1. Introduction (20%).

In this assignment, we aim to implement neural networks, forward pass, calculate loss, and backpropagation using the numpy library, mimicking those in the PyTorch library.

In the code, we use class object to create a network and define class function:

- forwardPass(self, x):
  In this function, we forward the input data x through the network and obtain the output. Additionally, we record the outputs of each layer in the network, which makes it easier for us to implement backpropagation.
- computeMSELoss(self, pred, label):
  In this function, we compute the Mean Squared Error (MSE) loss between the predicted results and the ground truth labels.
- Backpropagation(self):
  In this function, we implement backpropagation, including computing gradients and updating network weights.
- activateFunv(self, input, func='sigmoid'):
  In this function, we apply the input through the activation function, which can be chosen among ReLU and sigmoid by the variable 'func'.
- derivative_activateFunv(self, input, func='sigmoid'):
  In this function, we compute the gradient of the activation function 'activateFunv', which can also be chosen among ReLU and sigmoid.

2. Experiment setups (30%):

A. Sigmoid functions:

In this section, we illustrate the sigmoid function and its gradient.

Sigmoid function : $\sigma(\mathrm{x}) = \frac{1}{1+e^{-x}}$

Gradient of sigmoid function:

$$\sigma'(\mathrm{x}) = \left(\frac{1}{1+e^{-x}}\right)' = \frac{-e^{-x}}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}} \cdot \left(1 - \frac{1}{1+e^{-x}}\right) = \sigma(\mathrm{x}) \cdot (1 - \sigma(\mathrm{x}))$$

```python
def activateFunv(self, input, func='sigmoid'):
    if func == 'sigmoid':
        output = 1/(1+np.exp(-input))
```

```python
def derivative_activateFunv(self, input, func='sigmoid'):
    if func == 'sigmoid':
        x = self.activateFunv(input, func='sigmoid')
        output = np.multiply(x , 1.0 - x)
```

B. Neural network:

In this section, we illustrate some of the settings of the neural network. We initialize the following parameters in the '__init__ ' of the Network class.

```
def __init__(self,input_size = 2, hidden_size=32, output_size=1, lr=1e-3, actFuncs=['sigmoid','sigmoid','sigmoid']):

    #Initialize the network parameter
    self.W1 = np.random.normal(0, 1, (input_size, hidden_size))
    self.W2 = np.random.normal(0, 1, (hidden_size, hidden_size))
    self.W3 = np.random.normal(0, 1, (hidden_size, output_size))
    self.lr = lr
    self.actFuncs= actFuncs
```

- Initialized the network weight parameter: [0,1] normal random value.
- Activate function:

  We implement the following three activation functions,

  ①. ReLU: $\mathrm{ReLU(x)} = \max\{0, x\}$

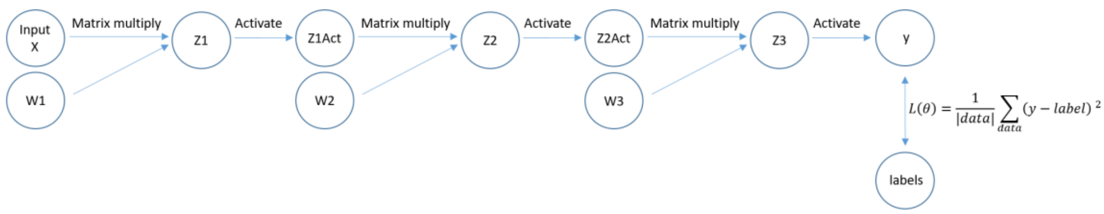
  ②. Sigmoid: $\sigma(\mathrm{x}) = \frac{1}{1+e^{-x}}$

  <u>We initialize the variable self.actFuncs as a list, where self.actFuncs[i] represents the choice of activation function for the i-th layer.</u>

C. Backpropagation:

In this part, we derive the gradient of each layer's parameters.

The network flowchart:



Then, we compute the gradients of W3, W2, and W1. For convenience, we set the activation function to sigmoid.

We use ■ denote the matrix multiplication and · denote the entrywise product.

$$\frac{\partial \mathrm{L}}{\partial \mathrm{y}} = 2(y - label)$$

$$\frac{\partial \mathrm{L}}{\partial \mathrm{Z3}} = \frac{\partial \mathrm{L}}{\partial \mathrm{y}}\frac{\partial \mathrm{y}}{\partial \mathrm{Z3}} = \frac{\partial \mathrm{L}}{\partial \mathrm{y}} \cdot \left[\frac{1}{1+e^{-Z3}}\left(1 - \frac{1}{1+e^{-Z3}}\right)\right],$$

$$\frac{\partial \mathrm{L}}{\partial \mathrm{W3}} = \frac{\partial \mathrm{L}}{\partial \mathrm{Z3}}\frac{\partial \mathrm{Z3}}{\partial \mathrm{W3}} = (Z2Act)^T \blacksquare \frac{\partial \mathrm{L}}{\partial \mathrm{Z3}}$$

$$\frac{\partial \mathrm{L}}{\partial \mathrm{Z2}} = \frac{\partial \mathrm{L}}{\partial \mathrm{Z3}}\frac{\partial \mathrm{Z3}}{\partial \mathrm{Z2Act}}\frac{\partial \mathrm{Z2Act}}{\partial \mathrm{Z2}} = \left(\frac{\partial \mathrm{L}}{\partial \mathrm{Z3}} \blacksquare W3\right) \cdot \left[\frac{1}{1+e^{-Z2}}\left(1 - \frac{1}{1+e^{-Z2}}\right)\right]$$

$$\frac{\partial \mathrm{L}}{\partial \mathrm{W2}} = \frac{\partial \mathrm{L}}{\partial \mathrm{Z2}}\frac{\partial \mathrm{Z2}}{\partial \mathrm{W2}} = (Z1Act)^T \blacksquare \frac{\partial \mathrm{L}}{\partial \mathrm{Z2}}$$

$$\frac{\partial \mathrm{L}}{\partial \mathrm{Z1}} = \frac{\partial \mathrm{L}}{\partial \mathrm{Z2}}\frac{\partial \mathrm{Z2}}{\partial \mathrm{Z1Act}}\frac{\partial \mathrm{Z1Act}}{\partial \mathrm{Z1}} = \left(\frac{\partial \mathrm{L}}{\partial \mathrm{Z2}} \blacksquare W2\right) \cdot \left[\frac{1}{1+e^{-Z1}}\left(1 - \frac{1}{1+e^{-Z1}}\right)\right]$$

$$\frac{\partial \mathrm{L}}{\partial \mathrm{W1}} = \frac{\partial \mathrm{L}}{\partial \mathrm{Z1}}\frac{\partial \mathrm{Z1}}{\partial \mathrm{W1}} = (Input\ \ X)^T \blacksquare \frac{\partial \mathrm{L}}{\partial \mathrm{Z1}}$$

Lastly, we update the parameter:

$$W1' = W1 - lr * \frac{\partial L}{\partial W1}$$

$$W2' = W2 - lr * \frac{\partial L}{\partial W2}$$

$$W3' = W3 - lr * \frac{\partial L}{\partial W3}$$

```
def Backpropagation(self):
    grad_LossAct = self.derivative_activateFunv(self.Z3, func=self.actFuncs[2]) * self.GradLoss
    grad_W3 = self.Z2Act.T @ grad_LossAct
    grad_Z2Act = grad_LossAct @ self.W3.T
    grad_Z2 = self.derivative_activateFunv(self.Z2, func=self.actFuncs[1]) * grad_Z2Act
    grad_W2 = self.Z1Act.T @ grad_Z2
    grad_Z1Act = grad_Z2 @ self.W2.T
    grad_Z1 = self.derivative_activateFunv(self.Z1, func=self.actFuncs[0]) * grad_Z1Act
    grad_W1 = x.T @ grad_Z1

    self.W3 -= self.lr * grad_W3
    self.W2 -= self.lr * grad_W2
    self.W1 -= self.lr * grad_W1
```

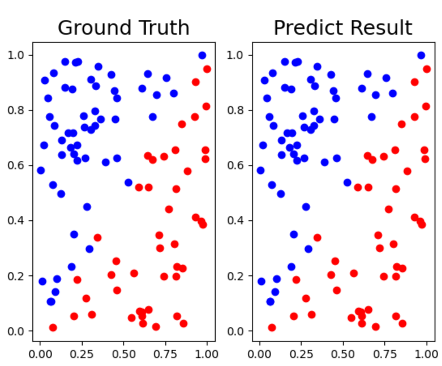
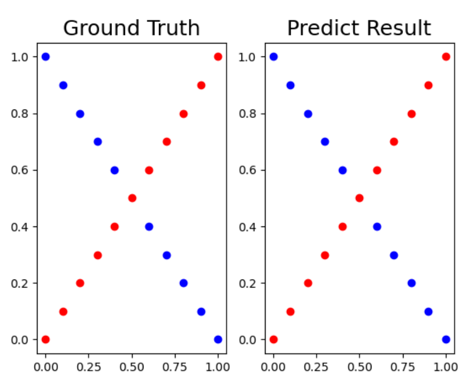
3. Results of your testing (20%)

In this part, we set following hyperparameters:

- Learning Rate = 5e-2
- Hidden layer size = 32
- Max Epochs = 10000
- Activate function(self.actFuncs) = [‘sigmoid’, ‘sigmoid’, ‘sigmoid’]

Since our the activate function of output layer is sigmoid, which value is within [0,1]. For the prediction, if value > 0.5, set predict class=1, otherwise set predict class=0.

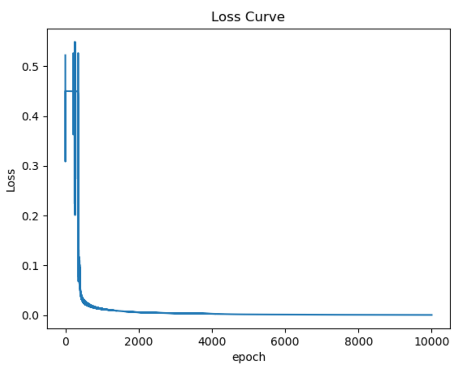
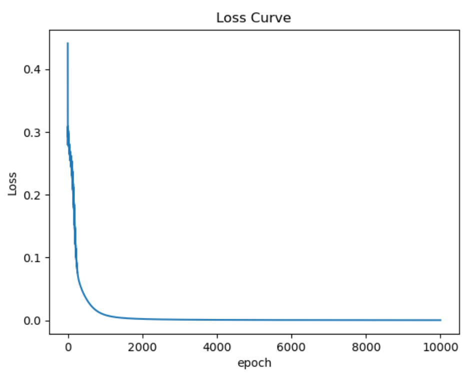
A. Screenshot and comparison figure

Linear(n=100)	XOR
	

B. Show the accuracy of your prediction

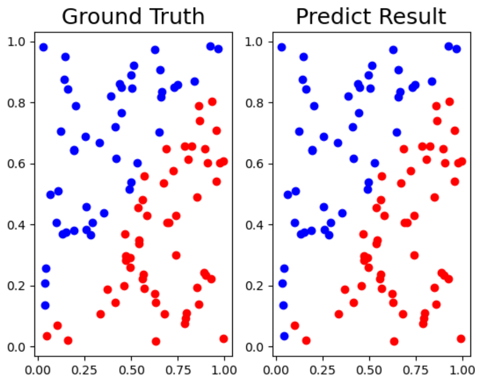
Linear(n=100)	XOR
Iter 95 | Ground Truth: 0| Prediction: 0.00000 | Iter 96 | Ground Truth: 0| Prediction: 0.00000 | Iter 97 | Ground Truth: 0| Prediction: 0.00000 | Iter 98 | Ground Truth: 0| Prediction: 0.00000 | Iter 99 | Ground Truth: 0| Prediction: 0.00000 | Iter100 | Ground Truth: 1| Prediction: 1.00000 | Loss = 0.00366, accuracy = 100.0%	Iter 16 | Ground Truth: 0| Prediction: 0.00837 | Iter 17 | Ground Truth: 1| Prediction: 0.99698 | Iter 18 | Ground Truth: 0| Prediction: 0.00515 | Iter 19 | Ground Truth: 1| Prediction: 0.99747 | Iter 20 | Ground Truth: 0| Prediction: 0.00331 | Iter 21 | Ground Truth: 1| Prediction: 0.99755 | Loss = 0.00975, accuracy = 100.0%

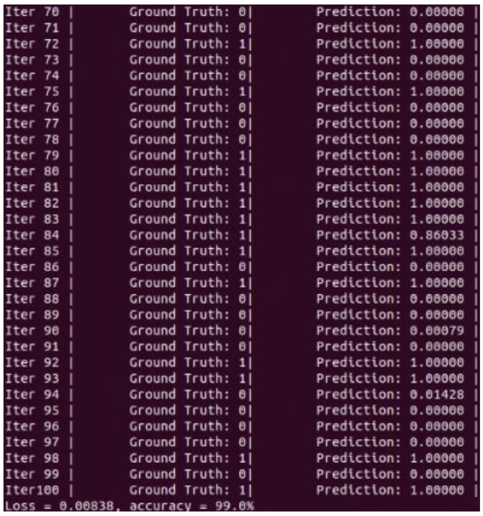
C. Learning curve (loss, epoch curve)

Linear(n=100)	XOR
epoch: 500 Loss: 0.02475 epoch: 1000 Loss: 0.01197 epoch: 1500 Loss: 0.00831 epoch: 2000 Loss: 0.00594 epoch: 2500 Loss: 0.00482 epoch: 3000 Loss: 0.00411 epoch: 3500 Loss: 0.00352 epoch: 4000 Loss: 0.00287 epoch: 4500 Loss: 0.00202 epoch: 5000 Loss: 0.00156 epoch: 5500 Loss: 0.00136 epoch: 6000 Loss: 0.00119 epoch: 6500 Loss: 0.00105 epoch: 7000 Loss: 0.00093 epoch: 7500 Loss: 0.00082 epoch: 8000 Loss: 0.00073 epoch: 8500 Loss: 0.00066 epoch: 9000 Loss: 0.00059 epoch: 9500 Loss: 0.00053 epoch: 10000 Loss: 0.00048	epoch: 500 Loss: 0.03408 epoch: 1000 Loss: 0.00857 epoch: 1500 Loss: 0.00358 epoch: 2000 Loss: 0.00207 epoch: 2500 Loss: 0.00140 epoch: 3000 Loss: 0.00104 epoch: 3500 Loss: 0.00082 epoch: 4000 Loss: 0.00067 epoch: 4500 Loss: 0.00057 epoch: 5000 Loss: 0.00049 epoch: 5500 Loss: 0.00043 epoch: 6000 Loss: 0.00038 epoch: 6500 Loss: 0.00034 epoch: 7000 Loss: 0.00031 epoch: 7500 Loss: 0.00028 epoch: 8000 Loss: 0.00026 epoch: 8500 Loss: 0.00024 epoch: 9000 Loss: 0.00023 epoch: 9500 Loss: 0.00021 epoch: 10000 Loss: 0.00020
	

D. Anything you want to present

In the case of linear case, after training, we generate another 100 data points (which the network has not seen) to test its performance.





```
Iter 70 |        Ground Truth: 0|        Prediction: 0.00000 |
Iter 71 |        Ground Truth: 0|        Prediction: 0.00000 |
Iter 72 |        Ground Truth: 1|        Prediction: 1.00000 |
Iter 73 |        Ground Truth: 0|        Prediction: 0.00000 |
Iter 74 |        Ground Truth: 0|        Prediction: 0.00000 |
Iter 75 |        Ground Truth: 1|        Prediction: 1.00000 |
Iter 76 |        Ground Truth: 0|        Prediction: 0.00000 |
Iter 77 |        Ground Truth: 0|        Prediction: 0.00000 |
Iter 78 |        Ground Truth: 0|        Prediction: 0.00000 |
Iter 79 |        Ground Truth: 1|        Prediction: 1.00000 |
Iter 80 |        Ground Truth: 1|        Prediction: 1.00000 |
Iter 81 |        Ground Truth: 1|        Prediction: 1.00000 |
Iter 82 |        Ground Truth: 1|        Prediction: 1.00000 |
Iter 83 |        Ground Truth: 1|        Prediction: 1.00000 |
Iter 84 |        Ground Truth: 1|        Prediction: 0.86033 |
Iter 85 |        Ground Truth: 1|        Prediction: 1.00000 |
Iter 86 |        Ground Truth: 0|        Prediction: 0.00000 |
Iter 87 |        Ground Truth: 1|        Prediction: 1.00000 |
Iter 88 |        Ground Truth: 0|        Prediction: 0.00000 |
Iter 89 |        Ground Truth: 0|        Prediction: 0.00000 |
Iter 90 |        Ground Truth: 0|        Prediction: 0.00079 |
Iter 91 |        Ground Truth: 0|        Prediction: 0.00000 |
Iter 92 |        Ground Truth: 1|        Prediction: 1.00000 |
Iter 93 |        Ground Truth: 1|        Prediction: 1.00000 |
Iter 94 |        Ground Truth: 0|        Prediction: 0.01428 |
Iter 95 |        Ground Truth: 0|        Prediction: 0.00000 |
Iter 96 |        Ground Truth: 0|        Prediction: 0.00000 |
Iter 97 |        Ground Truth: 0|        Prediction: 0.00000 |
Iter 98 |        Ground Truth: 1|        Prediction: 1.00000 |
Iter 99 |        Ground Truth: 0|        Prediction: 0.00000 |
Iter100 |        Ground Truth: 1|        Prediction: 1.00000 |
Loss = 0.00838, accuracy = 99.0%
```

As a result, we can observe that the network's performance remains good for the newly generated data.

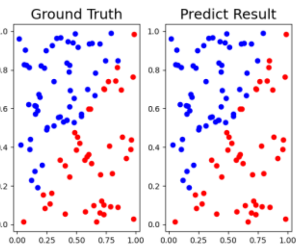
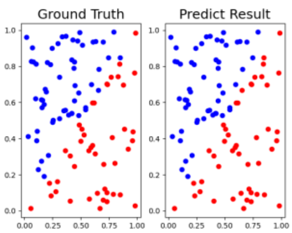
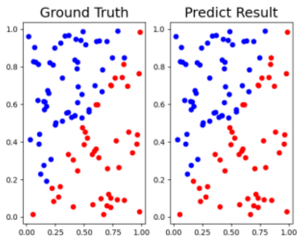
4. Discussion (30%)

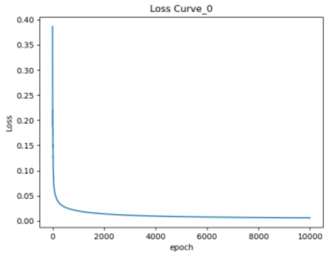
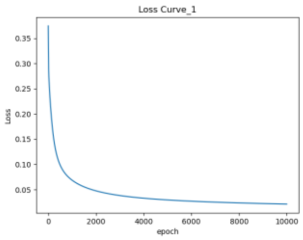
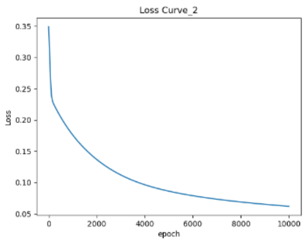
In the following A/B parts, except for the specifically discussed hyperparameter, all other settings remain the same.

- Learning Rate = 5e-2
- Hidden layer size = 32
- Max Epochs = 10000
- Activate function(self.actFuncs) = [‘sigmoid’, ‘sigmoid’, ‘sigmoid’]

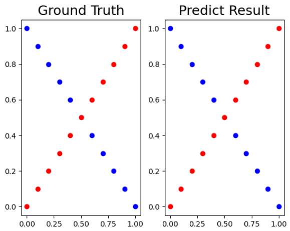
A. Try different learning rates

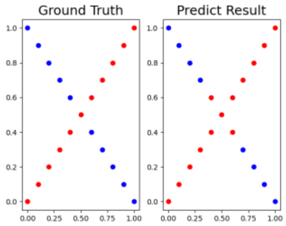
Linea Case:

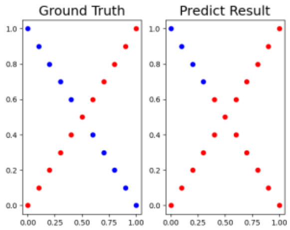
Learning rate = 1e-2	Learning rate = 1e-3	Learning rate = 1e-4
 Loss = 0.01943 Accuracy = 99.0%	 Loss = 0.06566 Accuracy=98.0%	 Loss = 0.19179 Accuracy = 98.0%

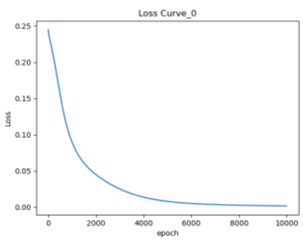
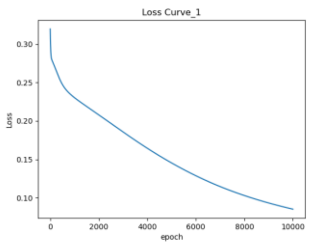
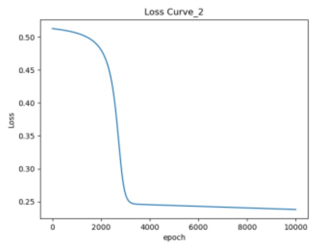
<table>
<tr>
<td>
</td>
<td>
</td>
<td>
</td>
</tr>
</table>

XOR case:

<table>
<tr>
<td>Learning rate = 1e-2</td>
<td>Learning rate = 1e-3</td>
<td>Learning rate = 1e-4</td>
</tr>
<tr>
<td>

Predict Loss = 0.02780
Accuracy = 100.0%</td>
<td>

Predict Loss = 0.26000
Accuracy=90.5%</td>
<td>

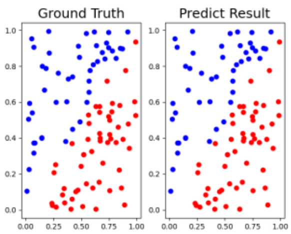
Predict Loss = 0.48589
Accuracy = 71.4%</td>
</tr>
<tr>
<td>
</td>
<td>
</td>
<td>
</td>
</tr>
</table>

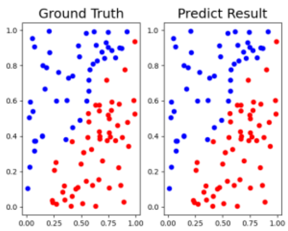
Conclusion:

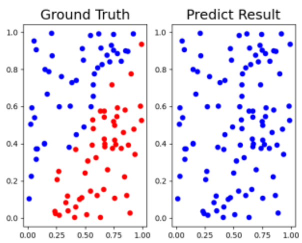
The appropriate learning rate can help the network find a better minimum. If the learning rate is too large, the network may fail to converge, while if it is too small, it may take more epochs to converge.

B. Try different numbers of hidden units

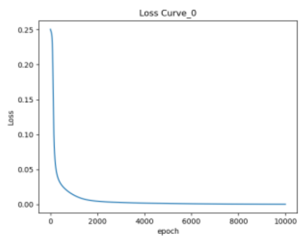
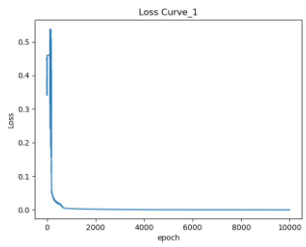
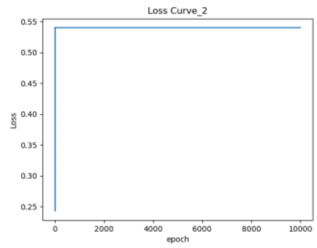
Linea Case:

<table>
<tr>
<td>Hidden units = 2</td>
<td>Hidden units = 32</td>
<td>Hidden units = 128</td>
</tr>
<tr>
<td>

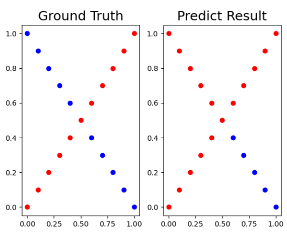
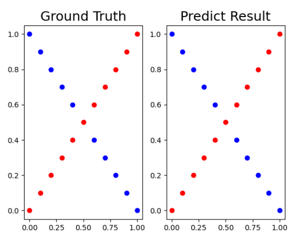
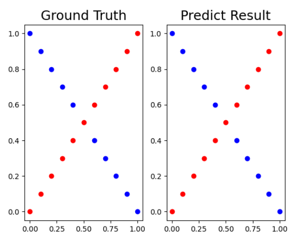
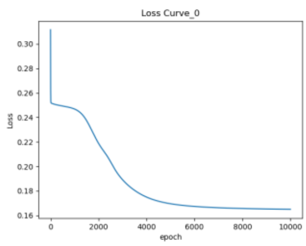
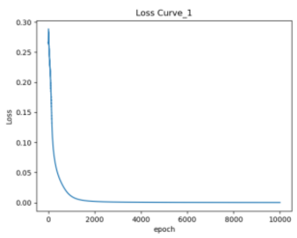
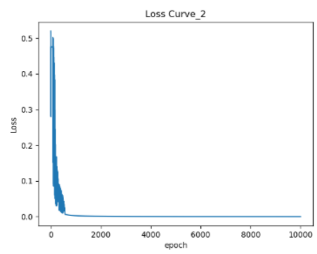
Loss = 0.00333
Accuracy = 100.0%</td>
<td>

Loss = 0.00214
Accuracy=100.0%</td>
<td>

Loss = 0.54000
Accuracy = 46.0%</td>
</tr>
</table>

XOR case:

Hidden units = 2	Hidden units = 32	Hidden units = 128
 Loss = 0.33285 Accuracy = 76.2%	 Loss = 0.00745 Accuracy=100.0%	 Loss = 0.00511 Accuracy = 100.0%
		

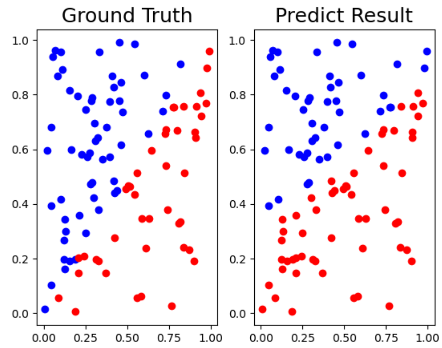
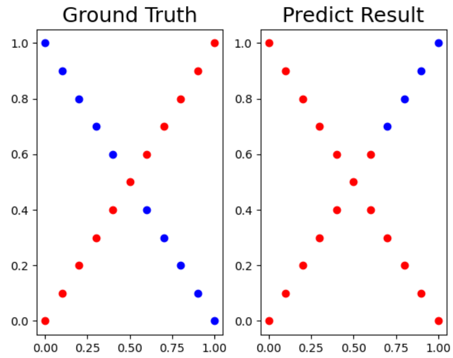
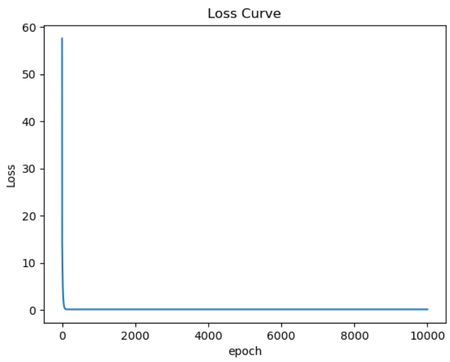
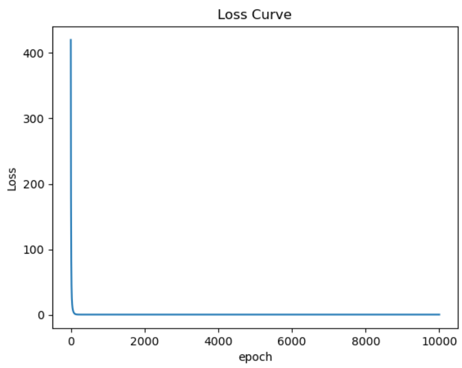
<u>Conclusion:</u>

The appropriate combination of learning rate and hidden layer units is necessary to achieve good training performance for the network. In the linear case with 128 hidden units, the learning rate is too small for the optimizer to escape from the local minimum. If we set larger learning rate, the problem can be solved.

C. Try without activation functions

In this section, due to the possibility of overflow without activation functions, we consider an alternative setting.

- Learning Rate = 5e-6
- Hidden layer size = 16
- Max Epochs = 10000

<table>
<tr><th>Linear(n=100)</th><th>XOR</th></tr>
<tr><td></td><td></td></tr>
<tr><td>Loss = 0.34084
Accuracy = 80.0%</td><td>Loss = 0.49494
Accuracy = 33.3%</td></tr>
<tr><td></td><td></td></tr>
</table>

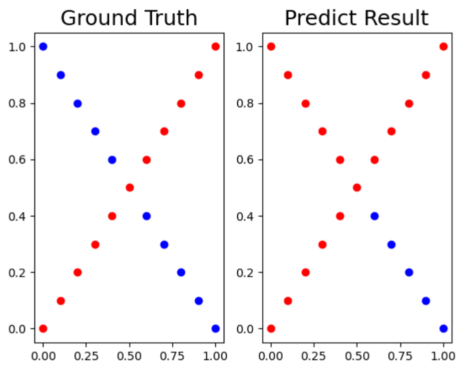
<u>Conclusion:</u>

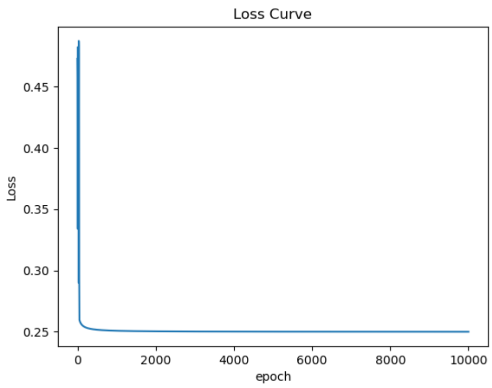
We can observe that without an activation function, the performance of the network is poor. The reason for this is that without an activation function, the network can only handle non-linear cases.

D. Anything you want to share

In the part of 4-C, we can observe that without an activation function, the network cannot handle the non-linear case. So, we add a sigmoid activation function to the output layer to see if it can handle non-linear case.

```
Loss = 0.26429, accuracy = 76.2%
```

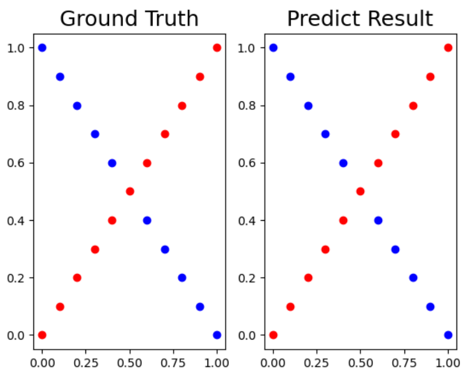


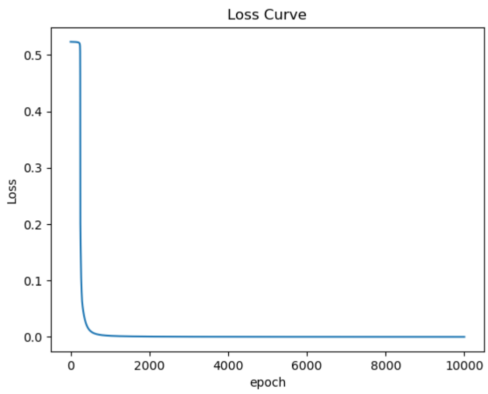


From the results, it seems that it still cannot handle non-linear cases.

Next, we add a sigmoid activation function to the last two layer to see if it can handle non-linear case.

```
Loss = 0.00808, accuracy = 100.0%
```





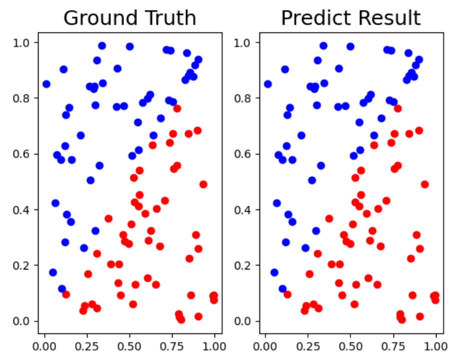
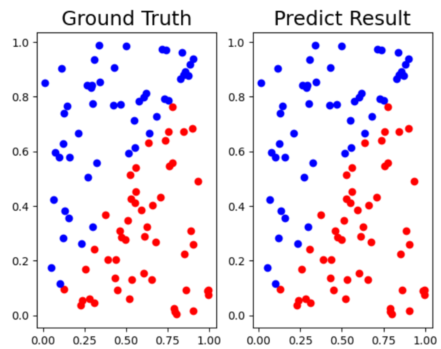
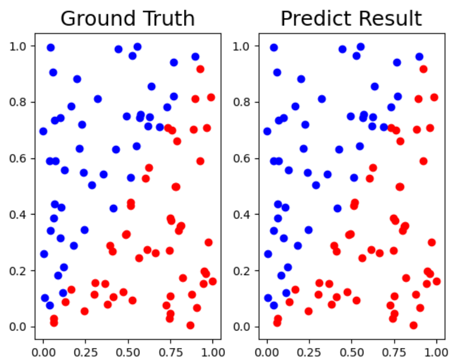
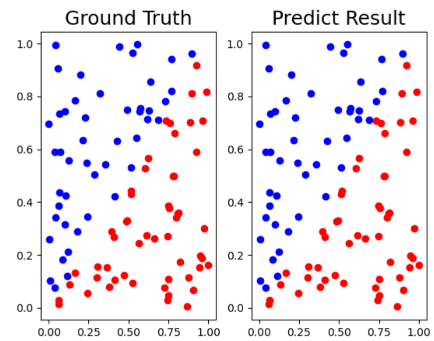
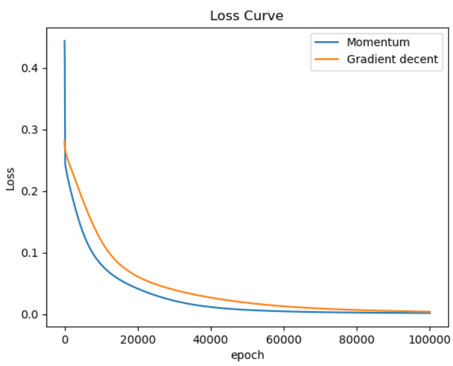
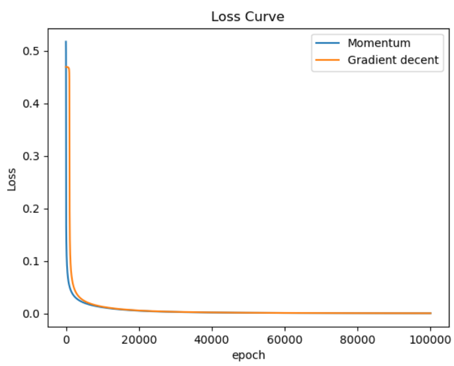
From the results, it seems that in this case, it can handle non-linear case.

5. Extra (10%)

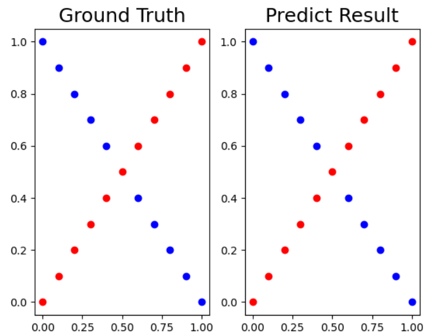
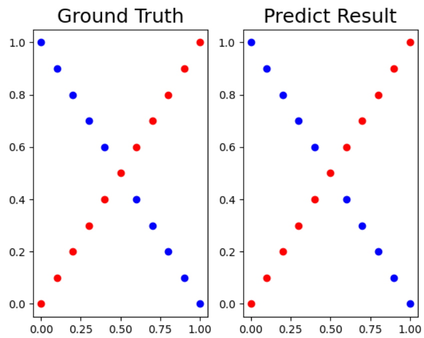
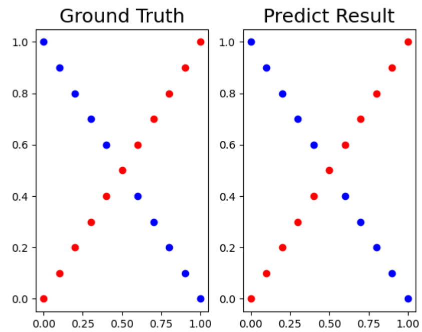
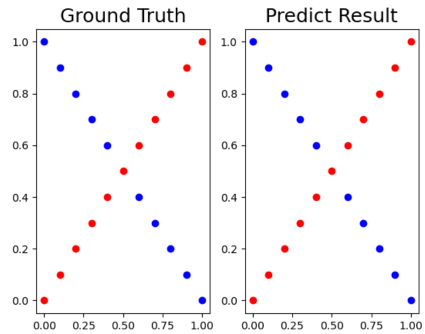
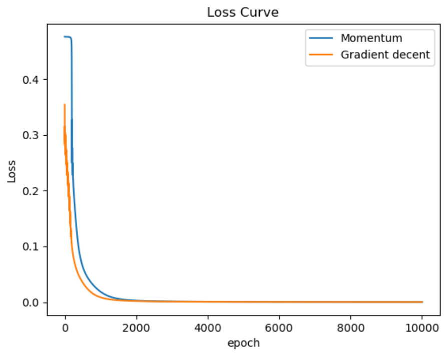
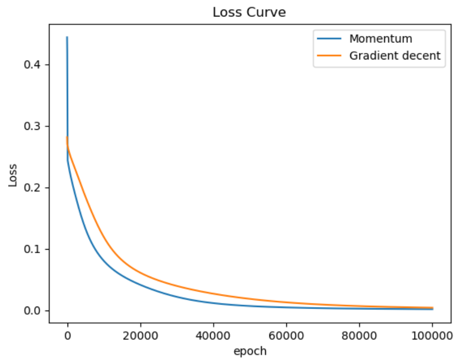
A. Implement different optimizers. (2%)

In this part, we implement the another optimizer momentum.

Linear case:

Hidden layer size = 32 Max Epochs = 10000 Learning rate=5e-2	Hidden layer size = 32 Max Epochs = 100000 Learning rate=1e-3
Gradient decent:  Momentum:  	Gradient decent:  Momentum: 
 	

XOR case:

<table>
<tr>
<td>Hidden layer size = 32
Max Epochs = 10000
Learning rate=5e-2</td>
<td>Hidden layer size = 32
Max Epochs = 100000
Learning rate=1e-3</td>
</tr>
<tr>
<td>Gradient decent:

Momentum:
</td>
<td>Gradient decent:

Momentum:
</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</table>

Conclusion:

In this part, we can observe that in some cases, using the Momentum optimizer can accelerate the convergence of the Loss.

B. Implement different activation functions. (3%)

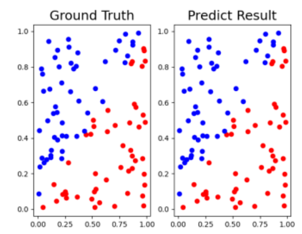
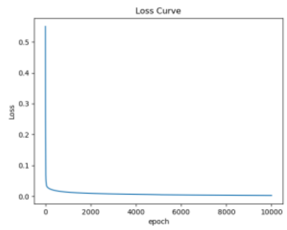
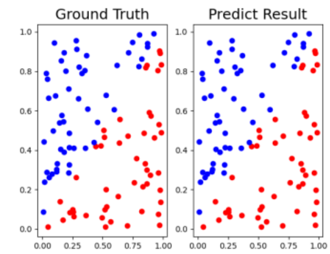
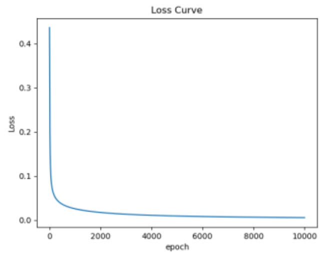
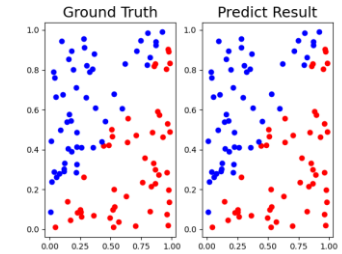
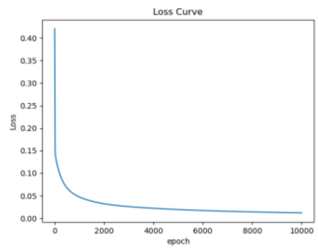
In this part, we consider the following hyperparameter setting:

- Hidden layer size = 32
- Max Epochs = 10000
- Learning rate=1e-3
- Optimizer: Gradient decent

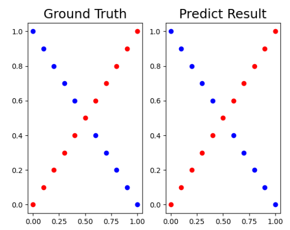
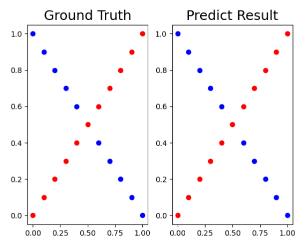
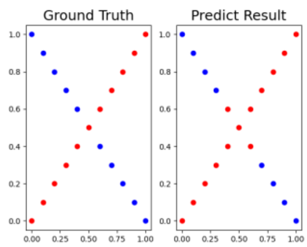
And we consider different combinations of activation functions.

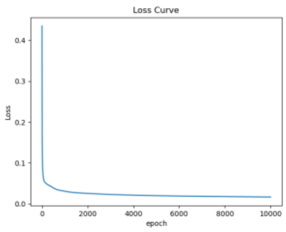
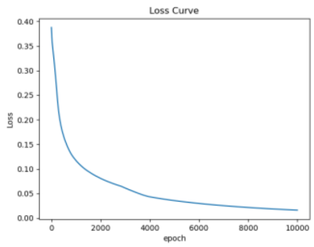
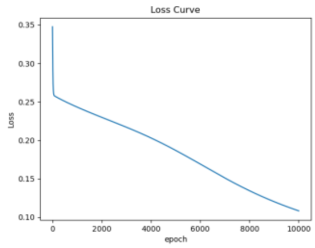
(Note: The i-th activation function in the list corresponds to the activation function used in the i-th layer.)

Linear case:

activation functions= [‘ReLU’, ‘ReLU’,’Sigmoid’]	activation functions= [‘ReLU’, ‘Sigmoid’,’Sigmoid’]	activation functions= [‘Sigmoid’, ‘Sigmoid’,’Sigmoid’]
 	 	 
Loss = 0.01474 Accuracy = 100.0%	Loss = 0.02490 Accuracy = 100.0%	Loss = 0.04699 Accuracy = 100.0%

XOR case:

activation functions= [‘ReLU’, ‘ReLU’,’Sigmoid’]	activation functions= [‘ReLU’, ‘Sigmoid’,’Sigmoid’]	activation functions= [‘Sigmoid’, ‘Sigmoid’,’Sigmoid’]
		

<table>
<tr>
<td>
</td>
<td>
</td>
<td>
</td>
</tr>
<tr>
<td>Loss = 0.05380
Accuracy = 100.0%</td>
<td>Loss = 0.09525
Accuracy = 100.0%</td>
<td>Loss = 0.30610
Accuracy = 90.5%</td>
</tr>
</table>

Conclusion:

In this experiment and in Part 3, we can observe that using more 'ReLU' activation functions converges with smaller learning rates. Conversely, with more 'sigmoid' activation functions, larger learning rates are needed for convergence