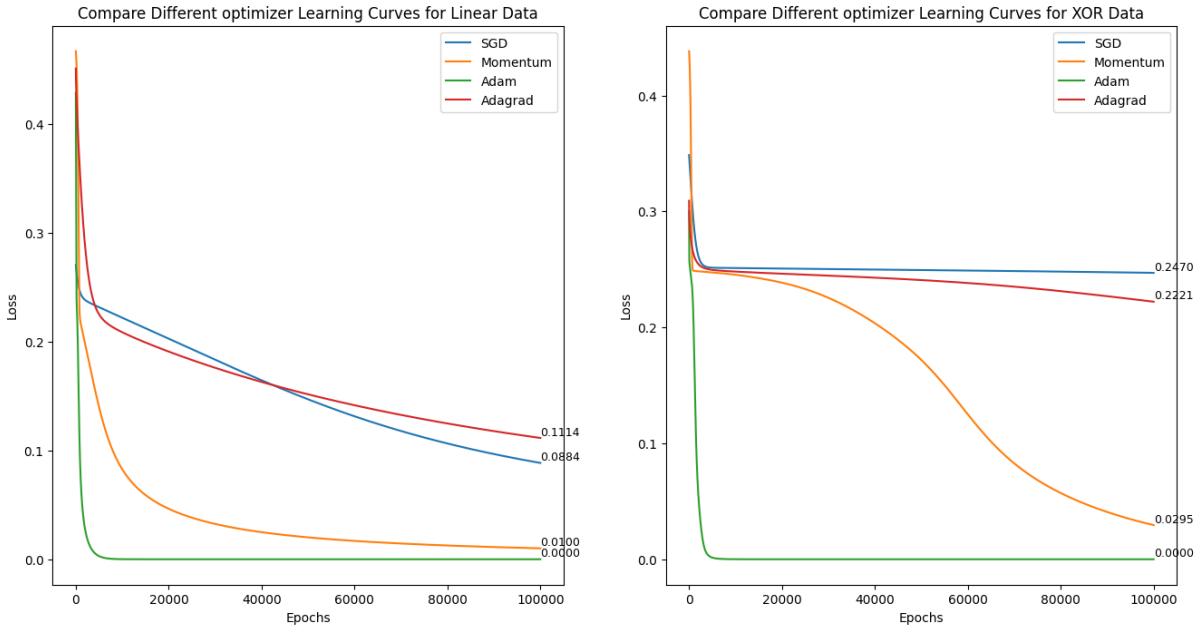

Figure 6. Comparison of different optimizer learning curves on linear data and XOR data.

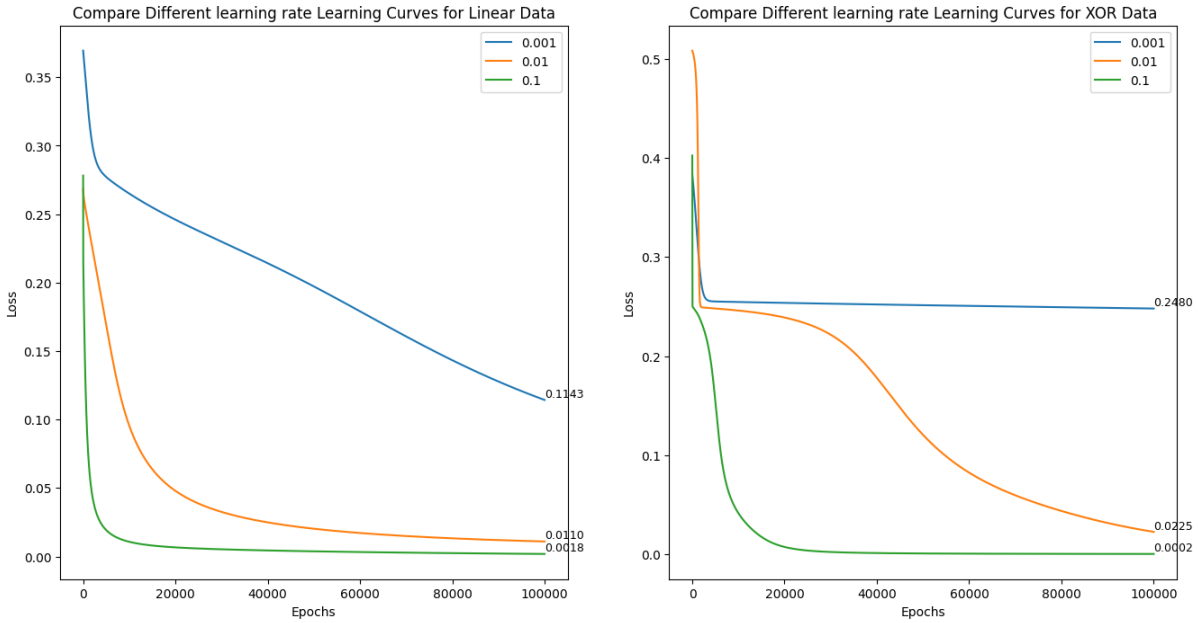

Figure 7. Comparison of different learning rate learning curves on linear data and XOR data.

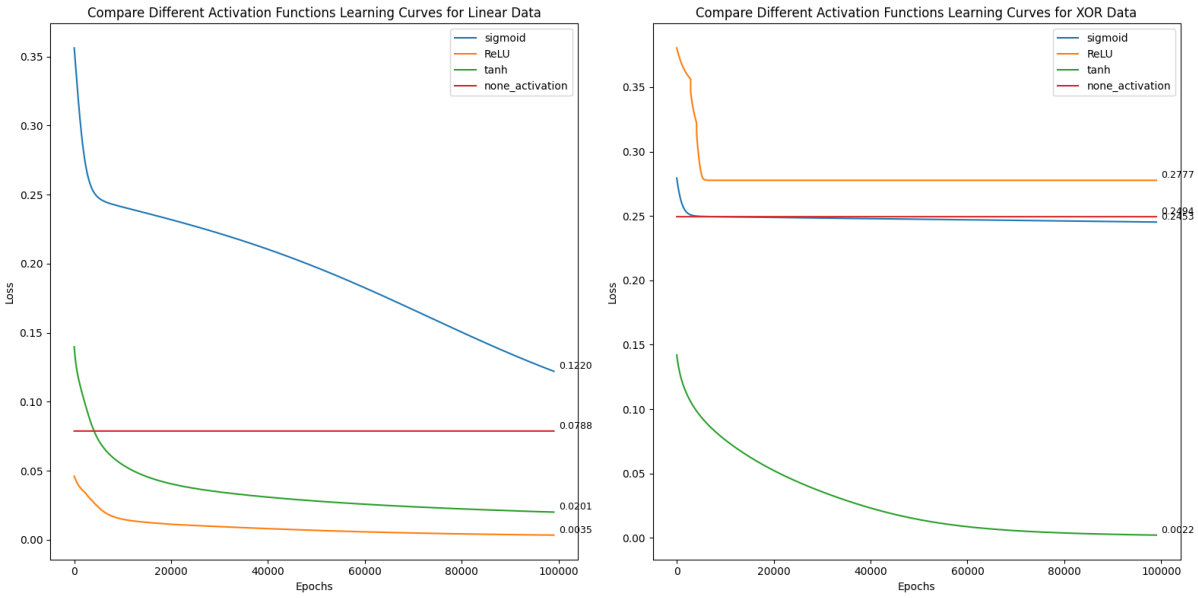

Figure 8. Comparison of learning curves for different activation functions on linear data and XOR data, starting from the 1000th loss.

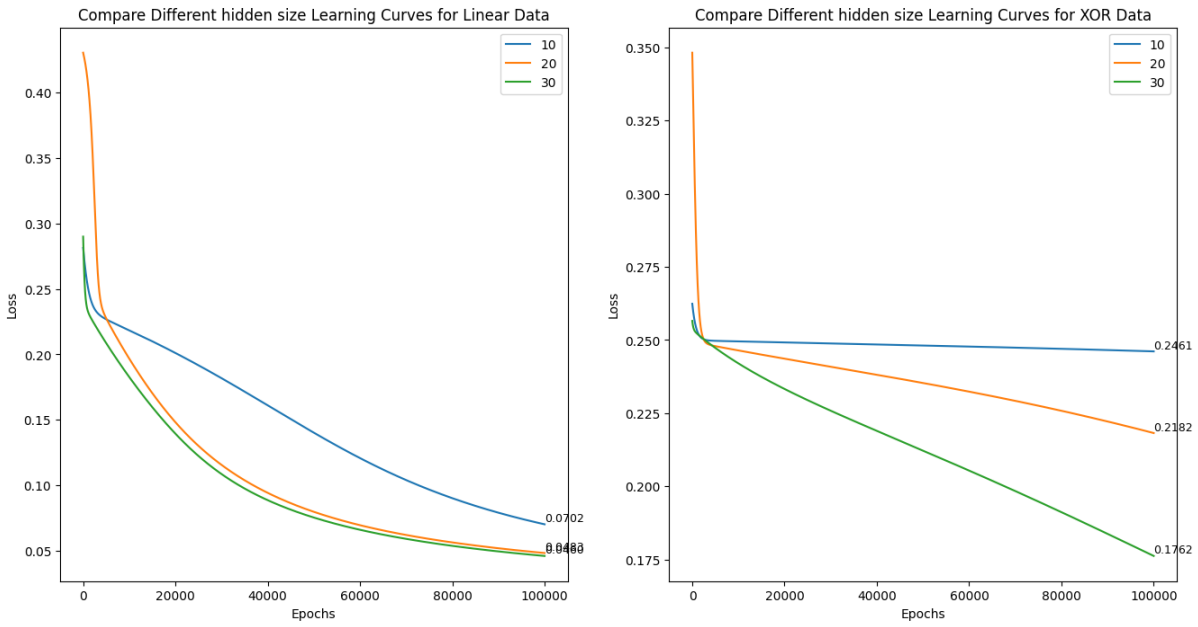

Figure 9. Comparison of different hidden size learning curves on linear data and XOR data.

4. Discussion

A. Try different learning rates

I experimented with various learning rates to observe their impact on the model's convergence speed and final performance. According to figure 7, it is observed that different learning rates significantly affected the model's performance. The learning rate of 0.1 got the lowest loss, indicating it converged effectively to a good solution.

In contrast, the learning rate of 0.001 resulted in the highest loss, showing slower convergence speed. Higher learning rates, such as 0.1, exhibited faster convergence speeds, enabling the model to reach a good solution more quickly.

The lowest learning rate 0.01 was getting stuck easily in local minima, which impeded its ability to find the global optimal solution.

B. Try different number of hidden units

During the experiment, we tested different numbers of hidden units (10, 20, and 30) to evaluate their impact on the model's performance. Our hypothesis was that using a higher number of hidden units would lead to better performance.

According to Figure 9, the results aligned with our expectations, showing that the model with 30 hidden units achieved the best performance. This result is likely because more hidden units provide the network with greater capacity to capture complex patterns and relationships in the data, enabling it to learn more effectively and generalize better to unseen data.

C. Try without activation functions

Function	Hidden units	Loss	Accuracy
generate_linear	10	5.074e-12	1.0
generate_linear	20	3.991e-12	1.0
generate_linear	30	5.642e-12	1.0
generate_XOR_easy	10	4.456e-12	1.0
generate_XOR_easy	20	5.703e-12	1.0
generate_XOR_easy	30	4.484e-12	1.0

Table 2. the results of using a sigmoid activation function with Adam optimizer

Function	Hidden units	Loss	Accuracy
generate_linear	10	0.08289	0.99
generate_linear	20	0.08289	0.99
generate_linear	30	0.08289	0.99
generate_XOR_easy	10	0.2494	0.5238
generate_XOR_easy	20	0.2494	0.5238
generate_XOR_easy	30	0.2494	0.5238

Table 3. the results of without using activation function with Adam optimizer

The experiment results clearly demonstrate the importance of using activation functions in neural networks. In the case of using the Sigmoid activation function with the Adam optimizer, the model achieved perfect accuracy (1.0) and extremely low loss values on both linear and XOR datasets, indicating its ability to capture complex patterns effectively.

Conversely, when the model was tested without activation functions, it showed a high accuracy of only 0.99 on linear data but a significant loss of 0.2494 and an accuracy of just 0.5238 on the XOR dataset. This highlights that the model struggled to learn non-linear relationships without activation functions.

Overall, these findings underscore that activation functions are essential for enabling neural networks to learn and generalize from complex data.

5. Extra

A. Implement different optimizers

The following optimizers will be implemented and I implement the experiment that was comparing the learning curves for linear data and xor data. The result was showed in figure 7.

1. **Adam (Adaptive Moment Estimation)**

Adam combines the advantages of both Momentum and Adagrad, allowing it to adaptively adjust the learning rate for each parameter based on the first and second moments of the gradients. This adaptability leads to faster convergence and better handling of varying data distributions, making it highly effective for training deep networks.

2. **Momentum**
   Momentum improves upon standard SGD by accumulating gradients over time, which helps to accelerate updates in the right direction and dampens oscillations. This leads to faster convergence compared to vanilla SGD, especially in ravines and steep areas of the loss landscape, making it more effective than SGD.

3. **Adagrad**
   Adagrad adapts the learning rate for each parameter individually based on the historical gradients, which can be beneficial for sparse data. However, its aggressive learning rate decay can lead to slow convergence in later stages of training. As a result, while it performs better than SGD, it is less effective than Momentum and Adam for deep networks.

4. **SGD (Stochastic Gradient Descent)**
   SGD is the simplest optimizer, relying solely on the gradient of the loss function. While it can perform well in some scenarios, it often suffers from slow convergence and is prone to oscillations, especially in complex loss landscapes. This makes it the least effective among the tested optimizers in this experiment.

B. Implement different activation functions

According to figure 8, the experiment was comparing different activation functions.

Performance on Linear Data

1. **ReLU**
   - ReLU performs the best on linear data because it effectively captures linear relationships without introducing non-linearity. Its simplicity allows the model to learn efficiently, leading to fast convergence and low loss.
2. **Tanh**
   - Tanh offers better performance than Sigmoid due to its output range of -1 to 1, which can help with normalization. However, it still introduces some non-linearity that may not be necessary for purely linear data.
3. **Sigmoid**
   - Sigmoid struggles the most on linear data due to its output range of 0 to 1, which can limit the model's ability to capture the full range of linear relationships. Additionally, it is more prone to the vanishing gradient problem.

Performance on XOR Data

1. **Tanh**
    - Tanh performs best on XOR data because it effectively captures the non-linear relationships inherent in the dataset. Its symmetric output helps the model better understand the underlying structure of the data.
2. **Sigmoid**
    - Sigmoid ranks second for XOR data as it can still capture non-linear patterns, albeit less effectively than Tanh. It provides a non-linear transformation but is more limited in range.
3. **ReLU**
    - ReLU performs the worst on XOR data because it is not designed to handle non-linear relationships effectively. It can lead to dead neurons, especially in cases where input values fall below zero, which is common in XOR data.

6. Conclusion

This lab explored the implementation of a Multi-Layer Perceptron (MLP) for binary classification, focusing on key factors such as learning rates, hidden unit sizes, activation functions, and optimizers. We found that activation functions play a critical role in model performance, with Sigmoid achieving perfect accuracy on linear data and Tanh performing best on the XOR dataset. Higher learning rates were observed to accelerate convergence, while increasing the number of hidden units improved the model's ability to capture complex patterns. Adam emerged as the most effective optimizer across experiments, underscoring its suitability for training deep neural networks. These findings underscore the importance of thoughtful configuration in achieving optimal neural network performance.