

DLP Lab1 Back-Propagation

Name: 丁祐承

ID: 0812212

1. Introduction

The purpose of this lab is to let us implement the back propagation process of the neural networks. For training a simple neural network, we first need to initiate the weights of the network and then prepare the training dataset. After having all of these, we can start to train the network. We iteratively train the network by forwarding the training data. After forwarding, we compute the loss and do the back propagation. Back propagation is a technique to compute the gradients of the weights by calculation the partial derivative using chain rule. After having the gradients, we optimize the loss function to get the optimal weight for the network. Above all is basically the main idea of this lab.

2. Experiment setups

A. Sigmoid function

```
class sigmoid():
    def __init__(self):
        self.output = None

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

    def backward(self, output_grad):
        return output_grad * self.output * (1 - self.output)
```

I write the sigmoid function as a class. It has two functions: one is the forward and the other one is backward. Forward function uses as the

forward process of the network and the backward function uses as the backward process to calculate the gradient.

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

Equation of Sigmoid function and its derivative

B. Neural network

```
class LinearLayer():
    def __init__(self, in_features, out_features):
        self.weight = np.random.randn(in_features, out_features)
        self.bias = np.zeros((1, out_features))
        # Use when the optimizer is Momentum
        self.vw = np.zeros((in_features, out_features))
        self.vb = np.zeros((1, out_features))

    def forward(self, x):
        self.input = x
        output = np.dot(self.input, self.weight) + self.bias
        return output

    def backward(self, output_grad):
        input_grad = np.dot(output_grad, self.weight.T)
        self.weight_grad = np.dot(self.input.T, output_grad)
        self.bias_grad = np.sum(output_grad, axis=0, keepdims=True)
        return input_grad

    def update(self, learning_rate, optimizer):
        # Gradient descent optimizer
        if optimizer == 'GD':
            self.weight -= learning_rate * self.weight_grad
            self.bias -= learning_rate * self.bias_grad
        # Momentum optimizer
        if optimizer == 'Momentum':
            self.vw = 0.9 * self.vw - learning_rate * self.weight_grad
            self.weight += self.vw
            self.vb = 0.9 * self.vb - learning_rate * self.bias_grad
            self.bias += self.vb
```

I write the Linear layer (Fully Connected Layer) as a class. You can decide the input and the output dimensions of the linear layer. It has three functions: forward, backward and update. Forward function and

backward use as forward and backward process of the network. Update function is used to update the weights of the network by gradient descent method.

```
class Network():
    def __init__(self, input_dim, hidden1_dim, hidden2_dim, output_dim, learning_rate, optimizer):
        self.fc1 = LinearLayer(input_dim, hidden1_dim)
        self.sigmoid1 = sigmoid()
        self.fc2 = LinearLayer(hidden1_dim, hidden2_dim)
        self.sigmoid2 = sigmoid()
        self.fc3 = LinearLayer(hidden2_dim, output_dim)
        self.sigmoid3 = sigmoid()
        self.loss = MSE()
        self.lr = learning_rate
        self.optimizer = optimizer

    def forward(self, x):
        # First hidden layer (Fully Connected Layer)
        x = self.fc1.forward(x)
        x = self.sigmoid1.forward(x)
        # Second hidden layer (Fully Connected Layer)
        x = self.fc2.forward(x)
        x = self.sigmoid2.forward(x)
        # Output layer
        x = self.fc3.forward(x)
        x = self.sigmoid3.forward(x)
        return x

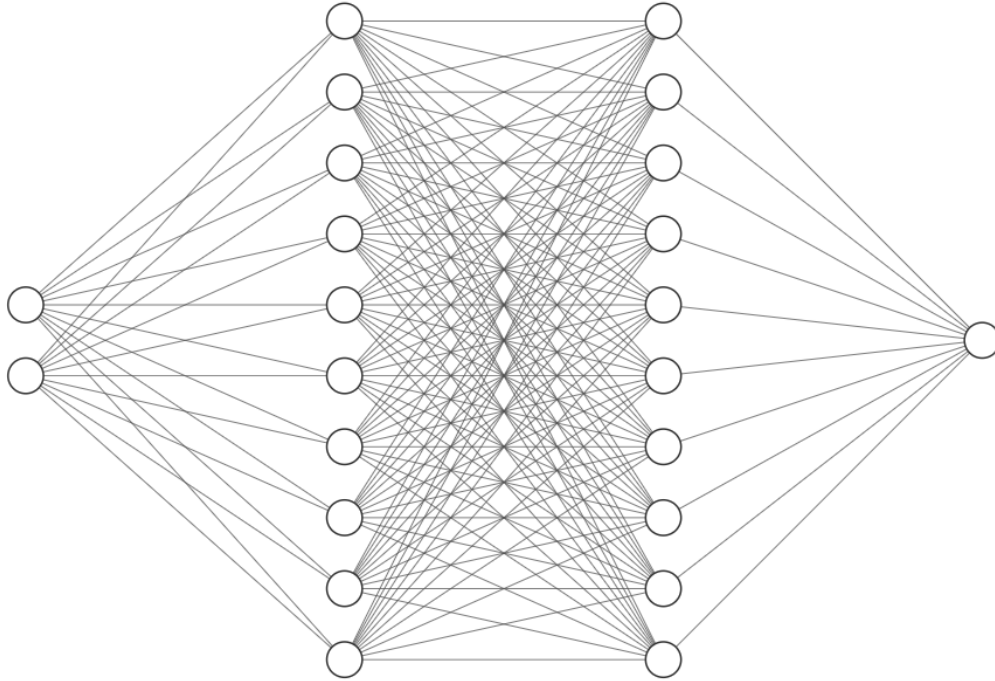
    def compute_loss(self, pred, label):
        loss = self.loss.forward(pred, label)
        return loss

    def backward(self):
        grad = self.loss.backward()
        grad = self.sigmoid3.backward(grad)
        grad = self.fc3.backward(grad)
        grad = self.sigmoid2.backward(grad)
        grad = self.fc2.backward(grad)
        grad = self.sigmoid1.backward(grad)
        grad = self.fc1.backward(grad)

    def update(self):
        self.fc1.update(learning_rate=self.lr, optimizer=self.optimizer)
        self.fc2.update(learning_rate=self.lr, optimizer=self.optimizer)
        self.fc3.update(learning_rate=self.lr, optimizer=self.optimizer)
```

This is the architecture of the network, which is a network of two hidden layers. It has four functions: forward, backward, compute_loss and update. Forward and backward functions are the forward and backward process of the network. Update function is to update the

network's weights to do the optimization. Compute_loss function is to compute the loss.



Architecture of the network of this lab

Input: 2 dimensions, Hidden1: 10 dimensions, Hidden2: 10 dimensions, Output: 1 dimension

```
class MSE():
    def __init__(self):
        self.pred = None
        self.label = None

    def forward(self, pred, label):
        self.pred = pred
        self.label = label
        output = np.mean((self.pred - self.label) ** 2)
        return output

    def backward(self):
        return 2 * (self.pred - self.label) / np.size(self.label)
```

For this lab, I use Mean Square Error as my loss function. I write it as a class and it has two functions: one is forward and the other one is

backward. Forward and backward functions use to do the forward and backward process of the network.

$$MSE = \sum_{i=0}^n (y_i - y_{i_{predict}})^2 \quad MSE' = \frac{2 (y_i - y_{i_{predict}})^2}{n}$$

Equation of Mean Square Error and its derivative

C. Backpropagation

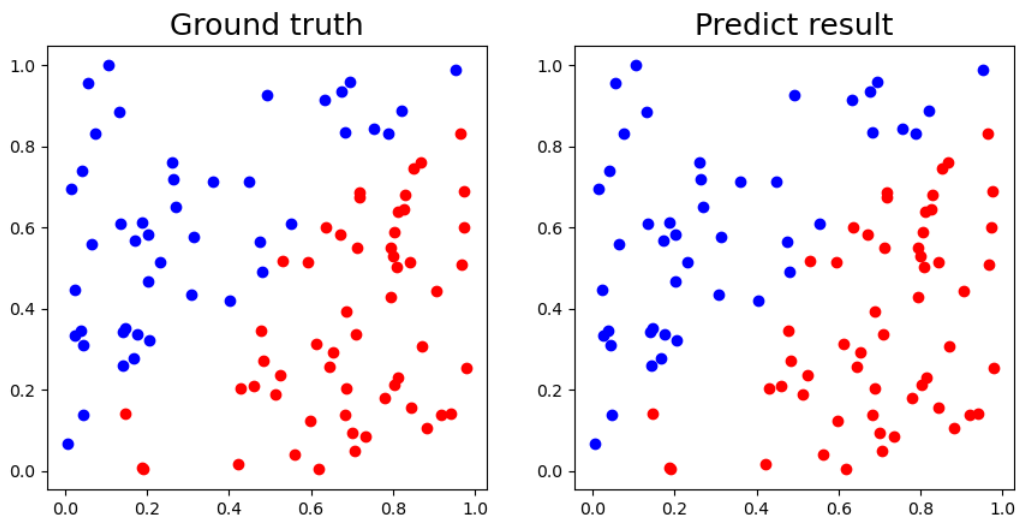
```
def backward(self):  
    grad = self.loss.backward()  
    grad = self.sigmoid3.backward(grad)  
    grad = self.fc3.backward(grad)  
    grad = self.sigmoid2.backward(grad)  
    grad = self.fc2.backward(grad)  
    grad = self.sigmoid1.backward(grad)  
    grad = self.fc1.backward(grad)
```

Backpropagation use chain rule to compute the gradient as mentioned above. The backward function of the network computes the gradients one by one and each sub backward can be found in the picture above.

3. Results of your testing

Linear dataset:

A. Screenshot and comparison figure



```

Task 1 for linear dataset
Epoch: 5000    loss: 0.12350983253255791
Epoch: 10000   loss: 0.06826516259391478
Epoch: 15000   loss: 0.04920117838421343
Epoch: 20000   loss: 0.03956107889413234
Epoch: 25000   loss: 0.033683896018540514
Epoch: 30000   loss: 0.02967105510145631
Epoch: 35000   loss: 0.02671683943630291
Epoch: 40000   loss: 0.024424627047321356
Epoch: 45000   loss: 0.02257713673798782
Epoch: 50000   loss: 0.021045469762395283
Epoch: 55000   loss: 0.019748209814595418
Epoch: 60000   loss: 0.01863118585631071
Epoch: 65000   loss: 0.017656721218839367
Epoch: 70000   loss: 0.016797598625085567
Epoch: 75000   loss: 0.016033513353848207
Epoch: 80000   loss: 0.015348902094656362
Epoch: 85000   loss: 0.014731563252618051
Epoch: 90000   loss: 0.01417174921783394
Epoch: 95000   loss: 0.013661549684488444
Epoch: 100000  loss: 0.013194460248180757

```

I trained the network for 100,000 epoch and print the loss every 5000 epochs. The final loss is 0.013194.

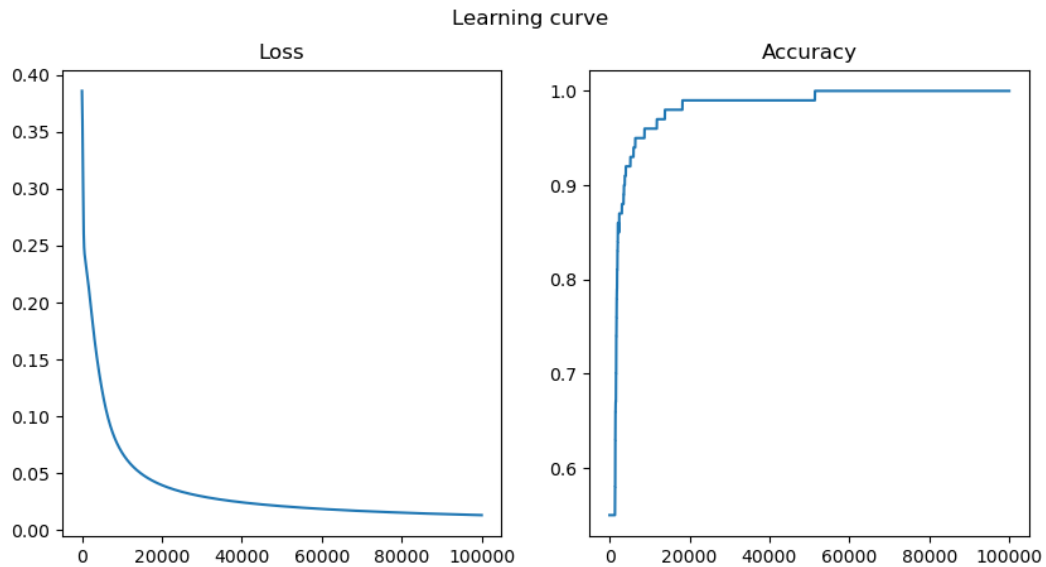
B. Show the accuracy of your prediction

```

Ground truth: [1] | prediction: 0.9425314946237633
Ground truth: [1] | prediction: 0.9934715595396175
Ground truth: [0] | prediction: 0.0010775636042220616
Ground truth: [0] | prediction: 0.0009795920211881784
Ground truth: [0] | prediction: 0.004318340600588959
Ground truth: [0] | prediction: 0.010196932783329966
Ground truth: [1] | prediction: 0.7923983644003417
Ground truth: [0] | prediction: 0.0018732103585983235
Ground truth: [0] | prediction: 0.0004933578597688558
Ground truth: [1] | prediction: 0.9944502624547004
Ground truth: [0] | prediction: 0.4146528694263135
Ground truth: [1] | prediction: 0.9991798475718248
Ground truth: [0] | prediction: 0.0020353400125099523
Ground truth: [0] | prediction: 0.0033032024451783654
Ground truth: [1] | prediction: 0.9992758237060494
Ground truth: [0] | prediction: 0.0018376947280931
Ground truth: [1] | prediction: 0.9970677354365985
Ground truth: [1] | prediction: 0.9933767714687958
Ground truth: [0] | prediction: 0.002671694705880353
Ground truth: [0] | prediction: 0.008462344411317246
Ground truth: [1] | prediction: 0.998117119797493
Accuracy: 100.0%

```

C. Learning curve (loss, epoch curve)

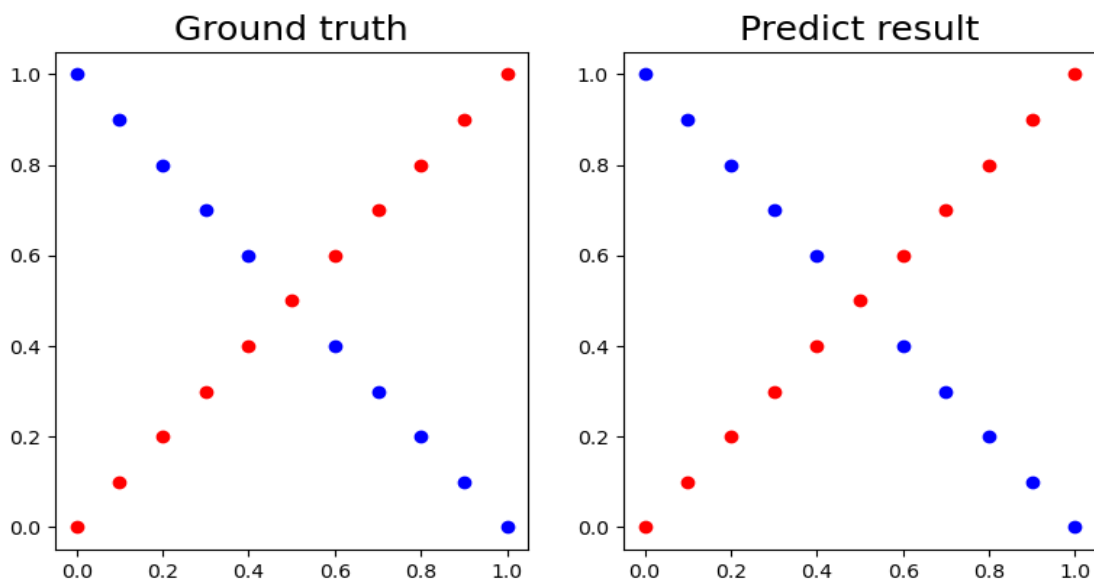


The loss curve dropped suddenly at first and then went flat at about 18000 epoch.

D. Anything you want to present

XOR dataset

Screenshot and comparison figure



```

Epoch: 5000      loss: 0.24841044370226
Epoch: 10000     loss: 0.24722997328463833
Epoch: 15000     loss: 0.24580484400434427
Epoch: 20000     loss: 0.24392763198426798
Epoch: 25000     loss: 0.2413076154264643
Epoch: 30000     loss: 0.23754216819095525
Epoch: 35000     loss: 0.2321552659779917
Epoch: 40000     loss: 0.22468156235113565
Epoch: 45000     loss: 0.21454457997091053
Epoch: 50000     loss: 0.20039338168425797
Epoch: 55000     loss: 0.1795710473986481
Epoch: 60000     loss: 0.15105367735752515
Epoch: 65000     loss: 0.12124047978454071
Epoch: 70000     loss: 0.09726995126462903
Epoch: 75000     loss: 0.0796757300956581
Epoch: 80000     loss: 0.0664751903614729
Epoch: 85000     loss: 0.05605526580849528
Epoch: 90000     loss: 0.047460509871189514
Epoch: 95000     loss: 0.04015713105391166
Epoch: 100000    loss: 0.033876264208719996

```

I trained the network for 100,000 epoch and print the loss every 5000 epochs. The final loss is 0.033876.

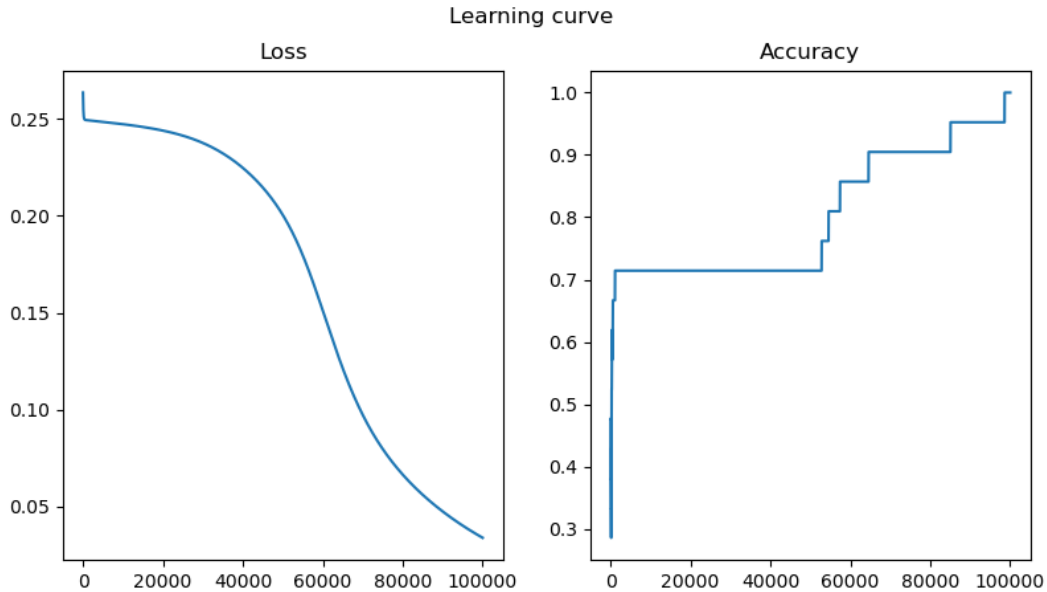
Show the accuracy of your prediction

```

Ground truth: [0] | prediction: 0.16761632526791792
Ground truth: [1] | prediction: 0.9848651433654255
Ground truth: [0] | prediction: 0.1664957906545853
Ground truth: [1] | prediction: 0.9799625496463747
Ground truth: [0] | prediction: 0.16551405550646844
Ground truth: [1] | prediction: 0.9656561690810715
Ground truth: [0] | prediction: 0.16466315002009516
Ground truth: [1] | prediction: 0.8994026562964675
Ground truth: [0] | prediction: 0.1639320490284902
Ground truth: [1] | prediction: 0.5120779954410275
Ground truth: [0] | prediction: 0.1633063681991056
Ground truth: [0] | prediction: 0.16276848242157363
Ground truth: [1] | prediction: 0.617180124447482
Ground truth: [0] | prediction: 0.16229808978436538
Ground truth: [1] | prediction: 0.9102677632187779
Ground truth: [0] | prediction: 0.16187317812525281
Ground truth: [1] | prediction: 0.9334177143861316
Ground truth: [0] | prediction: 0.16147128889134796
Ground truth: [1] | prediction: 0.9380210611184541
Ground truth: [0] | prediction: 0.16107092696517283
Ground truth: [1] | prediction: 0.9401651178572678
Accuracy: 100.0%

```


Learning curve (loss, epoch curve)



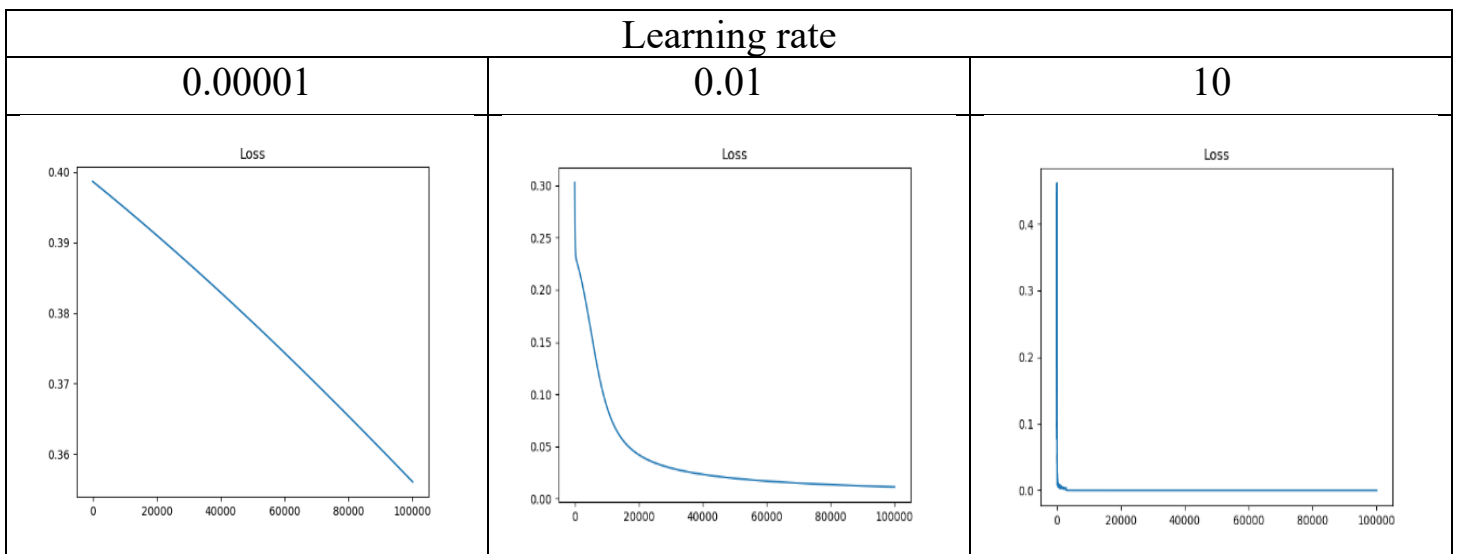
We can see that the loss curve had a huge drop at first and then went flat for a while. At last, it went down slowly at about 50000 epoch.

4. Discussion

A. Try different learning rates

I set the learning rate to 0.00001, 0.01, 100 to compare the results.

Network architecture, optimizer the same.



Accuracy: 48.0%	Accuracy: 100.0%	Accuracy: 100.0%
------------------------	-------------------------	-------------------------

We can see that for learning rate: 0.00001, the loss nearly does not decrease. For the first epoch, the loss is 0.39 and for the last epoch, the loss is 0.35.

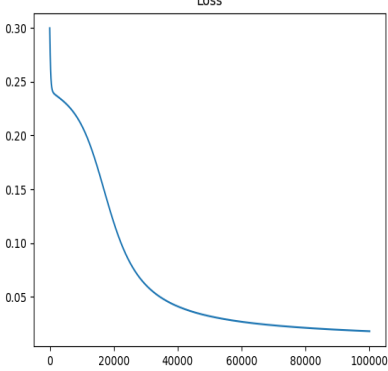
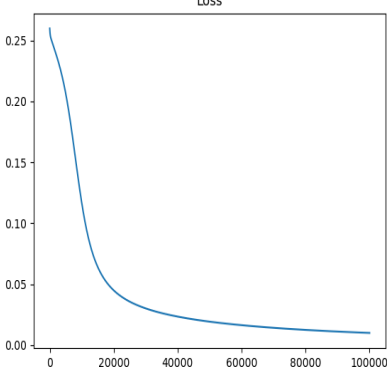
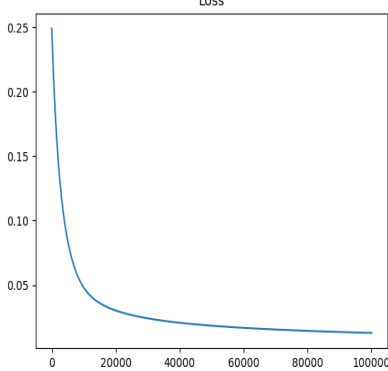
For learning rate: 0.01, it is normal. The loss value decreases as epoch goes up.

For learning rate: 10, the loss value drops so quick.

B. Try different numbers of hidden units

The original network has two hidden with each 10 neurons. Now I set the hidden neurons to each 2 and each 20.

The rest of the settings are the same.

Hidden units (first hidden layer, second hidden layer)		
(2, 2)	(10, 10)	(20, 20)
		
Accuracy: 98.0%	Accuracy: 100.0%	Accuracy: 100.0%

Only the (2, 2) hidden units has lower accuracy, rest of the experiments both have 100% accuracy. I think since the task is relatively easy, so the result does not have quite different.

C. Try without activation functions

I commented the activation function part (sigmoid function) to remove the activation function in the network.

```
class Network():
    def __init__(self, input_dim, hidden1_dim, hidden2_dim, output_dim, learning_rate, optimizer):
        self.fc1 = LinearLayer(input_dim, hidden1_dim)
        #self.sigmoid1 = sigmoid()
        self.fc2 = LinearLayer(hidden1_dim, hidden2_dim)
        #self.sigmoid2 = sigmoid()
        self.fc3 = LinearLayer(hidden2_dim, output_dim)
        #self.sigmoid3 = sigmoid()
        self.loss = MSE()
        self.lr = learning_rate
        self.optimizer = optimizer

    def forward(self, x):
        # First hidden layer (Fully Connected Layer)
        x = self.fc1.forward(x)
        #x = self.sigmoid1.forward(x)
        # Second hidden layer (Fully Connected Layer)
        x = self.fc2.forward(x)
        #x = self.sigmoid2.forward(x)
        # Output layer
        x = self.fc3.forward(x)
        #x = self.sigmoid3.forward(x)
        return x

    def compute_loss(self, pred, label):
        loss = self.loss.forward(pred, label)
        return loss

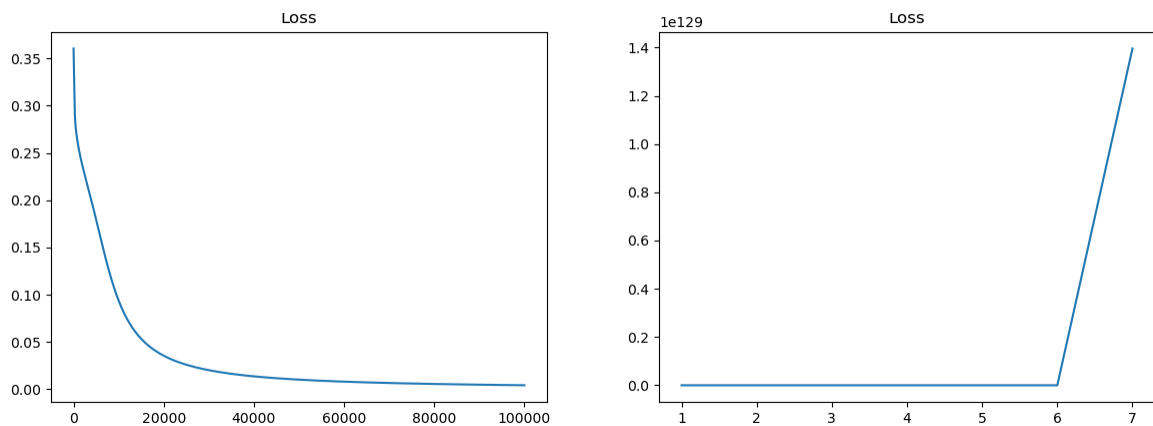
    def backward(self):
        grad = self.loss.backward()
        #grad = self.sigmoid3.backward(grad)
        grad = self.fc3.backward(grad)
        #grad = self.sigmoid2.backward(grad)
        grad = self.fc2.backward(grad)
        #grad = self.sigmoid1.backward(grad)
        grad = self.fc1.backward(grad)

    def update(self):
        self.fc1.update(learning_rate=self.lr, optimizer=self.optimizer)
        self.fc2.update(learning_rate=self.lr, optimizer=self.optimizer)
        self.fc3.update(learning_rate=self.lr, optimizer=self.optimizer)
```

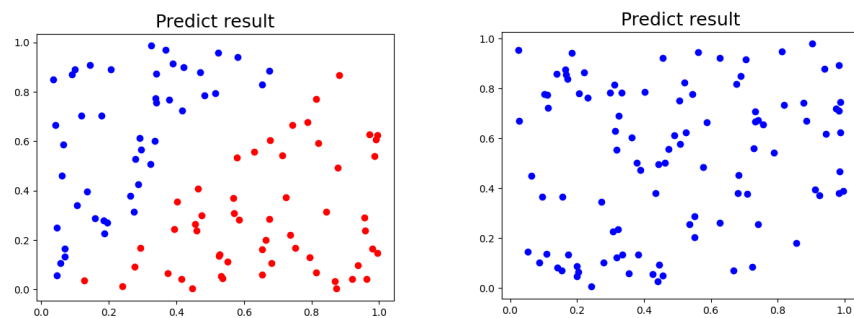
Result shows that the loss of the network just explode which leads to the prediction failed.

Epoch: 5000	loss: nan	Ground truth: [1]	prediction: nan
Epoch: 10000	loss: nan	Ground truth: [0]	prediction: nan
Epoch: 15000	loss: nan	Ground truth: [1]	prediction: nan
Epoch: 20000	loss: nan	Ground truth: [0]	prediction: nan
Epoch: 25000	loss: nan	Ground truth: [0]	prediction: nan
Epoch: 30000	loss: nan	Ground truth: [1]	prediction: nan
Epoch: 35000	loss: nan	Ground truth: [1]	prediction: nan
Epoch: 40000	loss: nan	Ground truth: [0]	prediction: nan
Epoch: 45000	loss: nan	Ground truth: [0]	prediction: nan
Epoch: 50000	loss: nan	Ground truth: [1]	prediction: nan
Epoch: 55000	loss: nan	Ground truth: [1]	prediction: nan
Epoch: 60000	loss: nan	Ground truth: [1]	prediction: nan
Epoch: 65000	loss: nan	Ground truth: [1]	prediction: nan
Epoch: 70000	loss: nan	Ground truth: [1]	prediction: nan
Epoch: 75000	loss: nan	Ground truth: [0]	prediction: nan
Epoch: 80000	loss: nan	Ground truth: [0]	prediction: nan
Epoch: 85000	loss: nan	Ground truth: [0]	prediction: nan
Epoch: 90000	loss: nan	Ground truth: [0]	prediction: nan
Epoch: 95000	loss: nan	Ground truth: [1]	prediction: nan
Epoch: 100000	loss: nan	Accuracy: 0.0%	

Training (left) and Testing (right) history without activation function



With (left) and without (right) activation function



Prediction result

5. Extra

A. Implement different optimizers

Momentum optimizer

```
def update(self, learning_rate, optimizer):  
    # Gradient descent optimizer  
    if optimizer == 'GD':  
        self.weight -= learning_rate * self.weight_grad  
        self.bias -= learning_rate * self.bias_grad  
    # Momentum optimizer  
    if optimizer == 'Momentum':  
        self.vw = 0.9 * self.vw - learning_rate * self.weight_grad  
        self.weight += self.vw  
        self.vb = 0.9 * self.vb - learning_rate * self.bias_grad  
        self.bias += self.vb
```

$$V_t \leftarrow \beta V_{t-1} - \eta \frac{\partial L}{\partial W}$$

$$W_t \leftarrow W_{t-1} + V_{t-1}$$

Momentum Weight update equation

B. Implement different activation functions

ReLU activation function

```
class relu():  
    def __init__(self):  
        self.input = None  
        self.output = None  
  
    def forward(self, x):  
        self.input = x  
        self.output = np.maximum(self.input, 0)  
        return self.output  
  
    def backward(self, output_grad):  
        relu_grad = float((self.input > 0))  
        input_grad = output_grad * relu_grad  
        return input_grad
```

$$f(x) = \max(0, x) \qquad f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

Equation of ReLU (left) and its derivative (right)