NYCU DL Lab1 Backpropagation

313551073 顏琦恩

I. Introduction

A simple neural network with two hidden layers is implemented in this lab. The structure of the neural network is presented in the following picture.

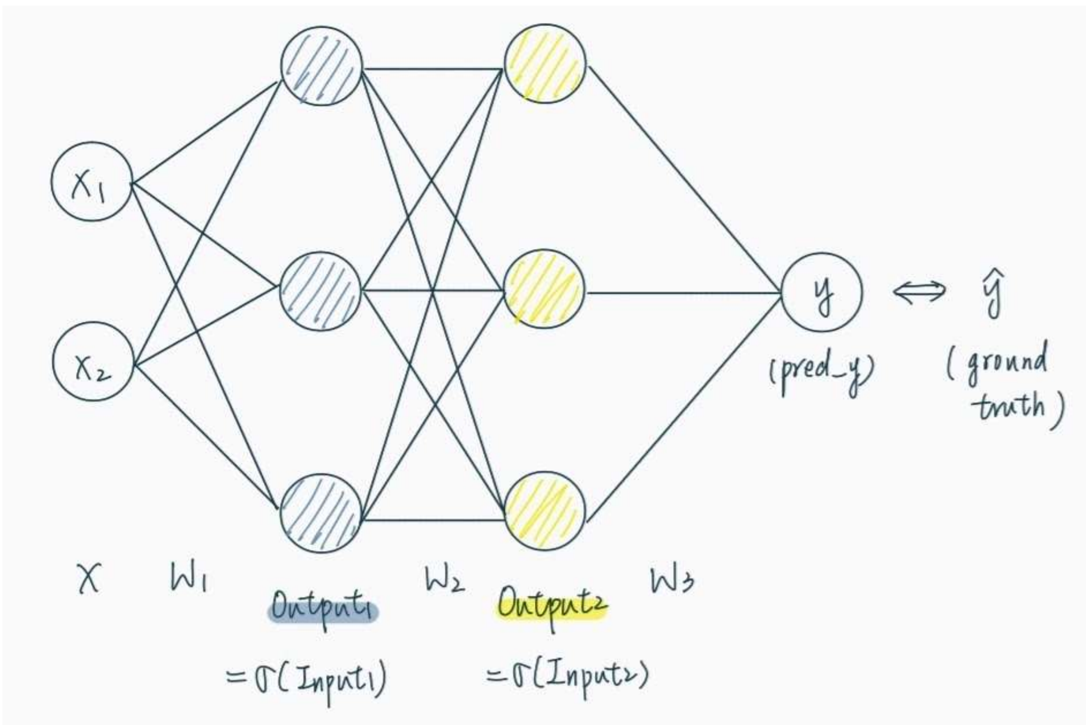


Figure 1. The structure of my neural network

In the neural network, there will be three nodes for each hidden layer, and the activation function I chose is the sigmoid function.

There are two main classes in my code, one is the Layer class, and the other is the NeuralNetwork class. The Layer class consists of inputs, outputs, weights, and gradients. Inputs stand for the values before going through the activation function, while outputs are the values after. The NeuralNetwork class does the main work of this lab, such as forward/backward processing, training, and testing. Activation functions are also defined in this section.

To train the neural network, I use the dataset provided by TA, which includes linear and XOR data. Also, I adjust hyperparameters such as the learning rate by myself to get a better result.

II. Experiment setups

A. Sigmoid functions

Sigmoid function is an activation function usually used in forward propagation. It is used to control input values in the neural network between 0 and 1, which makes it useful for binary classification and logistic regression problems. The function and its derivative is defined in the class NeuralNetwork. The following figures show the definition of these two functions.

```python
def sigmoid(self, x):
    return 1.0/(1.0 + np.exp(-x))
```

```python
def derivative_sigmoid(self, x):
    return np.multiply(x, 1.0-x)
```

Figure 2. Definition of sigmoid and its derivative function

B. Neural network

To create a neural network model, one should pass three parameters to the NeuralNetwork class: layer definition, activation function, and learning rate (Fig. 3). Layer definition is a list, which presents how many nodes in each network layer. There are three choices of activation function: sigmoid, relu, and none, one can choose their activation function by adding this parameter, the default value is sigmoid.

```python
myNN = NeuralNetwork([2,3,3,1], 'sigmoid', 0.1)
```

Figure 3. Example of constructing a neural network model

C. Backpropagation

Backpropagation is the main part of training a neural network. I use the vectorized formula to calculate the gradients of each layer. The following figures show the formula and how it is implemented.

$$\delta^{[l]} = (\delta^{[l+1]} W^{[l+1]T}) \odot \text{activation\_derivative}(Z^{[l]})$$

$$\nabla W^{[l]} = A^{[l-1]T} \delta^{[l]}$$

Figure 4. Formula of backpropagation

```python
def backward_pass(self, error):
    self.layers[-1].gradients = error * self.deriv(self.layers[-1].outputs)

    for i in range(self.num_of_layers - 2, -1, -1):
        delta = self.layers[i + 1].gradients.dot(self.layers[i + 1].weights.T) * self.deriv(self.layers[i].outputs)
        self.layers[i].gradients = delta

def update(self):
    for i in range(self.num_of_layers):
        if i == 0:
            self.layers[i].weights -= self.learning_rate * self.x.T.dot(self.layers[i].gradients)
        else:
            self.layers[i].weights -= self.learning_rate * self.layers[i-1].outputs.T.dot(self.layers[i].gradients)
```

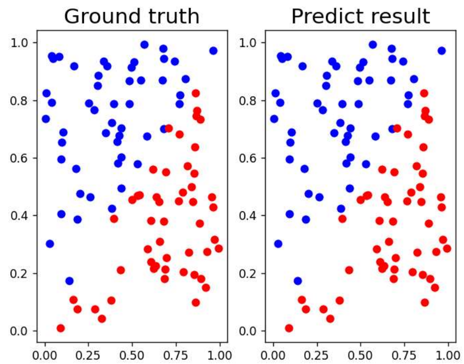
Figure 5. Code of backpropagation

III. Results of your testing

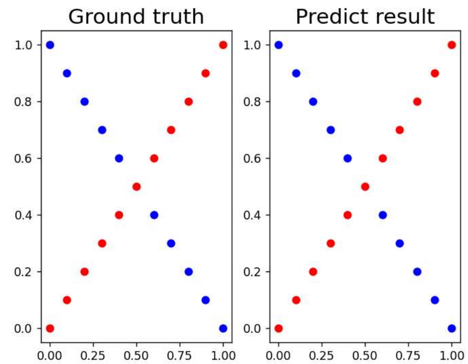
In the following section, the learning rate was 0.1 and trained 9000 epochs.

A. Screenshot and comparison figure

We can see that the model perfectly classified all data.



Linear data



XOR data

B. Accuracy of prediction

Below is the loss and prediction of the training section. We can see that linear data's loss converges faster than XOR data.

```
epoch    0 loss : 0.26809808043502625
epoch  500 loss : 0.010990509556571368
epoch 1000 loss : 0.006924316146301311
epoch 1500 loss : 0.00510938807987692
epoch 2000 loss : 0.0037965038479489837
epoch 2500 loss : 0.002705804677818774
epoch 3000 loss : 0.001863282338370656
epoch 3500 loss : 0.0012937984649232653
epoch 4000 loss : 0.000926989793534618
epoch 4500 loss : 0.00069120613725886
epoch 5000 loss : 0.0005365216236058433
epoch 5500 loss : 0.0004312741210151503
epoch 6000 loss : 0.00035671688979865444
epoch 6500 loss : 0.0003018978161577309
epoch 7000 loss : 0.0002602754362498412
epoch 7500 loss : 0.00022780324752457237
epoch 8000 loss : 0.00020188368029330027
epoch 8500 loss : 0.0001807901870022966
```

Linear data

```
epoch    0 loss : 0.33866461936390924
epoch  500 loss : 0.2516141310992486
epoch 1000 loss : 0.2512783990023108
epoch 1500 loss : 0.250310739336257
epoch 2000 loss : 0.23633446839273486
epoch 2500 loss : 0.11426765448966551
epoch 3000 loss : 0.023901845413036363
epoch 3500 loss : 0.007136076377756459
epoch 4000 loss : 0.0035004327636808425
epoch 4500 loss : 0.002195634249936492
epoch 5000 loss : 0.001563985758852134
epoch 5500 loss : 0.0012009579024429898
epoch 6000 loss : 0.0009684301359010101
epoch 6500 loss : 0.0008080552787914855
epoch 7000 loss : 0.0006913699426001366
epoch 7500 loss : 0.0006029777914081538
epoch 8000 loss : 0.0005338784899620143
epoch 8500 loss : 0.00047848303365337716
```

XOR data

```
[[9.99988297e-01]
 [9.99988583e-01]
 [1.49384417e-05]
 [9.99984300e-01]
 [2.54139186e-07]
 [1.28851786e-07]
 [9.99989345e-01]
 [1.28561886e-07]
 [9.99989218e-01]
 [1.75555249e-07]
 [1.26038278e-07]
 [1.21984113e-06]
 [1.23719483e-07]
 [9.99988926e-01]
 [9.99561111e-01]
 [9.61222565e-01]
 [1.18226664e-07]
 [1.37045518e-07]
 [2.70245616e-04]
 [9.99988014e-01]
 [9.99989302e-01]
```

Linear data

```
[[0.02722275]
 [0.99148221]
 [0.02489247]
 [0.99152008]
 [0.02287383]
 [0.99140158]
 [0.02112415]
 [0.98994218]
 [0.01960719]
 [0.95145282]
 [0.0182922 ]
 [0.01715316]
 [0.96203453]
 [0.01616801]
 [0.98514665]
 [0.01531813]
 [0.98645042]
 [0.01458777]
 [0.98714724]
 [0.01396368]
 [0.98750696]]
```

XOR data

Below is the loss and predictions of the testing section. We can see that the predictions of the model are very close to the ground truth, and can also see that it has a better performance on linear problems than XOR problems.

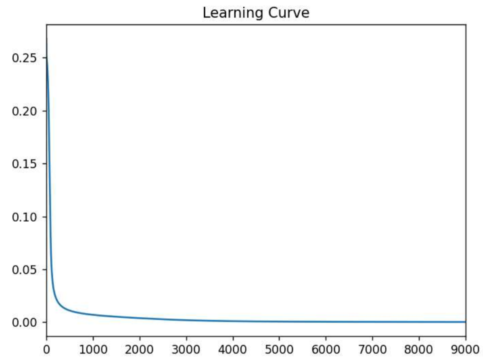
```
| Iter78 | Ground truth: 1.0 | prediction: 0.99999 |
| Iter79 | Ground truth: 1.0 | prediction: 0.99992 |
| Iter80 | Ground truth: 0.0 | prediction: 0.00000 |
| Iter81 | Ground truth: 0.0 | prediction: 0.00000 |
| Iter82 | Ground truth: 0.0 | prediction: 0.00000 |
| Iter83 | Ground truth: 0.0 | prediction: 0.00000 |
| Iter84 | Ground truth: 1.0 | prediction: 0.99999 |
| Iter85 | Ground truth: 1.0 | prediction: 0.99999 |
| Iter86 | Ground truth: 1.0 | prediction: 0.99698 |
| Iter87 | Ground truth: 0.0 | prediction: 0.00004 |
| Iter88 | Ground truth: 0.0 | prediction: 0.06923 |
| Iter89 | Ground truth: 0.0 | prediction: 0.00000 |
| Iter90 | Ground truth: 0.0 | prediction: 0.00000 |
| Iter91 | Ground truth: 1.0 | prediction: 0.99995 |
| Iter92 | Ground truth: 1.0 | prediction: 0.99999 |
| Iter93 | Ground truth: 1.0 | prediction: 0.99999 |
| Iter94 | Ground truth: 0.0 | prediction: 0.00001 |
| Iter95 | Ground truth: 1.0 | prediction: 0.99999 |
| Iter96 | Ground truth: 1.0 | prediction: 0.99999 |
| Iter97 | Ground truth: 0.0 | prediction: 0.00000 |
| Iter98 | Ground truth: 1.0 | prediction: 0.99998 |
| Iter99 | Ground truth: 0.0 | prediction: 0.00000 |
loss=0.00014 accuracy=100.00%
```

```
| Iter 0 | Ground truth: 0.0 | prediction: 0.02722 |
| Iter 1 | Ground truth: 1.0 | prediction: 0.99148 |
| Iter 2 | Ground truth: 0.0 | prediction: 0.02489 |
| Iter 3 | Ground truth: 1.0 | prediction: 0.99152 |
| Iter 4 | Ground truth: 0.0 | prediction: 0.02287 |
| Iter 5 | Ground truth: 1.0 | prediction: 0.99140 |
| Iter 6 | Ground truth: 0.0 | prediction: 0.02112 |
| Iter 7 | Ground truth: 1.0 | prediction: 0.98994 |
| Iter 8 | Ground truth: 0.0 | prediction: 0.01961 |
| Iter 9 | Ground truth: 1.0 | prediction: 0.95145 |
| Iter10 | Ground truth: 0.0 | prediction: 0.01829 |
| Iter11 | Ground truth: 0.0 | prediction: 0.01715 |
| Iter12 | Ground truth: 1.0 | prediction: 0.96203 |
| Iter13 | Ground truth: 0.0 | prediction: 0.01617 |
| Iter14 | Ground truth: 1.0 | prediction: 0.98515 |
| Iter15 | Ground truth: 0.0 | prediction: 0.01532 |
| Iter16 | Ground truth: 1.0 | prediction: 0.98645 |
| Iter17 | Ground truth: 0.0 | prediction: 0.01459 |
| Iter18 | Ground truth: 1.0 | prediction: 0.98715 |
| Iter19 | Ground truth: 0.0 | prediction: 0.01396 |
| Iter20 | Ground truth: 1.0 | prediction: 0.98751 |
loss=0.00043 accuracy=100.00%
```

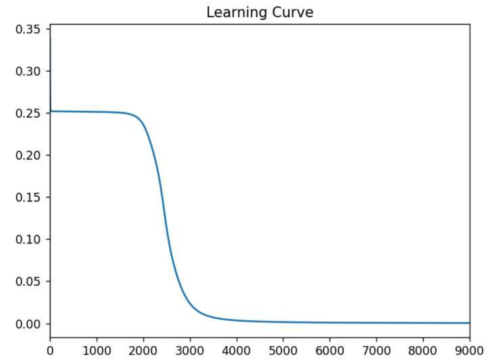
Linear data **XOR data**

C. Learning curve

We can see the same phenomenon with the training section: the model performs better in linear problems.



Linear data



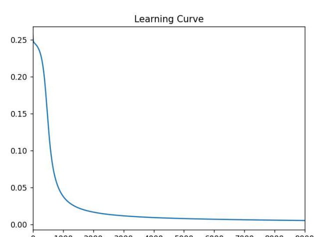
XOR data

IV. Discussion

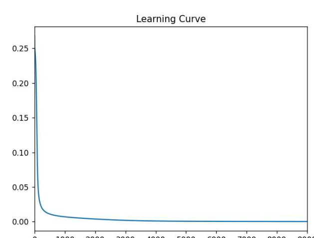
A. Different learning rates

Learning rate	Accuracy of linear data	Accuracy of XOR data
0.01	100%	80.95%
0.1	100%	100%
1	100%	100%

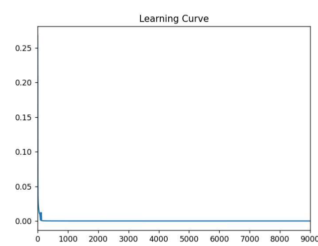
Learning curve of linear data:



Learning rate = 0.01

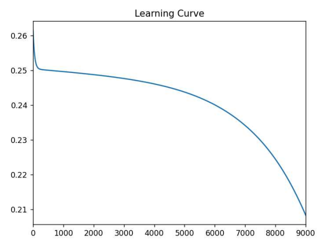
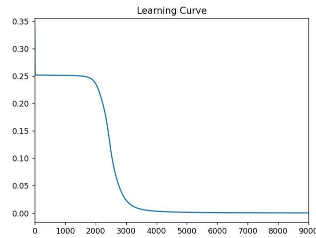
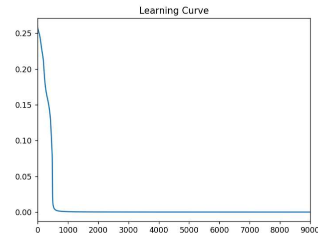


Learning rate = 0.1



Learning rate = 1

Learning curve of XOR data:

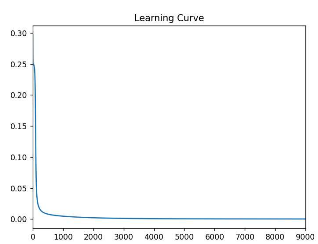
Learning rate = 0.01   Learning rate = 0.1   Learning rate = 1

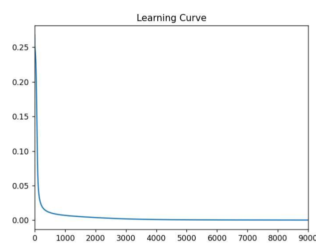
We can see that a higher learning rate leads to better performance and faster convergence in the learning curve.

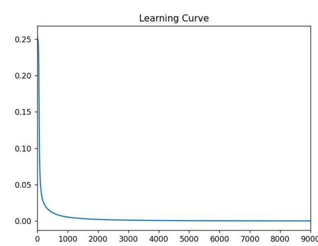
B. Different numbers of hidden units

Numbers of hidden units	Accuracy of linear data	Accuracy of XOR data
2	100%	76.19%
3	100%	100%
4	100%	100%

Learning curve of linear data:

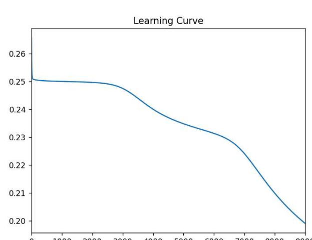


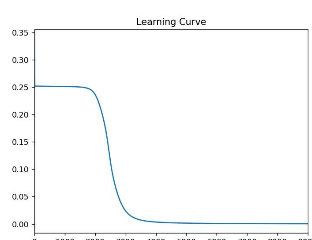


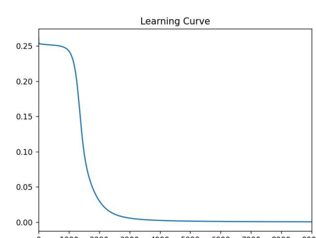


Hidden units = 2   Hidden units = 3   Hidden units = 4

Learning curve of XOR data:







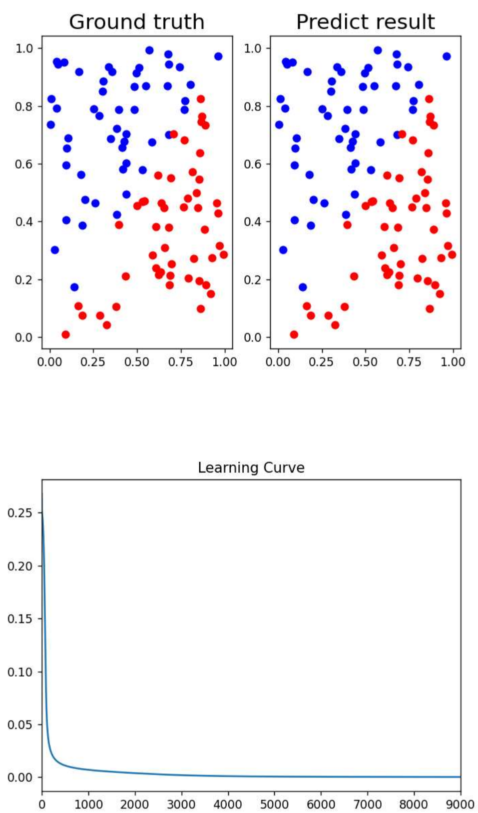
Hidden units = 2   Hidden units = 3   Hidden units = 4

We can see that the number of hidden units also affects the convergence speed of training. The more hidden units we set, the faster it converges.

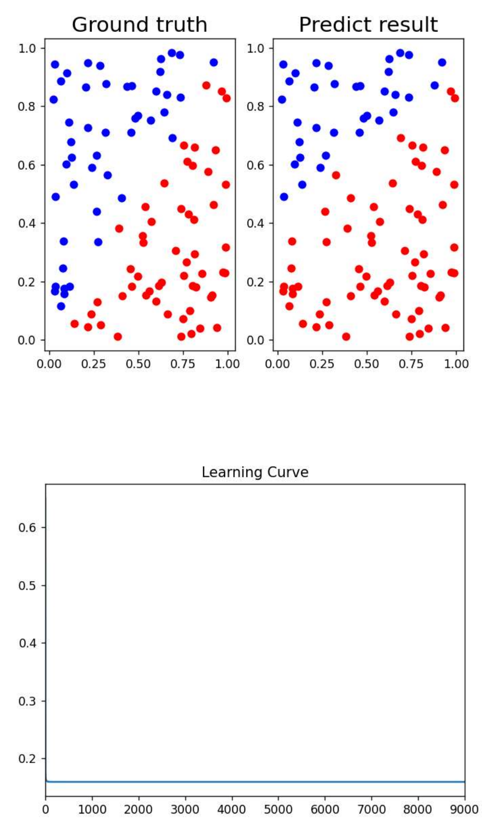
C. Without activation functions

	Accuracy of linear data	Accuracy of XOR data
With activation function (sigmoid)	100%	100%
Without activation function	86%	42.86%

Prediction and learning curve of linear data:

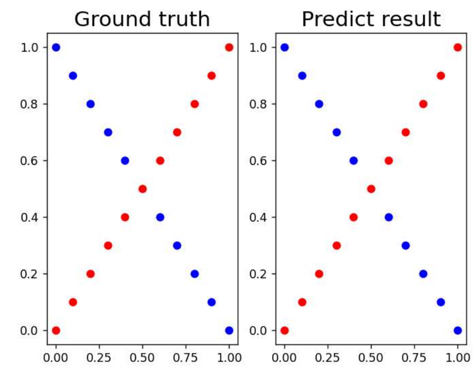


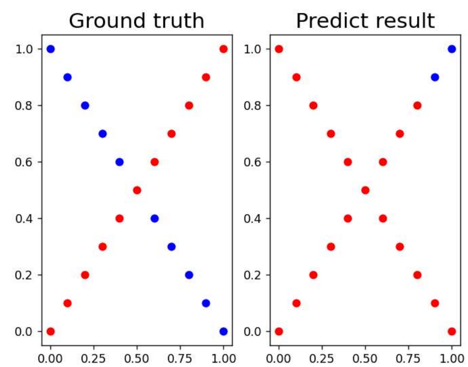
With activation function (sigmoid)

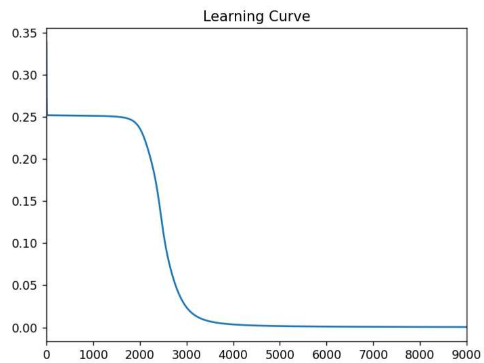


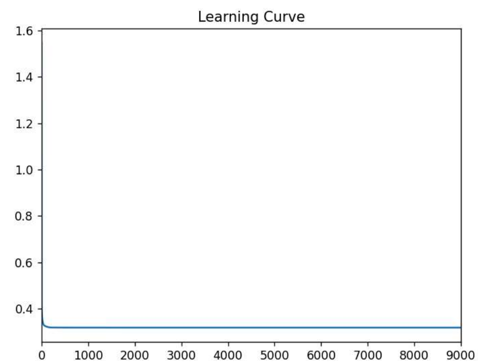
Without activation function

Prediction of XOR data:









With activation function (sigmoid) Without activation function

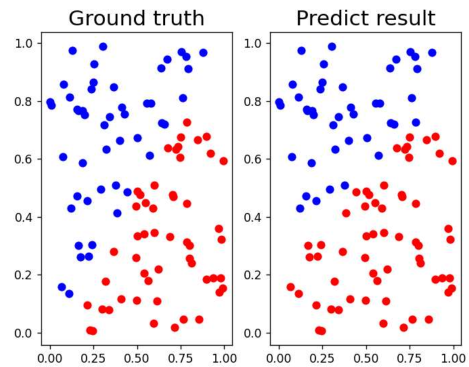
We can see that in the experiment, learning curves on the side without activation function don't converge. Although the model has a pretty good result in linear data, it performs badly in complicated problems such as XOR data.

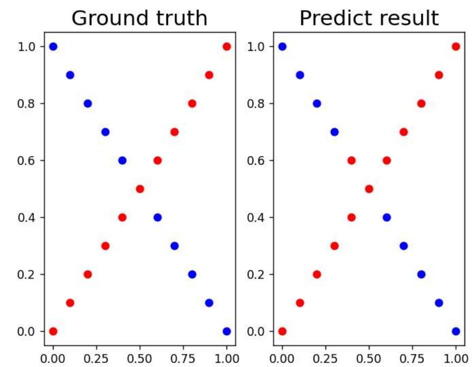
V. Extra

A. Implement different activation functions

I've implemented the relu function in my work, below are the results of linear data and XOR data.

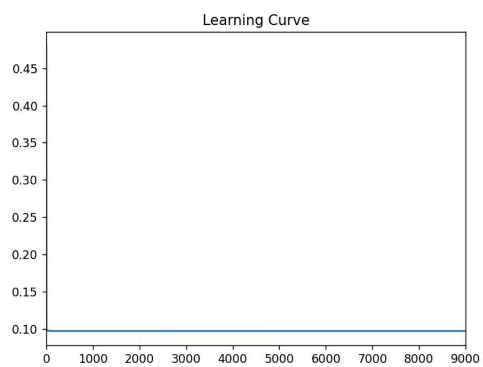
Linear **XOR**

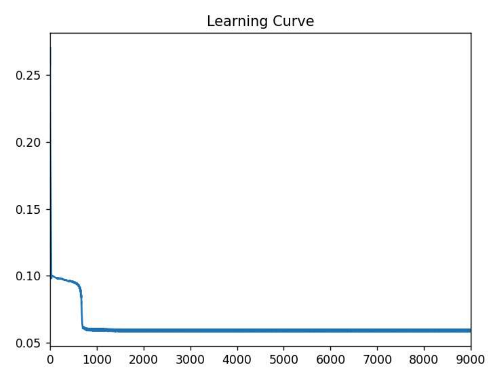





```
| Iter89 | Ground truth: 1.0 | prediction: 1.19337 |
| Iter90 | Ground truth: 1.0 | prediction: 0.67331 |
| Iter91 | Ground truth: 1.0 | prediction: 1.35995 |
| Iter92 | Ground truth: 1.0 | prediction: 0.85072 |
| Iter93 | Ground truth: 1.0 | prediction: 0.58660 |
| Iter94 | Ground truth: 0.0 | prediction: 0.29185 |
| Iter95 | Ground truth: 0.0 | prediction: 0.00000 |
| Iter96 | Ground truth: 0.0 | prediction: 0.46448 |
| Iter97 | Ground truth: 0.0 | prediction: 0.00000 |
| Iter98 | Ground truth: 1.0 | prediction: 0.84369 |
| Iter99 | Ground truth: 1.0 | prediction: 0.30044 |
loss=0.09353 accuracy=91.00%
```

```
| Iter10 | Ground truth: 0.0 | prediction: 0.02617 |
| Iter11 | Ground truth: 0.0 | prediction: 0.03141 |
| Iter12 | Ground truth: 1.0 | prediction: 0.57883 |
| Iter13 | Ground truth: 0.0 | prediction: 0.03664 |
| Iter14 | Ground truth: 1.0 | prediction: 0.72164 |
| Iter15 | Ground truth: 0.0 | prediction: 0.04188 |
| Iter16 | Ground truth: 1.0 | prediction: 0.86444 |
| Iter17 | Ground truth: 0.0 | prediction: 0.04711 |
| Iter18 | Ground truth: 1.0 | prediction: 1.00724 |
| Iter19 | Ground truth: 0.0 | prediction: 0.05235 |
| Iter20 | Ground truth: 1.0 | prediction: 1.15005 |
loss=0.05280 accuracy=95.24%
```





We can see that using relu function as activation function converges faster than using sigmoid, but the predictions of using relu are worse than using sigmoid.

I also observed that using relu may sometimes cause the vanishing gradient problem, the following figure shows the phenomenon of gradient vanishing.

```
epoch 5000 loss : 0.47619047619047616
epoch 5500 loss : 0.47619047619047616
epoch 6000 loss : 0.47619047619047616
epoch 6500 loss : 0.47619047619047616
epoch 7000 loss : 0.47619047619047616
epoch 7500 loss : 0.47619047619047616
epoch 8000 loss : 0.47619047619047616
epoch 8500 loss : 0.47619047619047616
```

Figure 6. Gradient vanishing problem