Deep Learning Lab1 Report

陳以瑄 (ID: 313551047)

1 Introduction

This report presents the implementation and evaluation of a simple neural network for binary classification on two datasets: linear (Case 1) and XOR (Case 2). It explores the effects of learning rates, hidden units, and activation functions, including ReLU and sigmoid, on model performance, emphasizing the importance of data structure and hyperparameter tuning in achieving optimal results.

2 Implementation Details

2.1 Sigmoid function

The sigmoid function is defined as: $\sigma(x) = \frac{1}{1+e^{-x}}$
Its derivative is given by: $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
Proof:

$\sigma(x) = \frac{1}{1+e^{-x}}$	$\sigma'(x) = \frac{0\cdot(1+e^{-x}) - 1\cdot(-e^{-x})}{(1+e^{-x})}$
$* \left(\frac{f}{g}\right)' = \frac{f'g - fg'}{g^2}$	$= \frac{e^{-x}}{(1+e^{-x})^2}$
	$= \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}}$
	$= \sigma(x)\cdot(1-\sigma(x))$

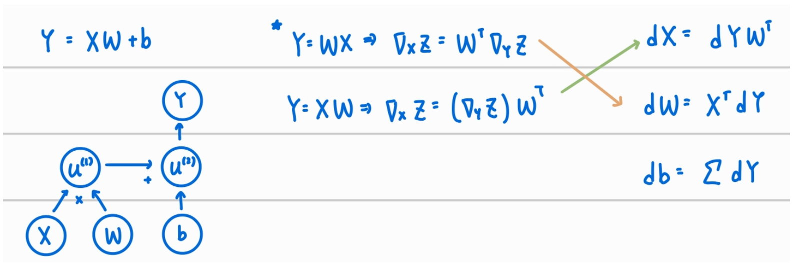
Implementation:

```python
'''Sigmoid:'''
class Sigmoid:
    def __init__(self):
        self.input = None
        self.output = None

    def forward(self, h):
        self.input = h
        self.output = 1.0/(1.0 + np.exp(-self.input))
        return self.output

    def backward(self, dY):
        dX = dY* self.output* (1 - self.output)
        return dX
```

2.2 Neural network architecture

The hidden layer I use is a linear layer.
It is defined as: $y = W^T X + b$, where X is the input vector, W is the weight matrix,and b is the bias vector.

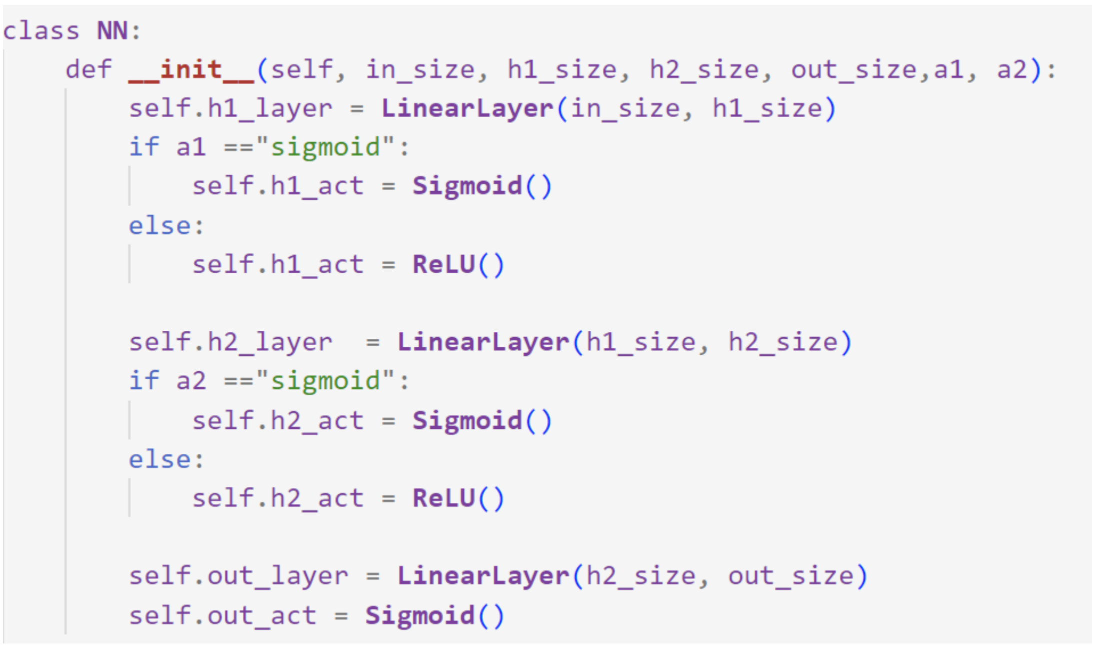
The gradients are computed as follows:
$dX = dYW^T$,
$dW = X^T dY$,
$db = \sum dY$
Proof:



Implementation:

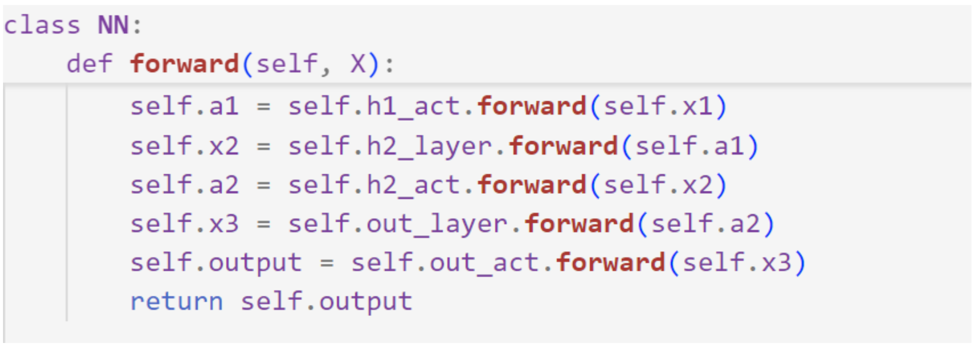
```
'''Linear Layer: y = W^T h + b'''
class LinearLayer:
    def __init__(self, in_size, out_size):
        self.weights = np.random.randn(in_size, out_size)
        self.bias = np.zeros((1,out_size))
        self.input = None
        self.dW = None
        self.db = None

    def forward(self, h):
        self.input = h
        return np.dot(self.input, self.weights) + self.bias

    def backward(self, dY, lr = 0.01):
        dX = np.dot(dY, self.weights.T)
        self.dW = np.dot(self.input.T, dY)
        self.db = np.sum(dY, axis = 0, keepdims = True)
        self.weights -= lr* self.dW
        self.bias -= lr* self.db
        return dX
```

As for the network architecture, I implemented a simple neural network with two hidden layers. The initialization is as follows:



```
class NN:
    def __init__(self, in_size, h1_size, h2_size, out_size,a1, a2):
        self.h1_layer = LinearLayer(in_size, h1_size)
        if a1 =="sigmoid":
            self.h1_act = Sigmoid()
        else:
            self.h1_act = ReLU()

        self.h2_layer  = LinearLayer(h1_size, h2_size)
        if a2 =="sigmoid":
            self.h2_act = Sigmoid()
        else:
            self.h2_act = ReLU()

        self.out_layer = LinearLayer(h2_size, out_size)
        self.out_act = Sigmoid()
```

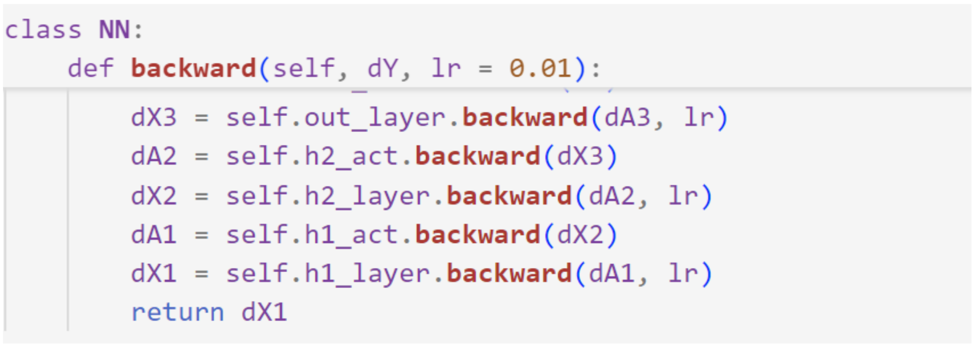
The forward pass is as follows:



```
class NN:
    def forward(self, X):
        self.a1 = self.h1_act.forward(self.x1)
        self.x2 = self.h2_layer.forward(self.a1)
        self.a2 = self.h2_act.forward(self.x2)
        self.x3 = self.out_layer.forward(self.a2)
        self.output = self.out_act.forward(self.x3)
        return self.output
```

2.3 Backpropagation

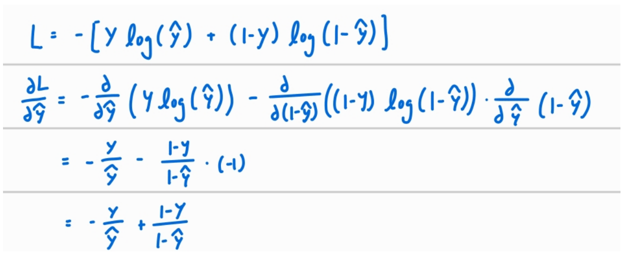
The backpropagation for the whole neuron network is as follows:



```
class NN:
    def backward(self, dY, lr = 0.01):
        dX3 = self.out_layer.backward(dA3, lr)
        dA2 = self.h2_act.backward(dX3)
        dX2 = self.h2_layer.backward(dA2, lr)
        dA1 = self.h1_act.backward(dX2)
        dX1 = self.h1_layer.backward(dA1, lr)
        return dX1
```

The first gradient is from the loss function. Since it is a binary classification problem, I chose cross-entropy as the loss function, which is defined as: $-\left[y\log(\hat{y}) + (1-y)\log(1-\hat{y})\right]$

and its gradient is: $\frac{dL}{d\hat{y}} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}$
Proof:



$$L = -[y \log(\hat{y}) + (1-y) \log(1-\hat{y})]$$

$$\frac{\partial L}{\partial \hat{y}} = -\frac{\partial}{\partial \hat{y}}(y \log(\hat{y})) - \frac{\partial}{\partial(1-\hat{y})}((1-y) \log(1-\hat{y})) \cdot \frac{\partial}{\partial \hat{y}}(1-\hat{y})$$

$$= -\frac{y}{\hat{y}} - \frac{1-y}{1-\hat{y}} \cdot (-1)$$

$$= -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}$$

Implementation:

```python
def cross_entropy_loss(y, y_pred):
    loss = -np.mean(y*np.log(y_pred) + (1-y)*np.log(1-y_pred))
    return loss

def d_cross_entropy(y, y_pred):
    dL = (-y/y_pred) + ((1-y)/(1-y_pred))
    return dL
```

3 Experimental Results

For the following results in this section, I set the parameters as follows:
For Case 1 (the linear dataset):

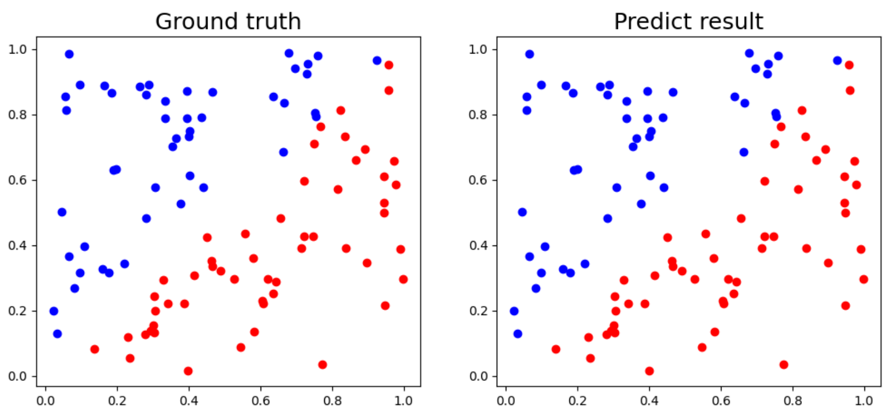
- Learning rate = 0.01
- First hidden layer units = 5
- Second hidden layer units = 3
- Total epochs = 500

For Case 2 (the XOR dataset):

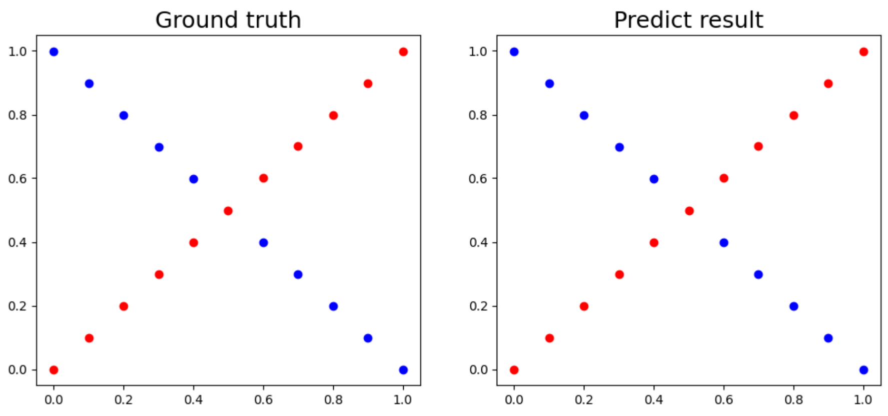
- Learning rate = 0.1
- First hidden layer units = 5
- Second hidden layer units = 3
- Total epochs = 1000

3.1 Screenshot and comparison figure

Case1:

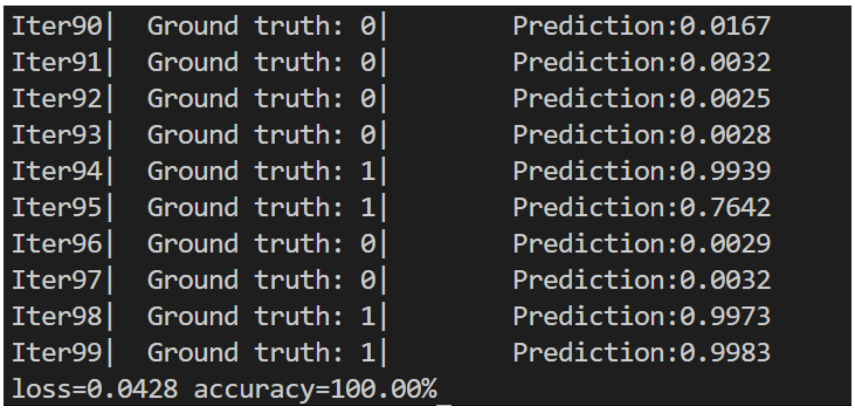


Case2:



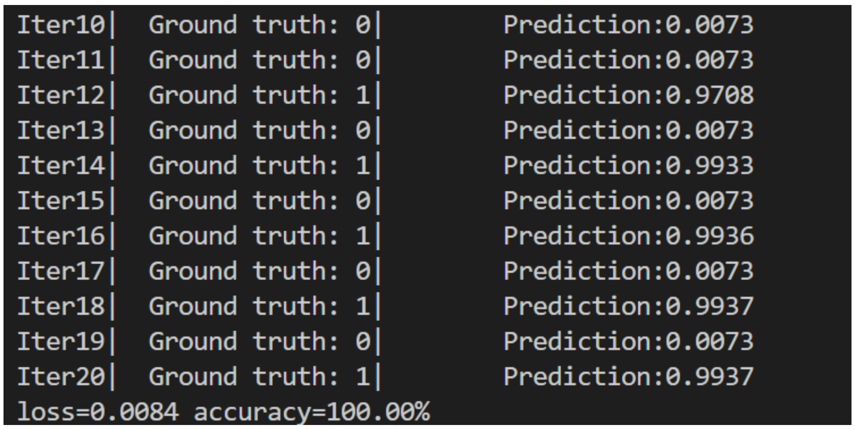
3.2 Show the accuracy of your prediction

Case1:



```
Iter90|   Ground truth: 0|         Prediction:0.0167
Iter91|   Ground truth: 0|         Prediction:0.0032
Iter92|   Ground truth: 0|         Prediction:0.0025
Iter93|   Ground truth: 0|         Prediction:0.0028
Iter94|   Ground truth: 1|         Prediction:0.9939
Iter95|   Ground truth: 1|         Prediction:0.7642
Iter96|   Ground truth: 0|         Prediction:0.0029
Iter97|   Ground truth: 0|         Prediction:0.0032
Iter98|   Ground truth: 1|         Prediction:0.9973
Iter99|   Ground truth: 1|         Prediction:0.9983
loss=0.0428 accuracy=100.00%
```

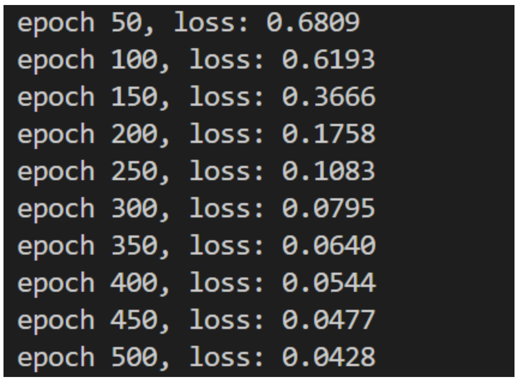
Case2:



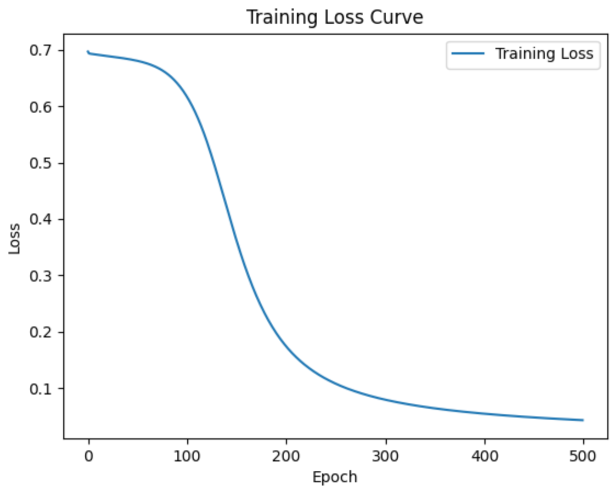
```
Iter10|  Ground truth: 0|        Prediction:0.0073
Iter11|  Ground truth: 0|        Prediction:0.0073
Iter12|  Ground truth: 1|        Prediction:0.9708
Iter13|  Ground truth: 0|        Prediction:0.0073
Iter14|  Ground truth: 1|        Prediction:0.9933
Iter15|  Ground truth: 0|        Prediction:0.0073
Iter16|  Ground truth: 1|        Prediction:0.9936
Iter17|  Ground truth: 0|        Prediction:0.0073
Iter18|  Ground truth: 1|        Prediction:0.9937
Iter19|  Ground truth: 0|        Prediction:0.0073
Iter20|  Ground truth: 1|        Prediction:0.9937
loss=0.0084 accuracy=100.00%
```

3.3 Learning curve

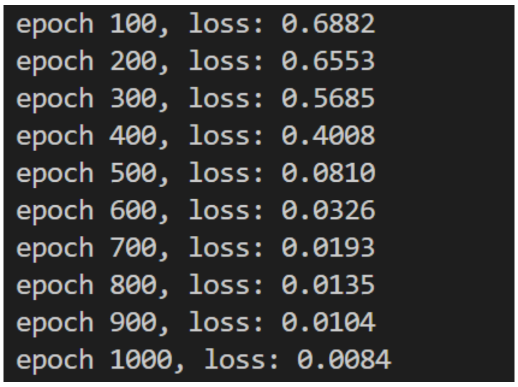
Case1:



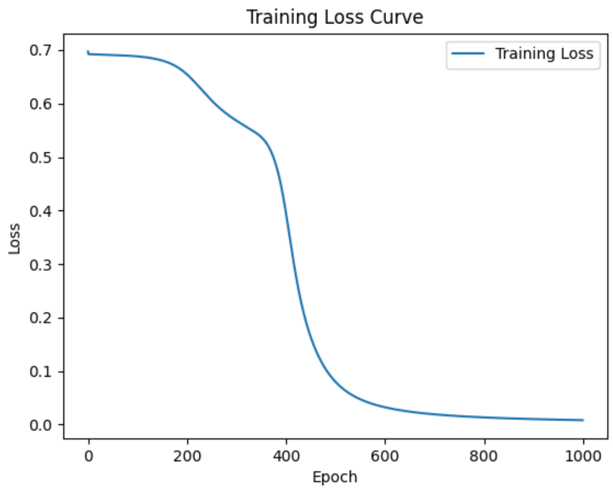
```
epoch 50, loss: 0.6809
epoch 100, loss: 0.6193
epoch 150, loss: 0.3666
epoch 200, loss: 0.1758
epoch 250, loss: 0.1083
epoch 300, loss: 0.0795
epoch 350, loss: 0.0640
epoch 400, loss: 0.0544
epoch 450, loss: 0.0477
epoch 500, loss: 0.0428
```



Case2:



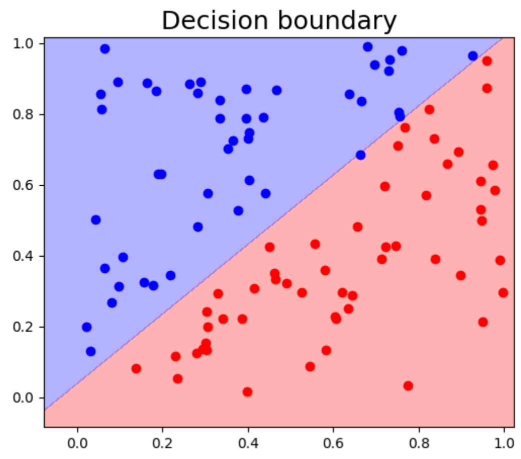
```
epoch 100, loss: 0.6882
epoch 200, loss: 0.6553
epoch 300, loss: 0.5685
epoch 400, loss: 0.4008
epoch 500, loss: 0.0810
epoch 600, loss: 0.0326
epoch 700, loss: 0.0193
epoch 800, loss: 0.0135
epoch 900, loss: 0.0104
epoch 1000, loss: 0.0084
```



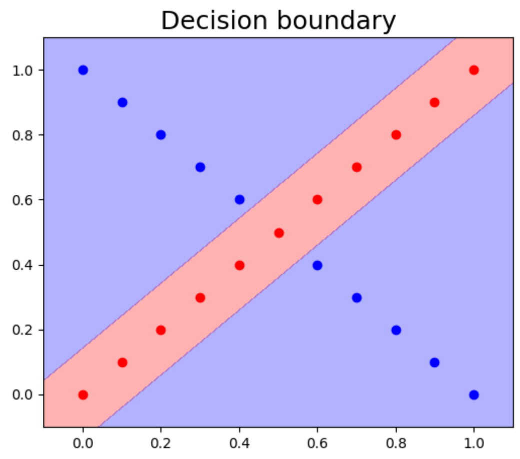
3.4 Anything you want to present

I also plotted the decision boundary. As we can observe in the XOR case, the non-linear transformation introduced by the activation function enables the model to produce a non-linear decision boundary, which is essential for classifying non-linearly separable data.

Case1:



Case2

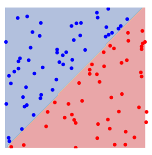
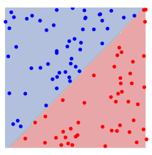
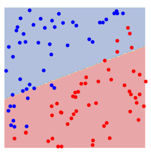
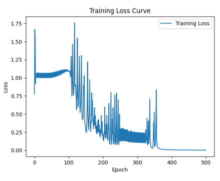
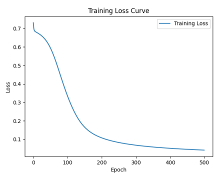
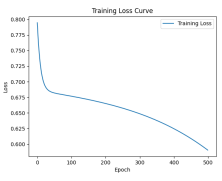


4 Discussion

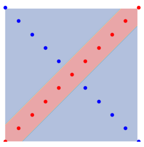
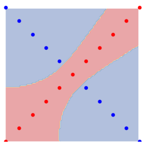
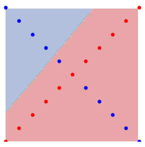
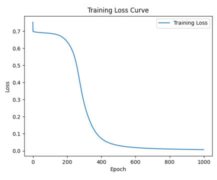
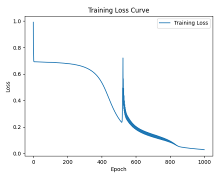
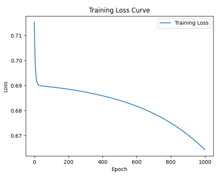
4.1 Try different learning rates

For Case 1 (the linear dataset), I tried three learning rates: 0.1, 0.01, and 0.001. The rest of the parameters are set as described in Section 3.

The dots in the "result" represent the ground truth, while the background color indicates the model decision boundary.

learning rate	0.1	0.01	0.001
accuracy	100%	100%	83.00%
result			
learning curve			

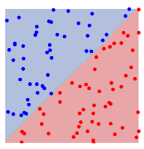
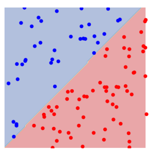
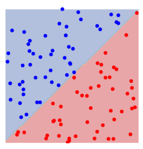
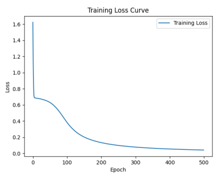
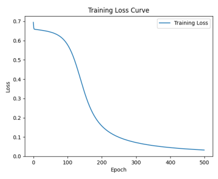
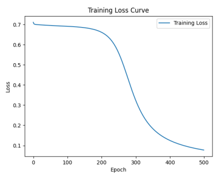
For Case 2 (the XOR dataset), I tried three learning rates: 0.1, 0.05, and 0.01. The rest of the parameters are set as described in Section 3.
The dots in the ”result” represent the ground truth, while the background color indicates the model decision boundary.

learning rate	0.1	0.05	0.01
accuracy	100%	100%	71.43%
result			
learning curve			

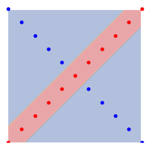
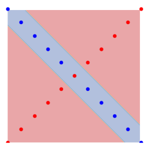
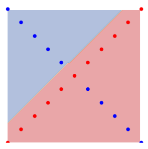
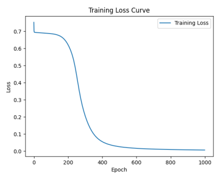
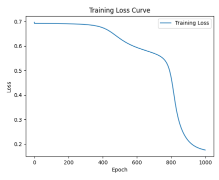
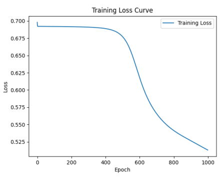
From the above tables, we can see that when the learning rate is too large, such as lr = 0.1 in Case 1, although the model ultimately reaches 100% accuracy, its loss curve is unstable during the training process. This instability might be due to the large learning rate overshooting the optimal solution. On the other hand, when the learning rate is too small, such as lr = 0.001 in Case 1 and lr = 0.01 in Case 2, the training process is very slow, and the model fails to converge within the given number of epochs.

4.2 Try different numbers of hidden units

For both cases, I tried three hidden unit settings: (h1, h2) = (1,1), (3,3), (5,3). The rest of the parameters are set as described in Section 3.
The dots in the ”result” represent the ground truth, while the background color indicates the model’s decision boundary.
Case1:

(h1,h2)	(5,3)	(3,3)	(1,1)
accuracy	100%	100%	100%
result			
learning curve			

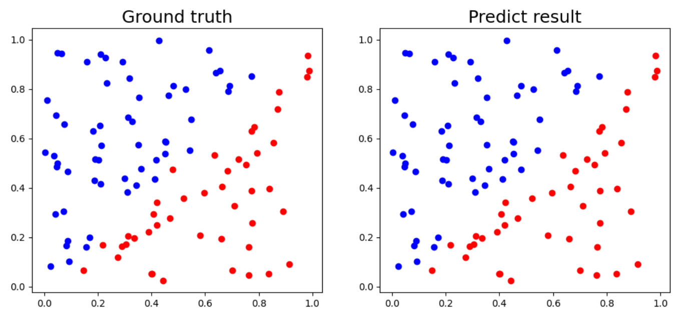
Case2:

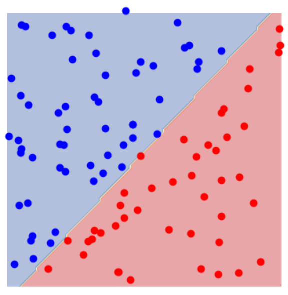
(h1,h2)	(5,3)	(3,3)	(1,1)
accuracy	100%	95.24%	76.19%
result			
learning curve			

From the above tables, we can see that for the linear dataset, even when we use only one hidden unit for each hidden layer, it is still possible to reach 100% accuracy. However, for the XOR dataset, the model fails to separate the classes well under this setting because the dataset is not linearly separable.

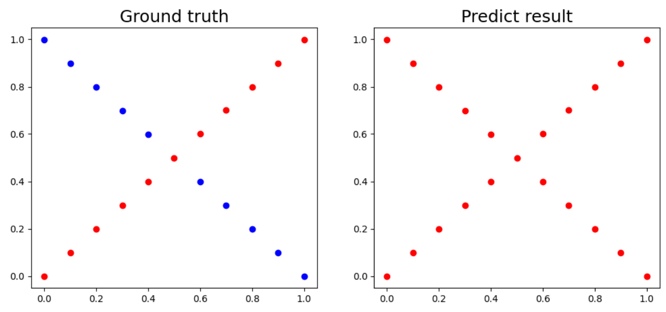
4.3 Try without activation functions

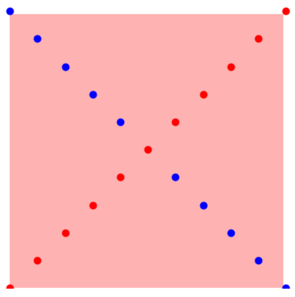
Case1:





Case2:





From the results, we can see that even without the activation function, the model can separate the linear dataset well; however, it fails to separate the XOR dataset due to the lack of the ability to perform non-linear separation.

4.4 Anything you want to share

When training a model, we often tune the hyperparameters to achieve the best performance. However, in this experiment, we can see that the performance of a model depends not only on the hyperparameters but also on the structure of the data. For example, in this section, we discuss the impact of learning rate, the number of hidden units, and the activation function. But in the end, the performance is determined by whether the dataset is linearly separable.

5 Questions

5.1 What is the purpose of activation functions?

As discussed in Section 4.3, the purpose of activation functions is to introduce non-linearity into the network, enabling the model to learn complex patterns. Without activation functions, the network would only perform linear transformations, restricting its ability to capture non-linear relationships in the data.

5.2 What might happen if the learning rate is too large or too small?

As mentioned in Section 4.1, if the learning rate is too large, the model may overshoot the optimal solution, leading to unstable training and causing the loss to fluctuate or diverge. It may also fail to converge to the best solution because the optimization steps are too large. On the other hand, if the learning rate is too small, the model will update the weights very slowly, potentially requiring excessive time to converge.

5.3 What is the purpose of weights and biases in a neural network?

The weights determine the importance of each input in the neural network. During training, the network adjusts the weights to learn the relationship between inputs and outputs, optimizing the network to minimize the loss function. The biases allow the model to shift the decision boundary and adjust the output independently of the inputs.

6 Extra

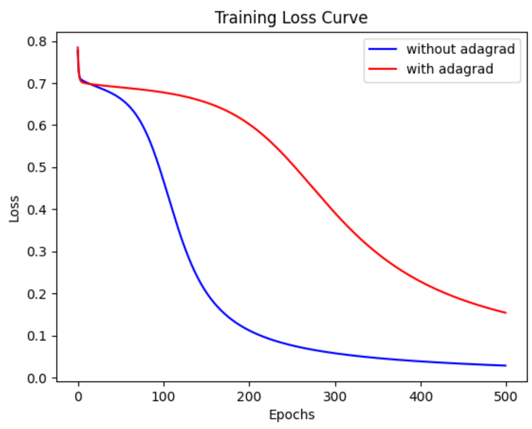
6.1 Implement different optimizers.

I implemented the adaptive gradient optimization algorithm, which adjusts the learning rate for each parameter individually based on historical gradient information.
Implementation:

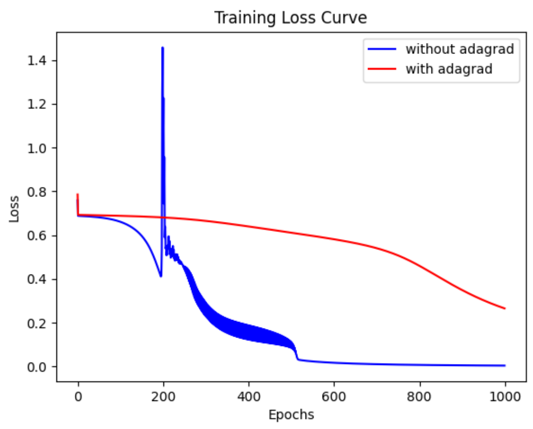
```
class Adagrad():
    def __init__(self, lr=0.01):
        self.lr = lr
        self.G = None
    def getlr(self, dY):
        if self.G is None:
            self.G = 1
        self.G += np.mean(dY**2)
        return self.lr / np.sqrt(self.G + 1e-7)
```

Result:

Case1:



Case2



We can see that, due to the gradually decreasing learning rates over time, applying Adagrad leads to slower convergence, but results in a much smoother loss curve.

6.2 Implement different activation functions.

I also implement the ReLU activation function, which is define as: ReLU(x) = max0,x

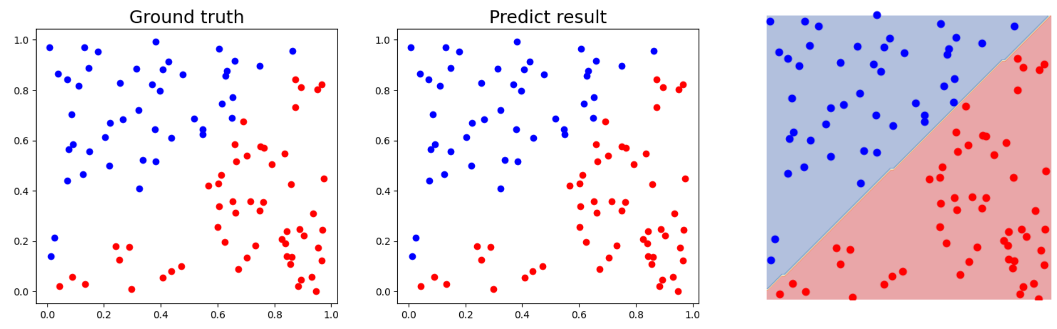
Its derivative is given by: $ReLU'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$

Implementation:

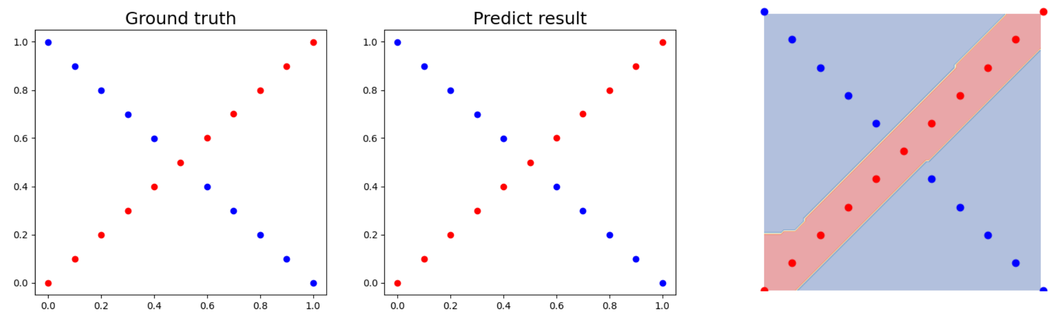
```
'''ReLU'''
class ReLU:
    def __init__(self):
        self.input = None
        self.output = None

    def forward(self, h):
        self.input = h
        self.output = np.maximum(0, self.input)
        return self.output

    def backward(self,dY):
        dX =  dY * np.where(self.output > 0, 1, 0)
        return dX
```

Result:

For Case 1, I set the parameters the same as in Section 3.



For Case 2, I set the parameters the same as in Section 3, except for the learning rate. This time, I set lr = 0.01.



Both cases using ReLU reach 100% accuracy, which is the same as the performance with the sigmoid function. However, there is a slight difference in the decision boundary in Case 2: the boundary has an angle with ReLU, whereas the sigmoid function produces a smoother boundary (see Section 4.1).

6.3 Implement convolution layers.

skip.