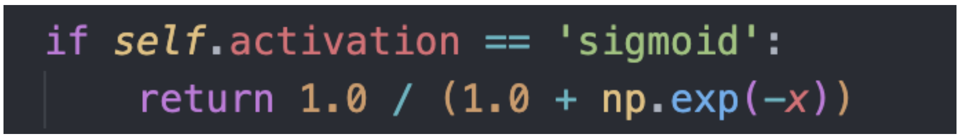
Introduction

In this lab, we need to implement a two-layer fully connected neural network using numpy. First, we will generate input data and labels using the `generateLinear` and `generate_XOR_easy` functions. Then, we will use forward propagation to obtain the predicted labels from the model. We will also calculate the difference between the predicted labels and the ground-truth labels using the MSE Loss. Finally, we will perform backpropagation to calculate the gradients and update the weights of each layer.

Experiment setups

Sigmoid Functions

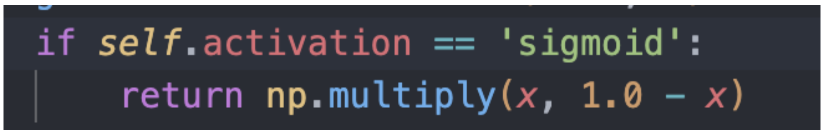
The pictures below illustrates the calculation of the sigmoid function and the implementation of the calculation of the sigmoid function.



```python
if self.activation == 'sigmoid':
    return 1.0 / (1.0 + np.exp(-x))
```

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

The pictures below illustrates the calculation of the derivative of the sigmoid function and the implementation of the calculation of the derivative of the sigmoid function.



```python
if self.activation == 'sigmoid':
    return np.multiply(x, 1.0 - x)
```

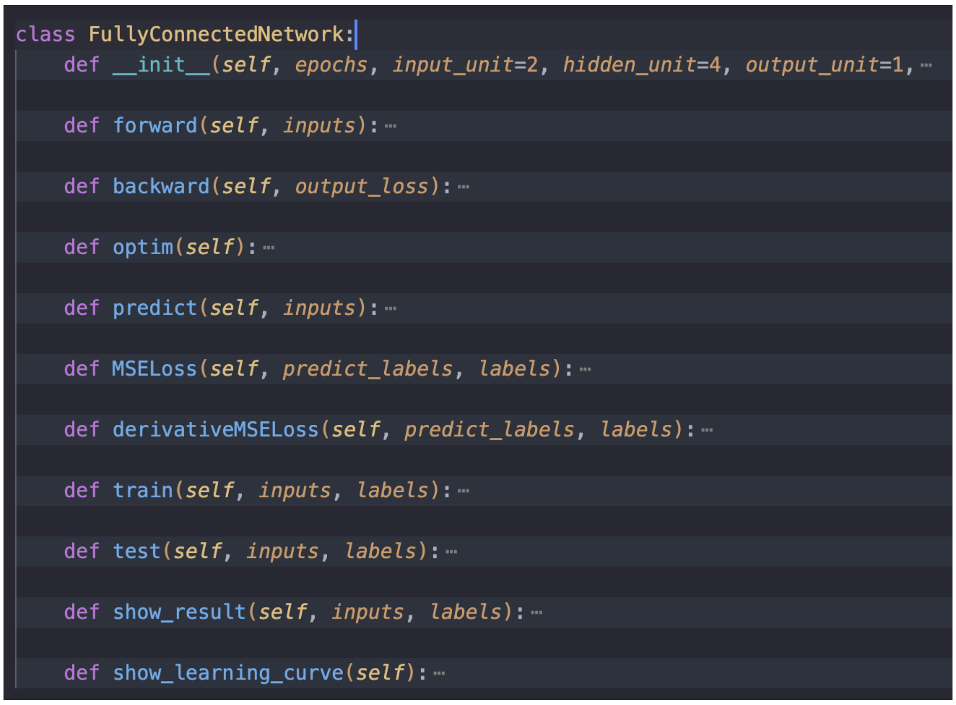
$$\frac{d\sigma(x)}{d(x)} = \sigma(x) \cdot (1 - \sigma(x)).$$

The calculation of the Sigmoid function is primarily used for forward propagation, while the derivative of the Sigmoid function is mainly used for backpropagation.

Neural Network

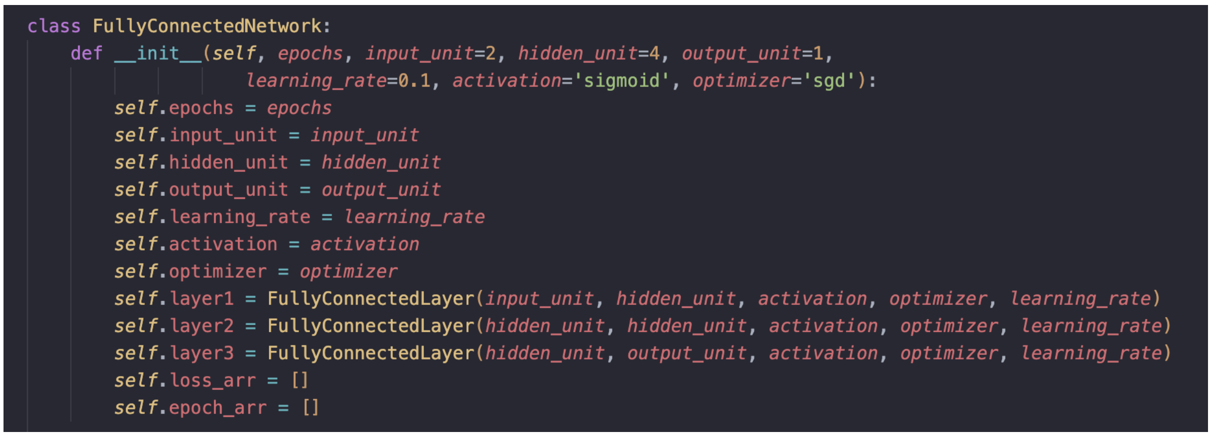
The implementation involves first creating a class for the `FullyConnectedNetwork`. Then, inside the network, there are three instances of `FullyConnectedLayer`.

Fully Connected Network



```python
class FullyConnectedNetwork:
    def __init__(self, epochs, input_unit=2, hidden_unit=4, output_unit=1,⋯

    def forward(self, inputs):⋯

    def backward(self, output_loss):⋯

    def optim(self):⋯

    def predict(self, inputs):⋯

    def MSELoss(self, predict_labels, labels):⋯

    def derivativeMSELoss(self, predict_labels, labels):⋯

    def train(self, inputs, labels):⋯

    def test(self, inputs, labels):⋯

    def show_result(self, inputs, labels):⋯

    def show_learning_curve(self):⋯
```

In the `__init__()` function of the network, some parameters, including activation function and optimizer, are defined here.

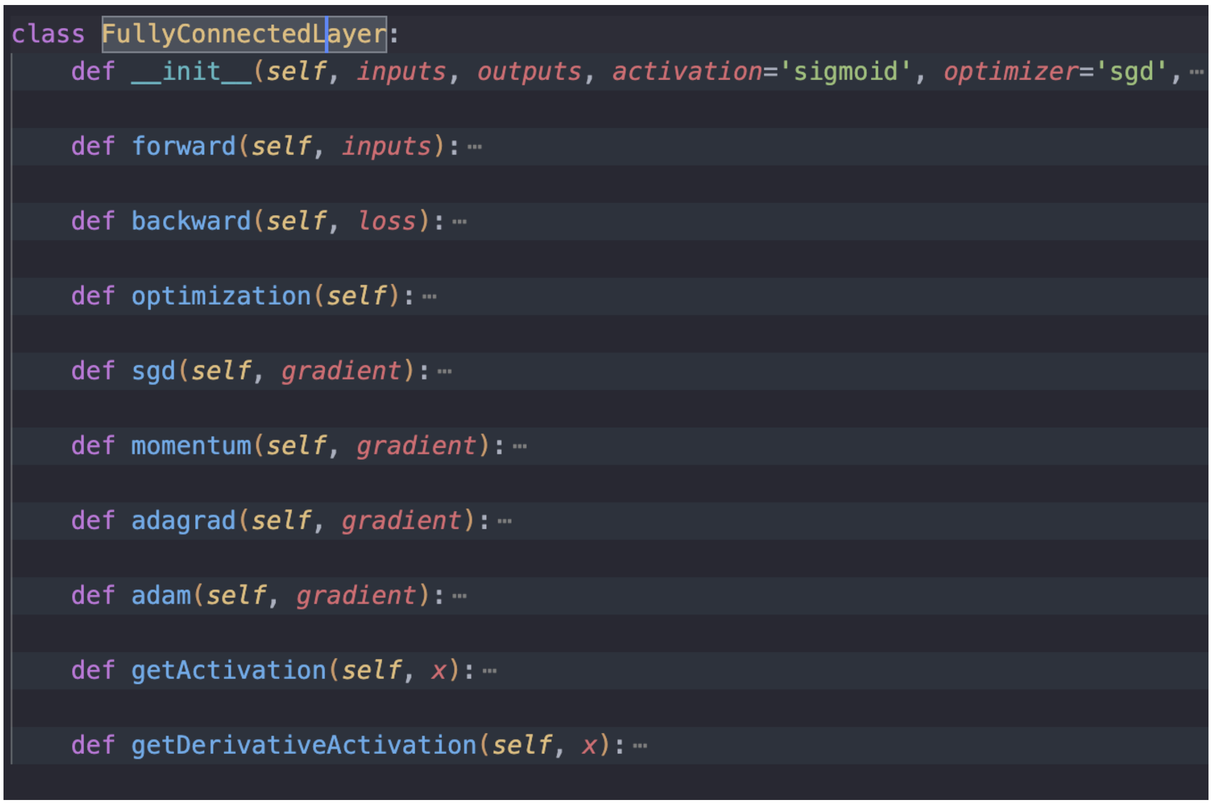


```python
class FullyConnectedNetwork:
    def __init__(self, epochs, input_unit=2, hidden_unit=4, output_unit=1,
                learning_rate=0.1, activation='sigmoid', optimizer='sgd'):
        self.epochs = epochs
        self.input_unit = input_unit
        self.hidden_unit = hidden_unit
        self.output_unit = output_unit
        self.learning_rate = learning_rate
        self.activation = activation
        self.optimizer = optimizer
        self.layer1 = FullyConnectedLayer(input_unit, hidden_unit, activation, optimizer, learning_rate)
        self.layer2 = FullyConnectedLayer(hidden_unit, hidden_unit, activation, optimizer, learning_rate)
        self.layer3 = FullyConnectedLayer(hidden_unit, output_unit, activation, optimizer, learning_rate)
        self.loss_arr = []
        self.epoch_arr = []
```

The `FullyConnectedNetwork` class includes several functions and methods. The meaning of the most important ones is listed below:

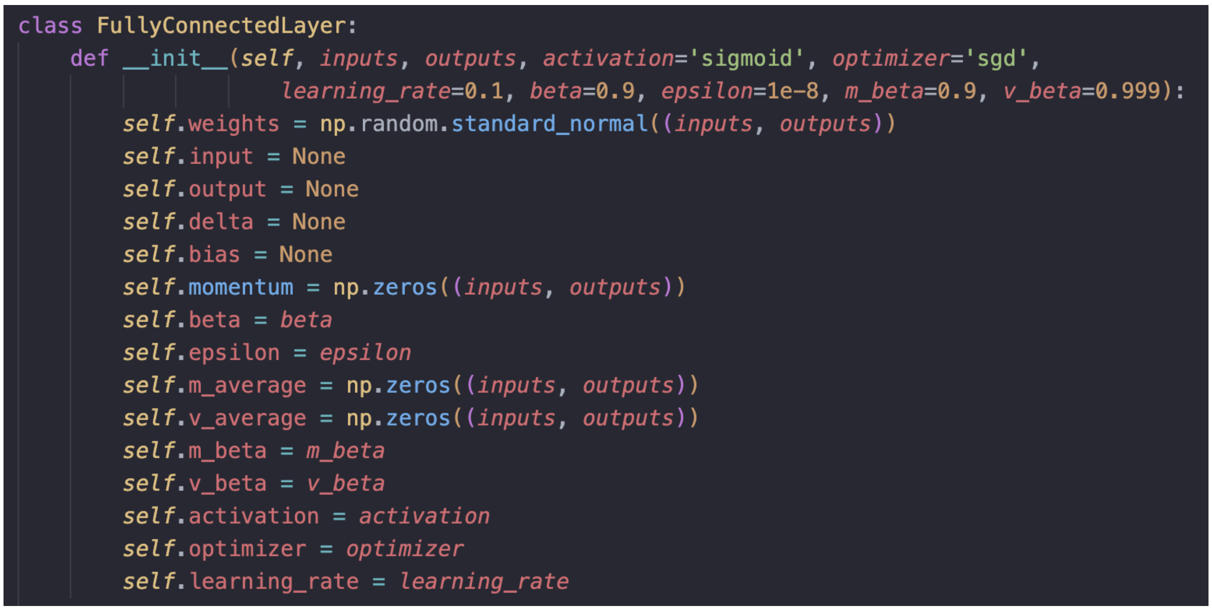
- `forward`: perform forward propagation
- `backward`: perform backpropagation
- `optim`: update the weights of each layer

Fully Connected Layer



```python
class FullyConnectedLayer:
    def __init__(self, inputs, outputs, activation='sigmoid', optimizer='sgd',⋯

    def forward(self, inputs):⋯

    def backward(self, loss):⋯

    def optimization(self):⋯

    def sgd(self, gradient):⋯

    def momentum(self, gradient):⋯

    def adagrad(self, gradient):⋯

    def adam(self, gradient):⋯

    def getActivation(self, x):⋯

    def getDerivativeActivation(self, x):⋯
```

The `__init__` function of the `FullyConnectedLayer` will also define various parameters such as the optimizer, activation function, learning rate, and other optimizer-related parameters like beta and epsilon.



```python
class FullyConnectedLayer:
    def __init__(self, inputs, outputs, activation='sigmoid', optimizer='sgd',
                 learning_rate=0.1, beta=0.9, epsilon=1e-8, m_beta=0.9, v_beta=0.999):
        self.weights = np.random.standard_normal((inputs, outputs))
        self.input = None
        self.output = None
        self.delta = None
        self.bias = None
        self.momentum = np.zeros((inputs, outputs))
        self.beta = beta
        self.epsilon = epsilon
        self.m_average = np.zeros((inputs, outputs))
        self.v_average = np.zeros((inputs, outputs))
        self.m_beta = m_beta
        self.v_beta = v_beta
        self.activation = activation
        self.optimizer = optimizer
        self.learning_rate = learning_rate
```

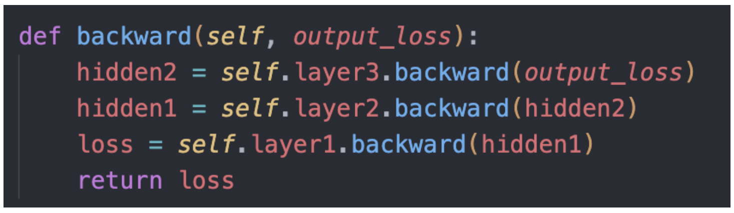
The `FullyConnectedLayer` class also includes several methods:

- `forward`: Performs forward propagation within a layer, passing the output to the next layer as input
- `backward`: Performs backpropagation within a layer, using the loss calculated from the next layer as input to compute the gradient and pass it to the previous layer
- `optimization`: Updates the weights' parameters based on the selected optimizer

- `getActivation`: Computes the activation function's output, mainly used for forward propagation
- `getDerivativeActivation`: Computes the derivative of the activation function, mainly used for backpropagation

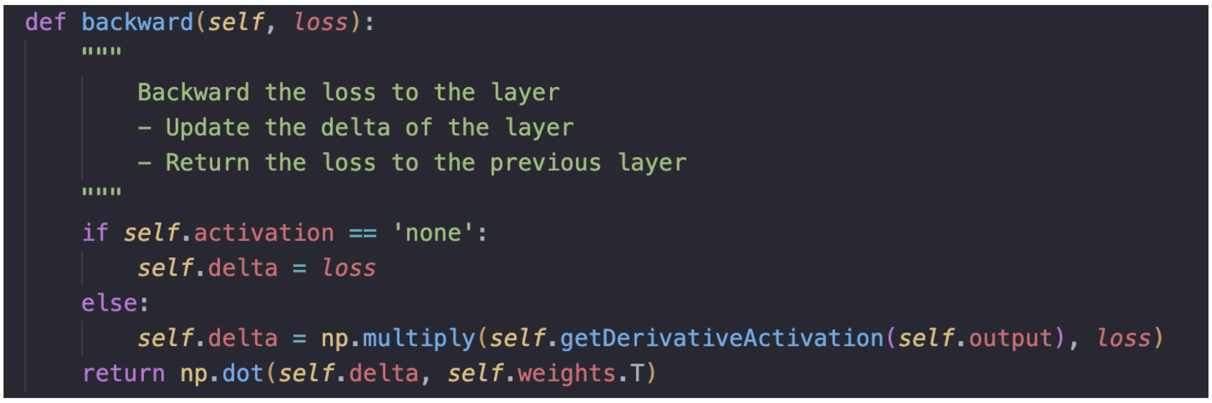
Backpropagation

In the `backward` function of `FullyConnectedNetwork`, the gradient is passed layer by layer from the output layer to the input layer.



```python
def backward(self, output_loss):
    hidden2 = self.layer3.backward(output_loss)
    hidden1 = self.layer2.backward(hidden2)
    loss = self.layer1.backward(hidden1)
    return loss
```

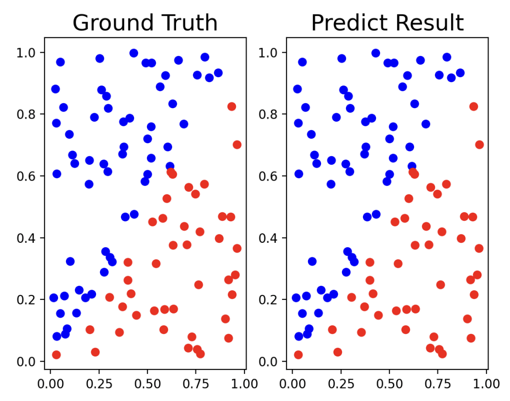
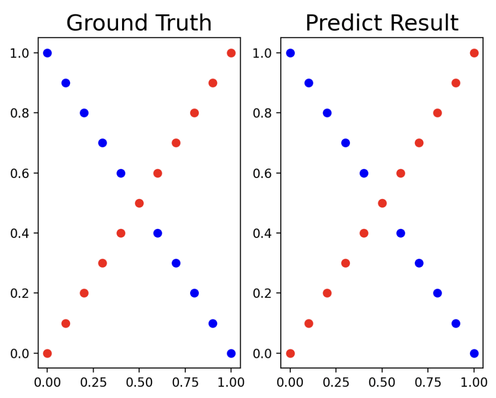
In the `backward` function of `FullyConnectedLayer`, the delta is the derivative of the output multiplied by the loss propagated from the previous layer, and the dot product of delta and the weights gives the loss for the current layer.



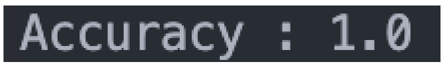
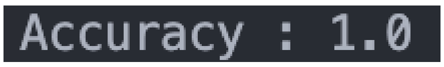
```python
def backward(self, loss):
    """
        Backward the loss to the layer
        - Update the delta of the layer
        - Return the loss to the previous layer
    """
    if self.activation == 'none':
        self.delta = loss
    else:
        self.delta = np.multiply(self.getDerivativeActivation(self.output), loss)
    return np.dot(self.delta, self.weights.T)
```

Results

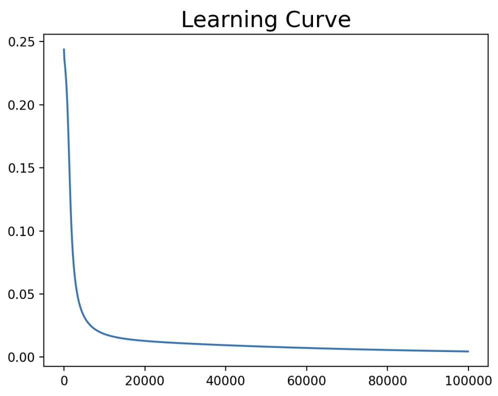
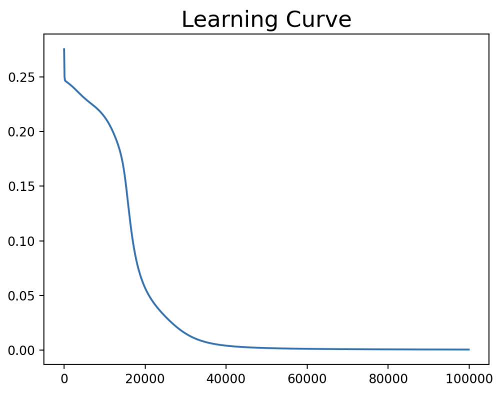
Screenshot and Comparison Figure

Linear	XOR
	

The accuracy of my prediction

Linear	XOR
 `Accuracy : 1.0`	 `Accuracy : 1.0`

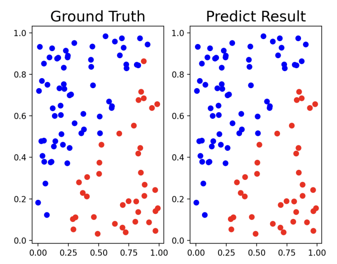
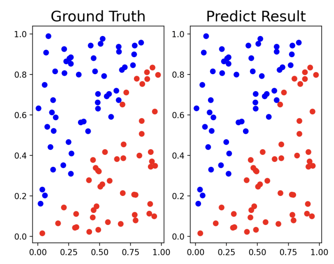
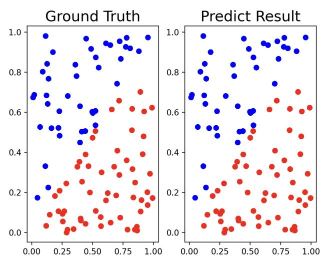
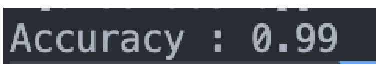
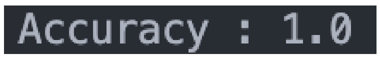
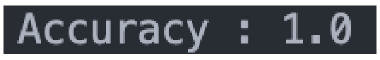
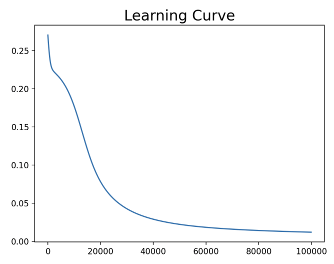
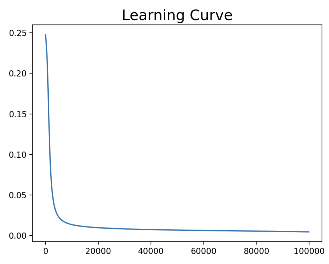
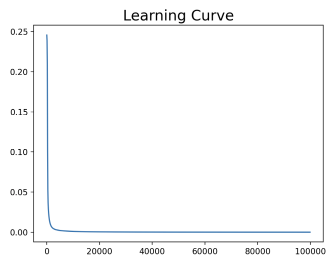
Learning Curve

Linear	XOR
	

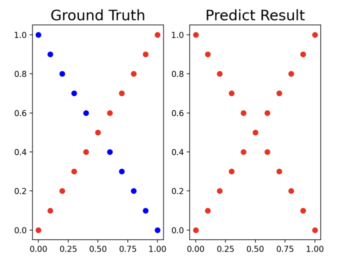
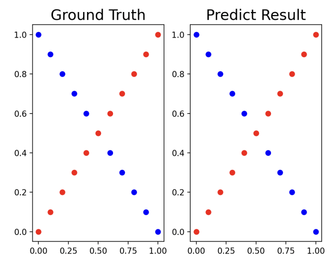
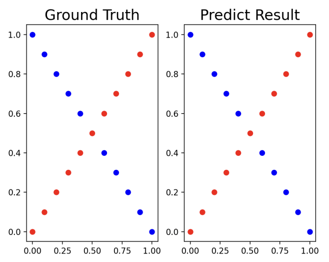
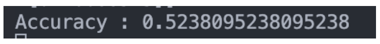
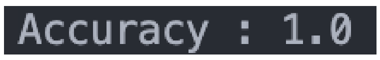
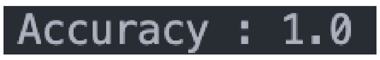
Discussion

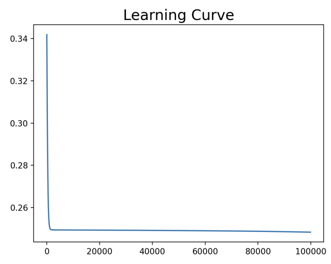
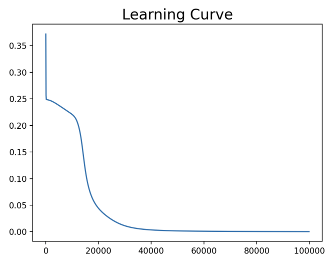
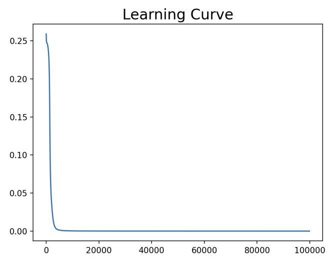
Try different learning rates

Linear data:

Learning Rate = 0.01	Learning Rate = 0.1	Learning Rate = 1
		
 Accuracy : 0.99	 Accuracy : 1.0	 Accuracy : 1.0
		

XOR data:

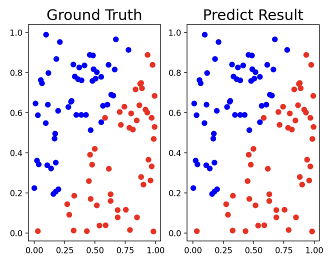
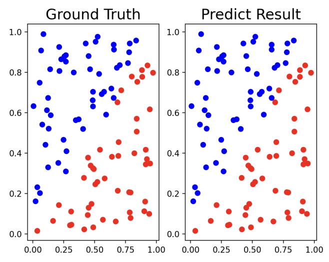
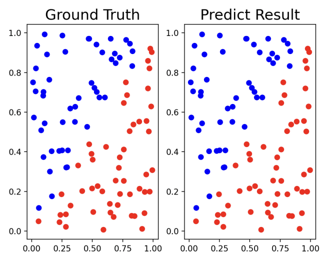
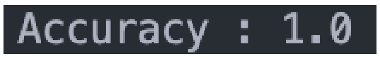
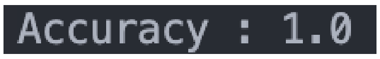
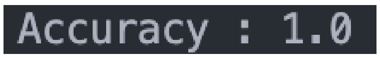
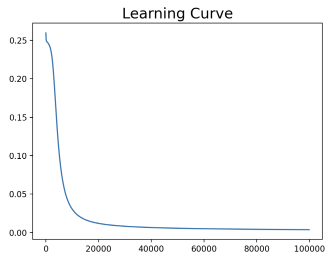
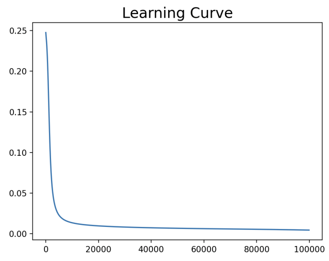
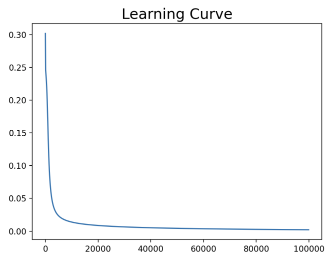
Learning Rate = 0.01	Learning Rate = 0.1	Learning Rate = 1
		
 Accuracy : 0.5238095238095238	 Accuracy : 1.0	 Accuracy : 1.0

From the table, we can see that for Linear Data, a smaller learning rate can result in areas where the loss is slow to decrease. For XOR Data, a smaller learning rate can cause the loss to plateau in a relatively flat region and stop decreasing.

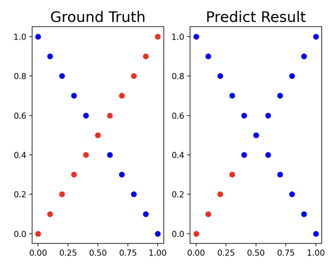
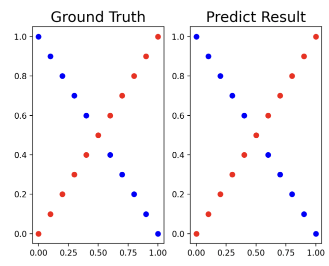
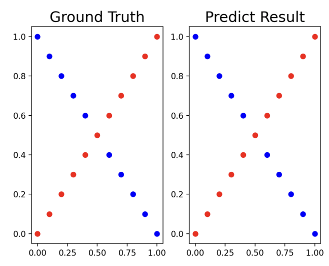
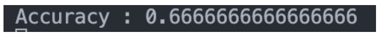
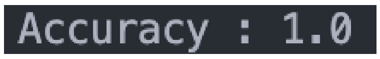
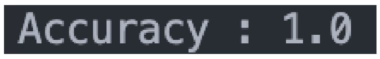
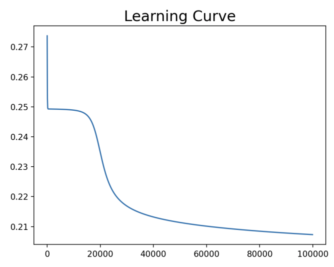
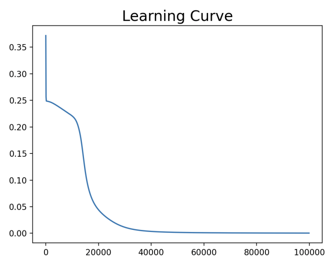
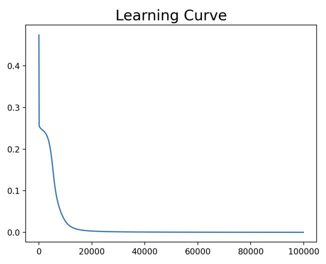
Try different number of hidden units

Linear data:

2 hidden units	4 hidden units	8 hidden units
		
`Accuracy : 1.0`	`Accuracy : 1.0`	`Accuracy : 1.0`
		

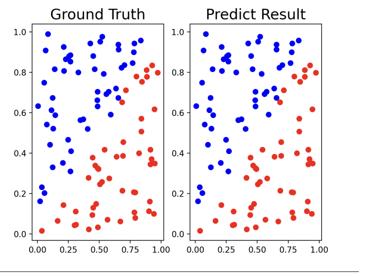
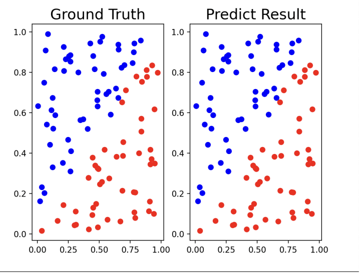
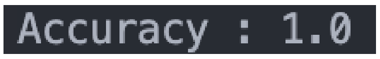
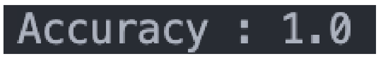
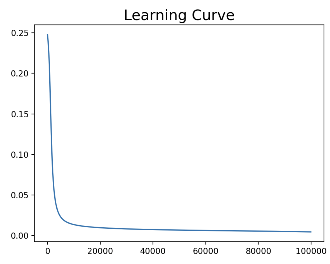
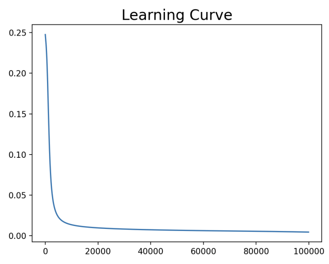
XOR data:

2 hidden units	4 hidden units	8 hidden units

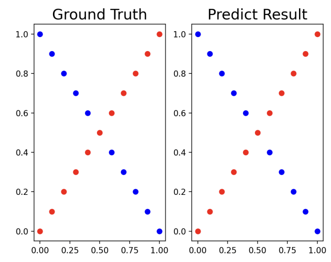
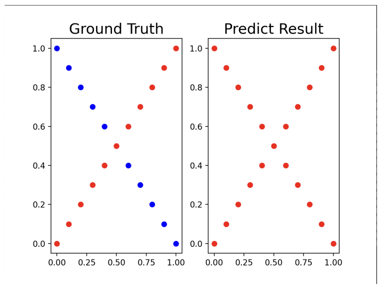
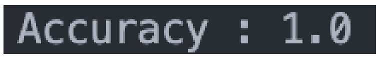
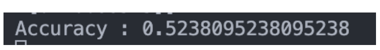
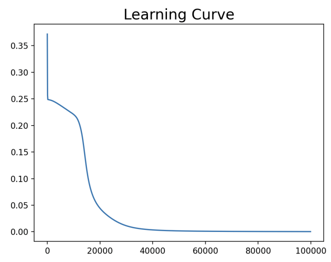
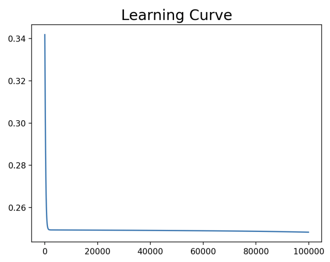
		
 Accuracy : 0.6666666666666666	 Accuracy : 1.0	 Accuracy : 1.0
		

Try without activation functions

Linear data:

With	Without
	
 Accuracy : 1.0	 Accuracy : 1.0
	

XOR data:

With	Without
	
 Accuracy : 1.0	 Accuracy : 0.5238095238095238
	

From the table, we can see that without an activation function, underfitting may occur. When the model is too simple, the loss stops decreasing after a certain point.

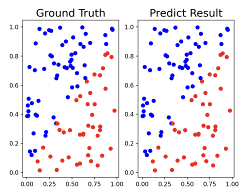
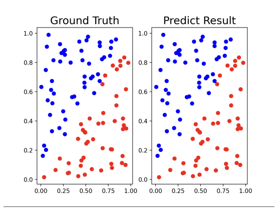
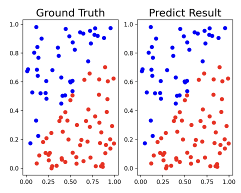
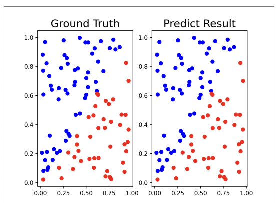
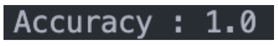
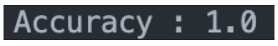
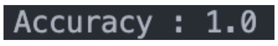
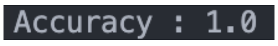
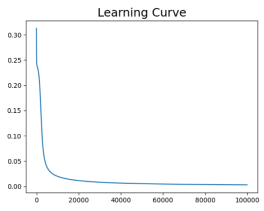
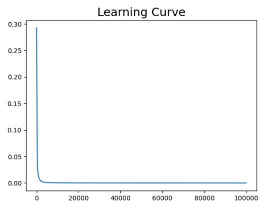
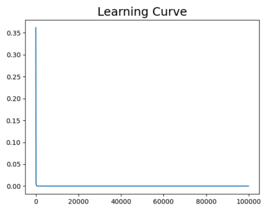
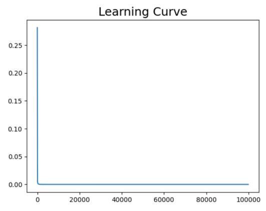
Anything to share

Initially, while implementing this model, it was observed that the loss was not decreasing effectively. After extensive debugging, it was discovered that weight initialization is crucial. Originally, `np.random.uniform` was used, but switching to `np.random.standard_normal` and adding a bias significantly improved the results.

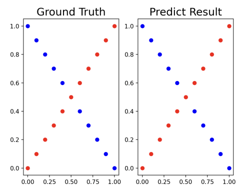
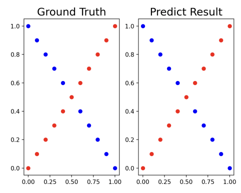
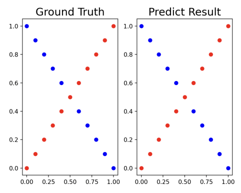
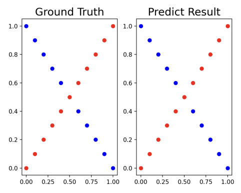
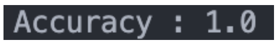
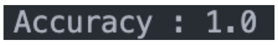
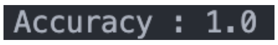
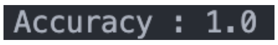
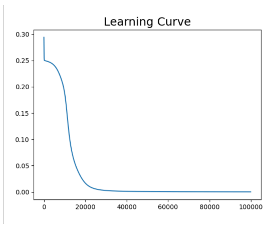
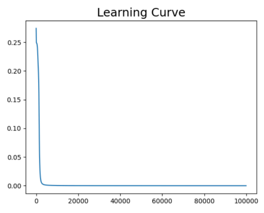
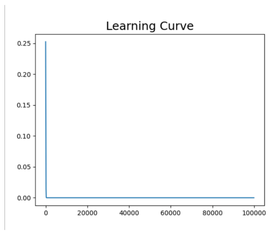
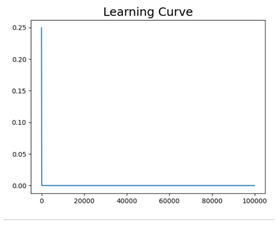
Extra

Implement different optimizers

Linear data:

SGD	Momentum	Adagrad	Adam
			
 Accuracy : 1.0	 Accuracy : 1.0	 Accuracy : 1.0	 Accuracy : 1.0
			

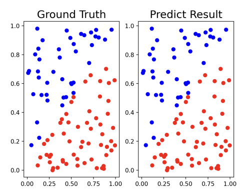
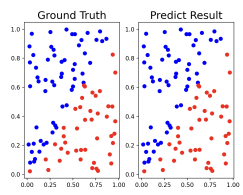
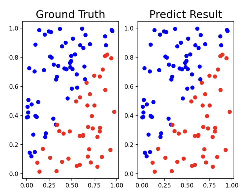
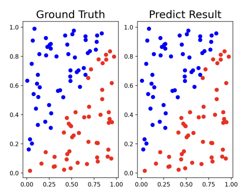
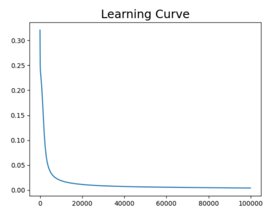
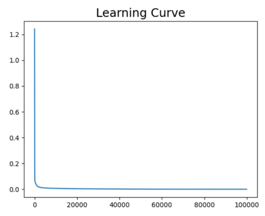
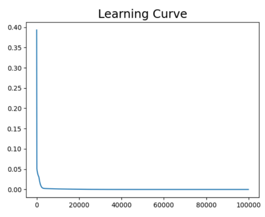
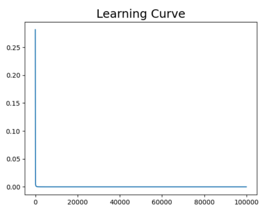
XOR data:

SGD	Momentum	Adagrad	Adam
			
 Accuracy : 1.0	 Accuracy : 1.0	 Accuracy : 1.0	 Accuracy : 1.0
			

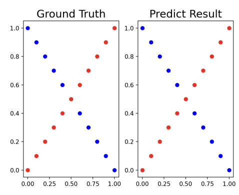
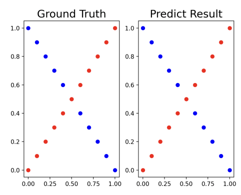
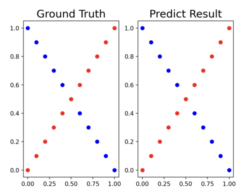
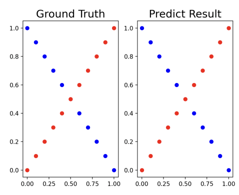
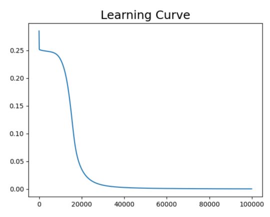
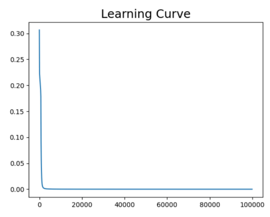
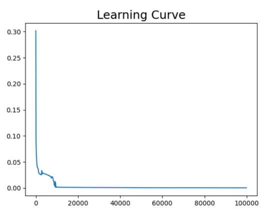
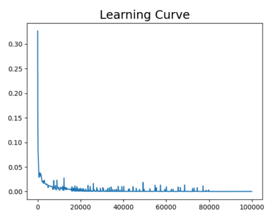
From the table, we can see that the speed of loss reduction is as follows: Adam > Adagrad > Momentum > SGD.

Implement different activation functions

Linear data:

Sigmoid	Tanh	ReLU	Leakly ReLU
			
Accuracy : 1.0	Accuracy : 1.0	Accuracy : 1.0	Accuracy : 1.0
			

XOR data:

Sigmoid	Tanh	ReLU	Leakly ReLU
			
Accuracy : 1.0	Accuracy : 1.0	Accuracy : 1.0	Accuracy : 1.0
			

From the table, we can see that the learning curves for ReLU and Leaky ReLU tend to fluctuate more.