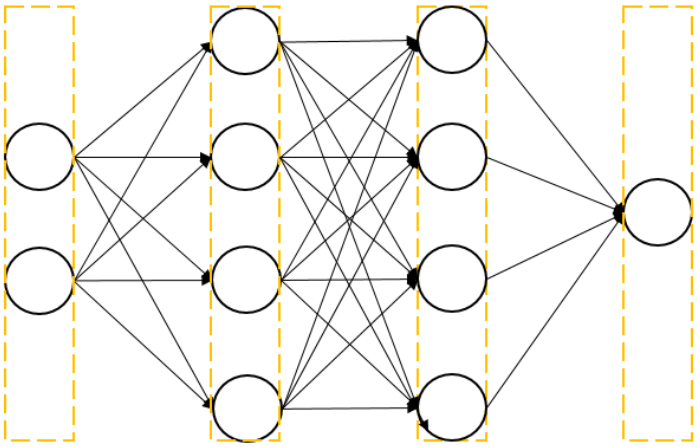
Lab 2: back-propagation

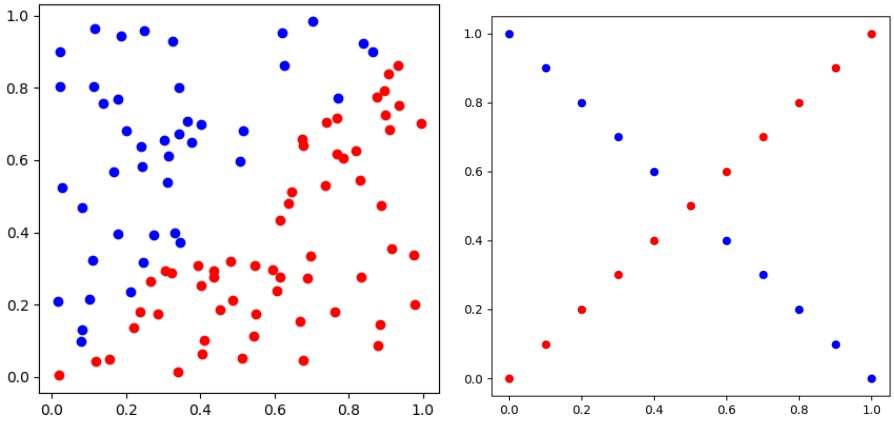
學號: 310551083 姓名: 許嘉倫

1. Introduction

In this lab, we will need to implement a simple neural network with forwarding pass and back-propagation. And we can only use Numpy and python standard libraries, any other frameworks like Tensorflow and Pytorch are not allowed to use in this lab.



Our work is to classify the dataset provided by TA. The dataset include linear data and XOR data.

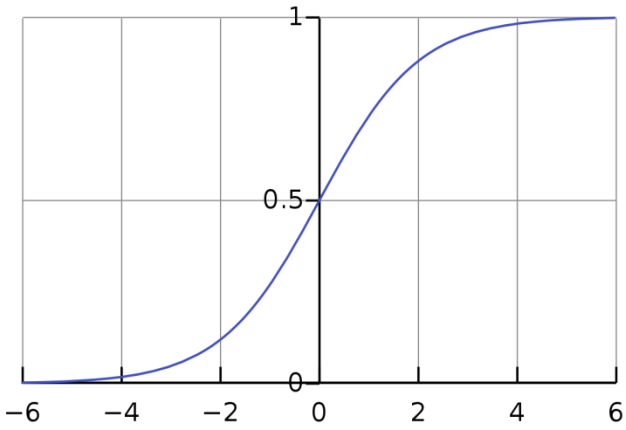


2. Experiment setups

A. Sigmoid functions

In this lab, I use sigmoid functions as my activation, and it can make the value between 0 and 1. The formula is shown below:

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



Sigmoid function

The derivative of sigmoid function is shown below:

$$\frac{d\sigma(x)}{dx} = \frac{d}{dx}(1+e^{-x})^{-1}$$
$$= -(1+e^{-x})^{-2}(-e^{-x})$$
$$= \frac{e^{-x}}{(1+e^{-x})^2}$$
$$= \frac{1}{(1+e^{-x})}\frac{e^{-x}}{(1+e^{-x})}$$
$$= \frac{1}{(1+e^{-x})}\frac{1+e^{-x}-1}{(1+e^{-x})}$$
$$= \sigma(x)(1-\sigma(x))$$

The code is provided by TA

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

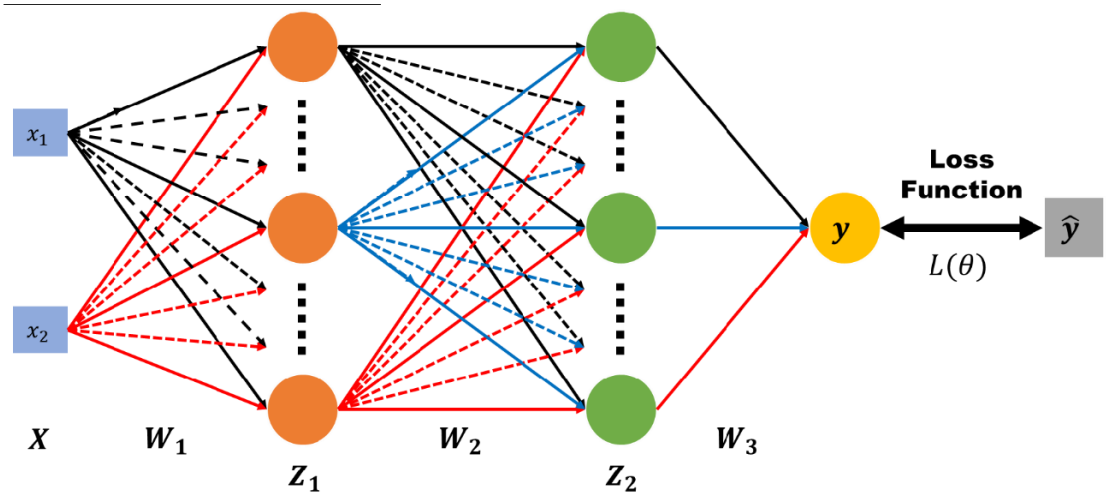
```
def derivative_sigmoid(x):
    return np.multiply(x, 1.0 - x)
```

B. Neural network

I implement a simple neural network with 2 hidden layers and each hidden layer has 10 neuron units. And I choose MSE (mean square error) as my loss function.

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - pred\_y_i)^2$$

$$\frac{d\ \text{MSE}(y_i,\ \text{pred\_}y_i)}{d\ pred\_y_i} = \frac{-2(y_i - pred\_y_i)}{N}$$

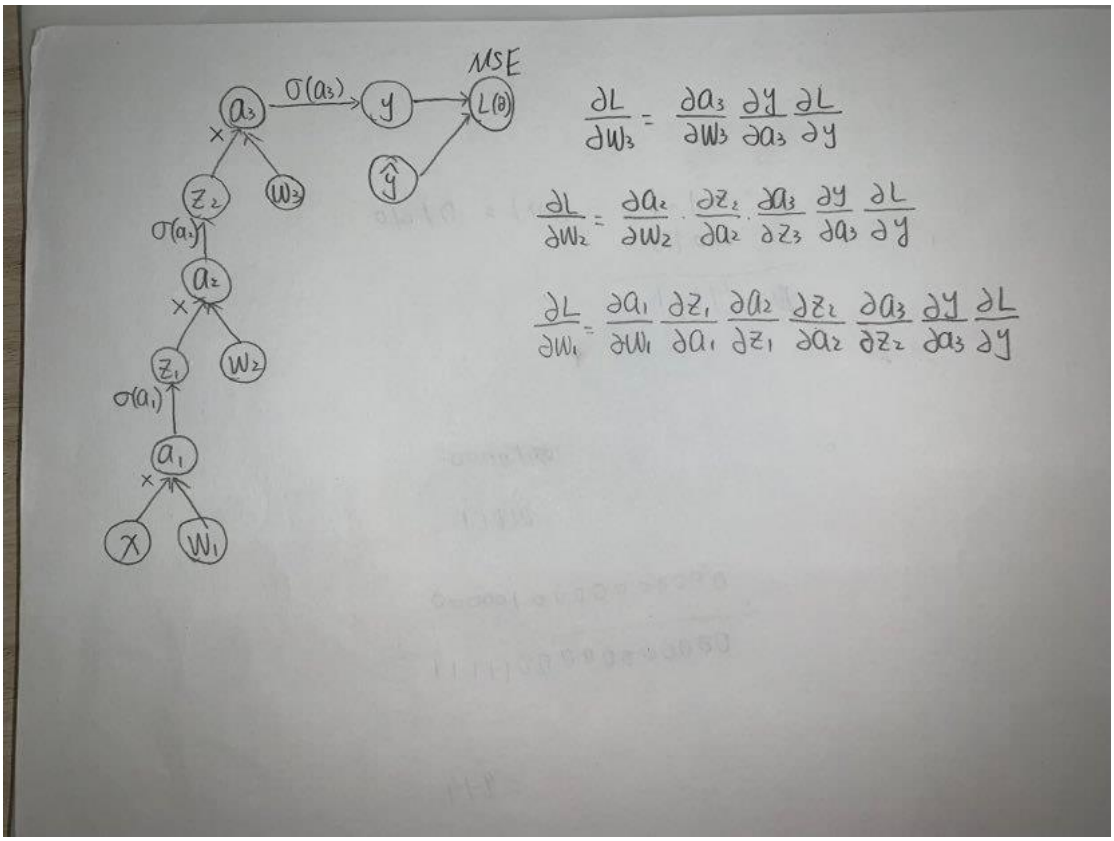


The forwarding pass of the network is shown below:

$\mathrm{Z}_1 = \sigma(\mathrm{X}\mathrm{W}_1), \quad \mathrm{Z}_2 = \sigma(\mathrm{Z}_1\mathrm{W}_2), \quad \mathrm{y} = \sigma(\mathrm{Z}_2 W_3)$

C. Backpropagation

After get the value of loss function, we need to use back-propagation to update our model weights. And we should use chain rule to compute $\frac{dL}{dW_1}$, $\frac{dL}{dW_2}$, $\frac{dL}{dW_3}$. The picture below is my computation graph.



After computing the gradient of weights, we can update the model weights (lr means learning rate).

$W_1 = W_1 - lr * \nabla W_1$

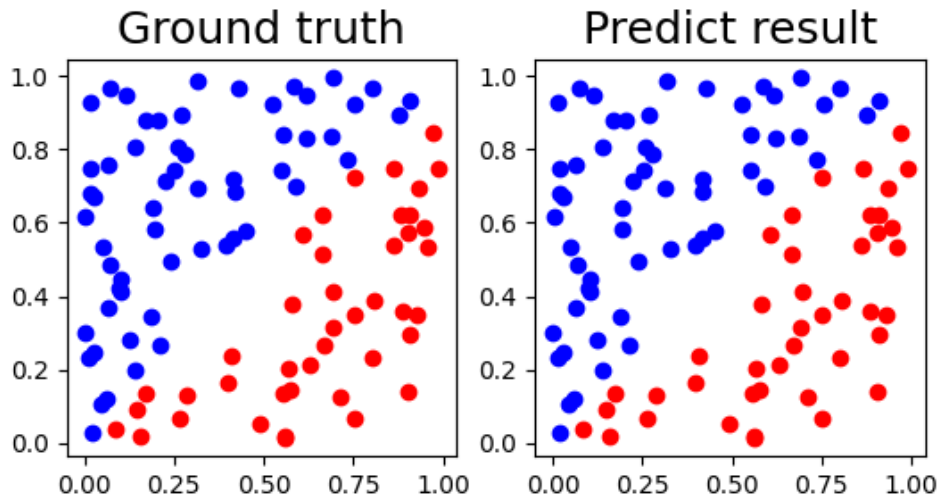
$W_2 = W_2 - lr * \nabla W_2$

$W_3 = W_3 - lr * \nabla W_3$

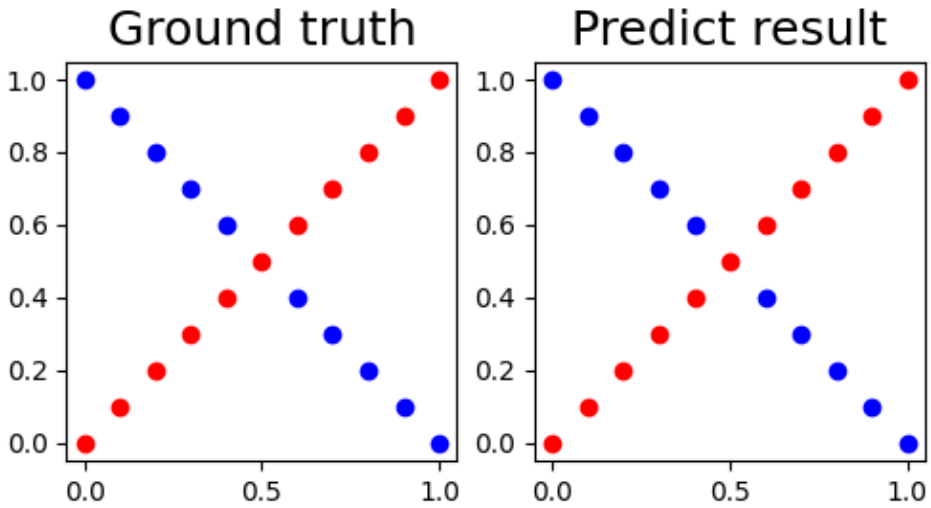
3. Results of my testing

A. Screenshot and comparison figure

Linear data



XOR data



B. Show the accuracy of my prediction

Linear data

epoch 500 loss : 0.020983361997710483

epoch 1000 loss : 0.011927374435777413

epoch 1500 loss : 0.008004625370048628

epoch 2000 loss : 0.005702100897995381

epoch 2500 loss : 0.004226482856125826
epoch 3000 loss : 0.0032371213506189227
epoch 3500 loss : 0.0025515926212312267
epoch 4000 loss : 0.0020625173508259384
epoch 4500 loss : 0.0017039724808783968
epoch 5000 loss : 0.001434430879678489
epoch 5500 loss : 0.001227122992498512
epoch 6000 loss : 0.0010643856035135683
epoch 6500 loss : 0.0009342891917579352
epoch 7000 loss : 0.0008285922490891021
epoch 7500 loss : 0.000741477640752342
epoch 8000 loss : 0.0006687564018653134
epoch 8500 loss : 0.0006073556915103063
epoch 9000 loss : 0.0005549827433047098
epoch 9500 loss : 0.0005098999520882248
epoch 10000 loss : 0.00047077150660527025
[[9.99992695e-01]
 [9.99994545e-01]
 [1.92082778e-02]
 [9.99908598e-01]
 [9.99746033e-01]
 [9.99988362e-01]
 [9.99998428e-01]
 [9.99998363e-01]
 [9.99739442e-01]
 [9.90490785e-01]
 [9.99942259e-01]
 [9.99996168e-01]
 [9.06410135e-01]
 [6.48546872e-02]
 [9.99998493e-01]
 [9.99994980e-01]
 [9.08588433e-06]
 [1.00966315e-05]
 [9.99865325e-01]
 [9.99998516e-01]
 [3.54939484e-05]
 [9.99997827e-01]

[9.99995714e-01]
[6.34089635e-06]
[1.09023969e-04]
[9.99998395e-01]
[9.78244952e-01]
[8.16748388e-06]
[8.40265177e-06]
[9.99171382e-01]
[9.99994480e-01]
[1.78495971e-05]
[9.99997769e-01]
[1.44349933e-04]
[1.19538613e-05]
[5.74094790e-06]
[9.99998425e-01]
[2.68169675e-05]
[6.59500567e-06]
[1.22786720e-01]
[2.49616770e-05]
[9.30720508e-06]
[9.99998309e-01]
[9.99997964e-01]
[2.48674224e-02]
[9.99998642e-01]
[9.99555322e-01]
[7.04187895e-05]
[1.38295601e-05]
[9.99998283e-01]
[9.99961133e-01]
[7.05064647e-06]
[1.95868758e-05]
[9.99998687e-01]
[9.99993746e-01]
[5.54712743e-02]
[3.57064693e-05]
[9.99998463e-01]
[9.99939419e-01]
[6.46164438e-06]

[9.07084658e-01]
[9.99985339e-01]
[1.42998381e-05]
[9.99991791e-01]
[9.99918357e-01]
[9.99996371e-01]
[6.78552249e-06]
[6.63109249e-06]
[8.90968034e-06]
[7.37866177e-04]
[8.56925220e-06]
[9.96596711e-01]
[9.97347583e-01]
[9.99997707e-01]
[6.74552612e-06]
[9.99998685e-01]
[1.03429426e-05]
[9.99997724e-01]
[9.99996499e-01]
[7.80949087e-04]
[9.99987339e-01]
[9.99971918e-01]
[3.10826820e-04]
[9.99997144e-01]
[2.69133476e-02]
[9.99998407e-01]
[4.36907250e-05]
[9.99957277e-01]
[9.08995846e-06]
[9.99994295e-01]
[8.62979152e-06]
[5.97913748e-06]
[9.99998025e-01]
[9.99997371e-01]
[9.99996637e-01]
[5.10169709e-05]
[9.51129697e-01]
[9.48686405e-01]

[9.99986343e-01]
[9.99994830e-01]]
Acc: 100.0%

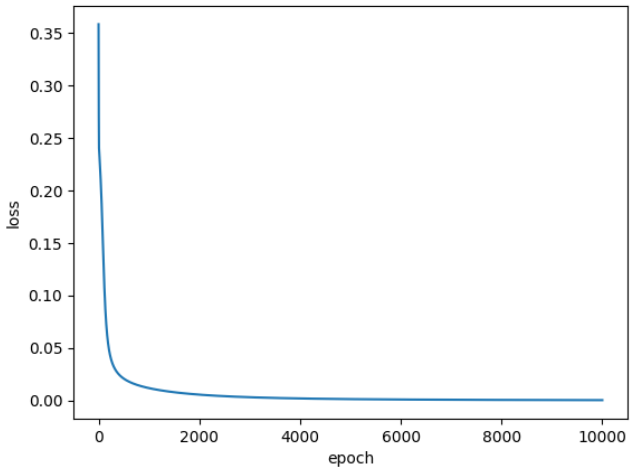
XOR data

epoch 500 loss : 0.21968544716284363
epoch 1000 loss : 0.040234486222964536
epoch 1500 loss : 0.00827349315506516
epoch 2000 loss : 0.003068285849365246
epoch 2500 loss : 0.0016906176681789921
epoch 3000 loss : 0.0011204322503488985
epoch 3500 loss : 0.0008214955859323218
epoch 4000 loss : 0.0006412384135460388
epoch 4500 loss : 0.0005221421337764315
epoch 5000 loss : 0.00043824783412203713
epoch 5500 loss : 0.0003762890913277403
epoch 6000 loss : 0.00032884054454470187
epoch 6500 loss : 0.0002914474864214833
epoch 7000 loss : 0.0002612871891039821
epoch 7500 loss : 0.00023649021358282098
epoch 8000 loss : 0.00021577250377642995
epoch 8500 loss : 0.00019822470066520776
epoch 9000 loss : 0.00018318583202013003
epoch 9500 loss : 0.00017016458612701074
epoch 10000 loss : 0.00015878857815176538
[[0.00574457]
[0.99979648]
[0.00800135]
[0.99977364]
[0.01090054]
[0.99970842]
[0.01376059]
[0.99934431]
[0.01564438]
[0.97170316]
[0.01606038]
[0.01521639]
[0.97070673]

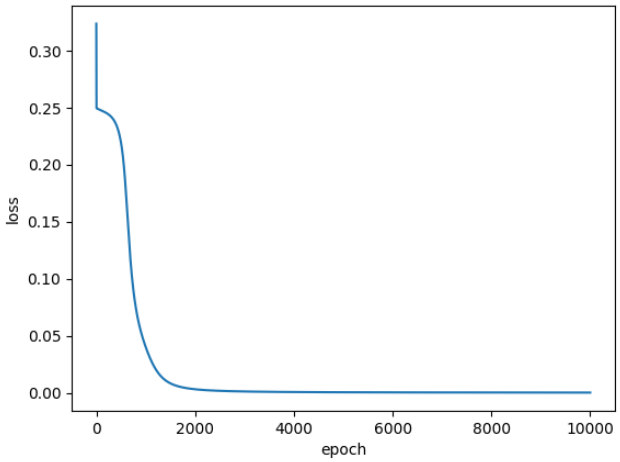
[0.01366397]
[0.99707821]
[0.01189886]
[0.99753519]
[0.01021716]
[0.99759439]
[0.00874532]
[0.99760769]]
Acc: 100.0%

C. Learning curve

Linear data



XOR data



4. Discussion

A. Try different learning rates

Accuracy of learning rate

Learning rate	Linear data	XOR data
1	100%	100%
0.1	100%	100%
0.01	93%	38%

As we can see, when learning rate decreasing to 0.01, the accuracy of linear data model and XOR data model also decrease.

B. Try different numbers of hidden units

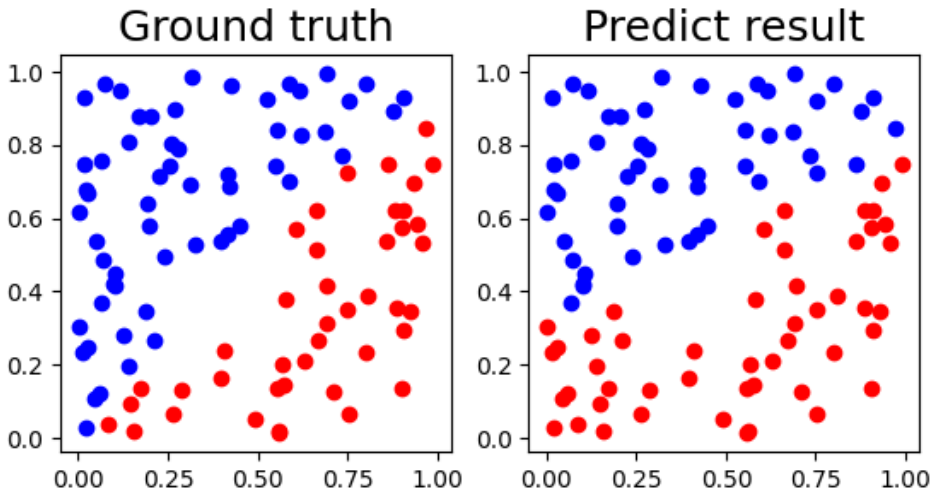
units	Linear data	XOR data
2	100%	52%
4	100%	100%
6	100%	100%
8	100%	100%
10	100%	100%

As we can see, only two units in each hidden layer of XOR data model are too few to train its model. But it's enough for linear data model to fit its data.

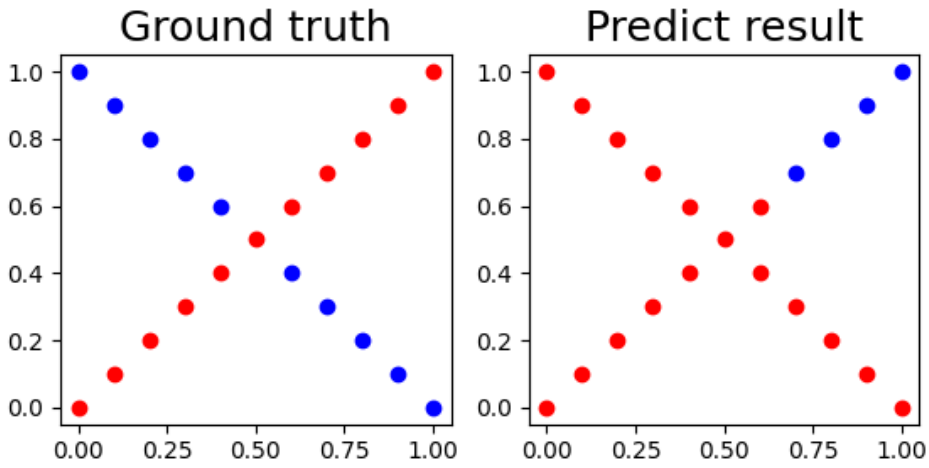
C. Try without activation functions

In this experiment, I try to remove the activation function of each layer and get the following result.

The linear data model still has 87% accuracy.



But the accuracy of XOR data model decrease to 33%.



The result shows that XOR data model without nonlinear activation functions can't classify XOR problem. Because the model without nonlinear activation function is like single layer perceptron, it can't solve XOR problem.

5. Extra

A. Implement different optimizers.

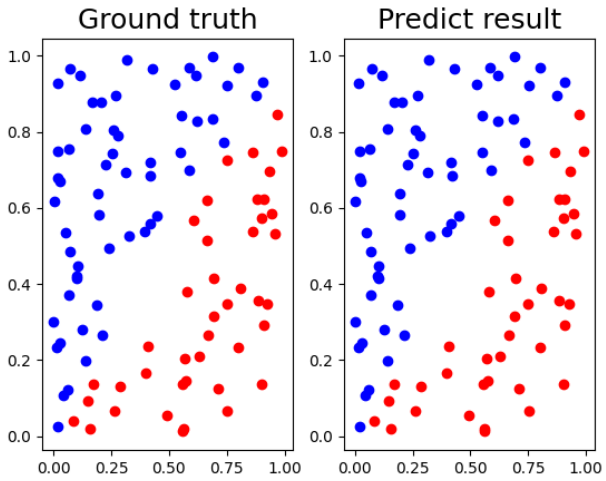
The Adagrad formula is shown below.

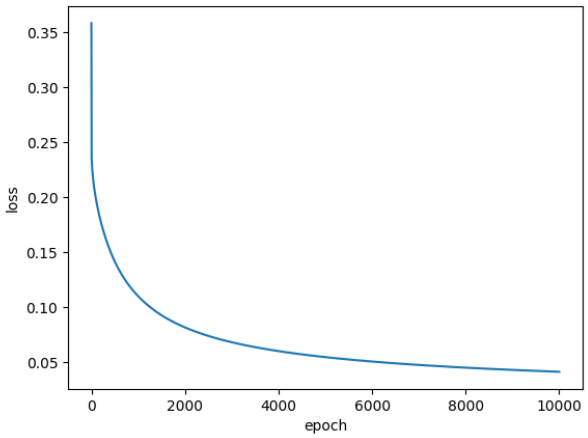
g^{i} means the gradient of weights.

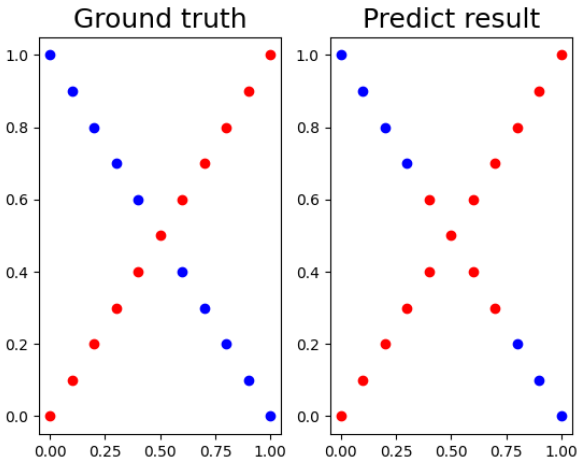
$$\mathrm{W}^{\mathrm{t+1}} = W^{t} - \frac{lr}{\sqrt{\sum_{i=0}^{t}(g^{\mathrm{i}})^{2}}} g^{t}$$

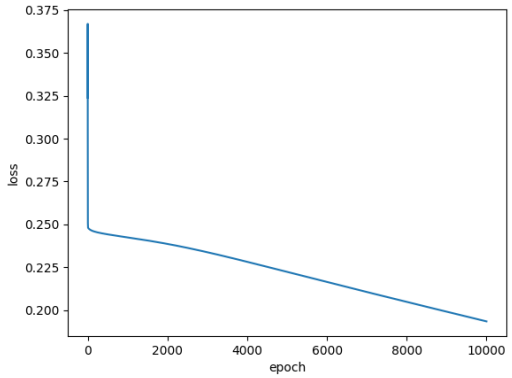
```
def update(self, lr, optimizer=''):
    '''
    Update weight
    '''
    self.lr = lr
    if optimizer == 'adagrad':
        self.w1_sum += (self.w1**2)
        self.w2_sum += (self.w2**2)
        self.w3_sum += (self.w3**2)
        self.w1 = self.w1 - self.lr * self.dl_dw1 / (self.w1_sum**0.5)
        self.w2 = self.w2 - self.lr * self.dl_dw2 / (self.w2_sum**0.5)
        self.w3 = self.w3 - self.lr * self.dl_dw3 / (self.w3_sum**0.5)
```

I also train the model with Adagrad optimizer, but don't get better result on XOR data


Ground truth
Predict result
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.25
0.50
0.75
1.00


loss
epoch
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0
2000
4000
6000
8000
10000


Ground truth
Predict result
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.25
0.50
0.75
1.00



B. Implement different activation functions (ReLU)

The ReLU function formula is shown below:

$$ReLU(\mathrm{x}) = \begin{cases} 0, & \mathrm{x} < 0 \\ \mathrm{x}, & \mathrm{x} \geq 0 \end{cases}$$

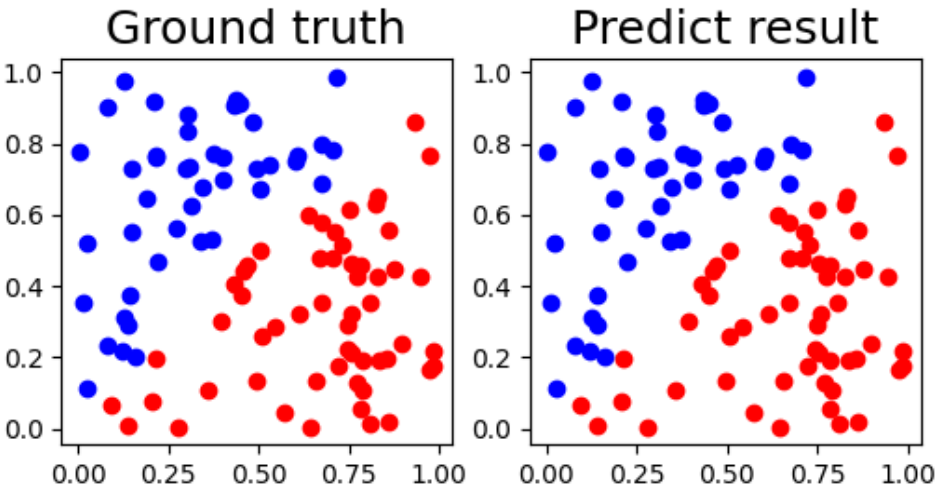
The implementation is shown below:

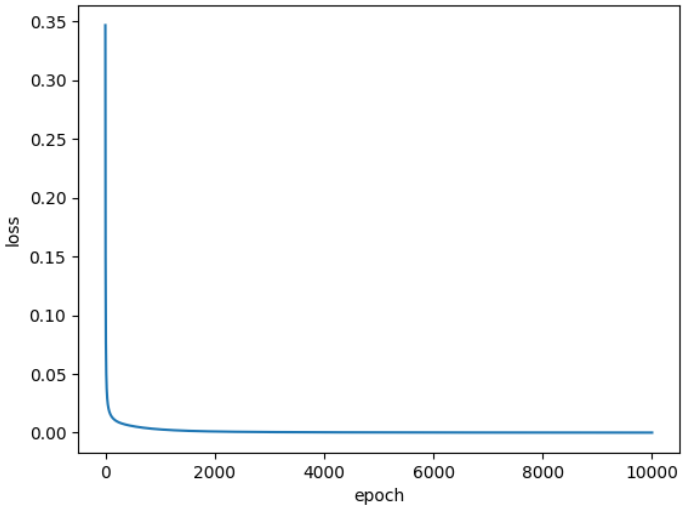
```
def ReLU(x):
    x[x<0] = 0
    return x

def derivative_ReLU(x):
    index_1 = x>=0
    index_2 = x<0
    x[index_1] = 1
    x[index_2] = 0
    return x
```

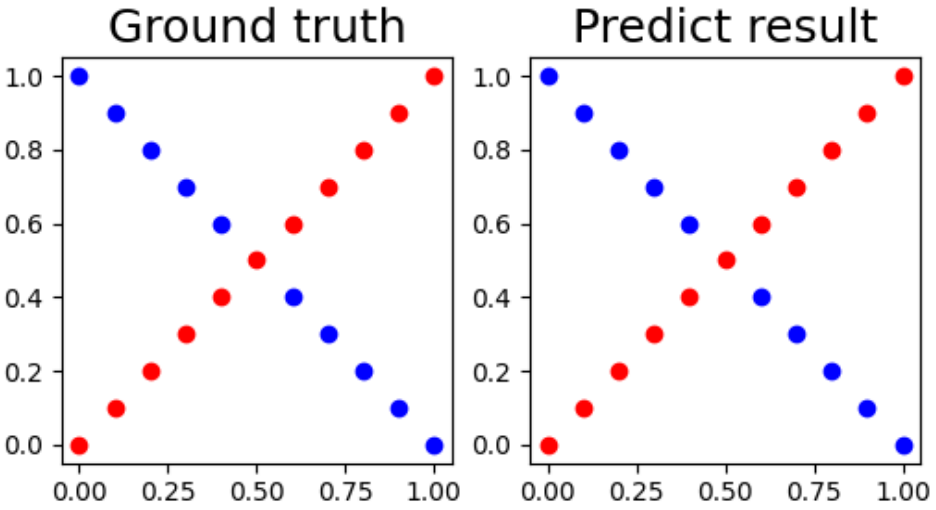
I also train the model with ReLU function, the results are shown below.

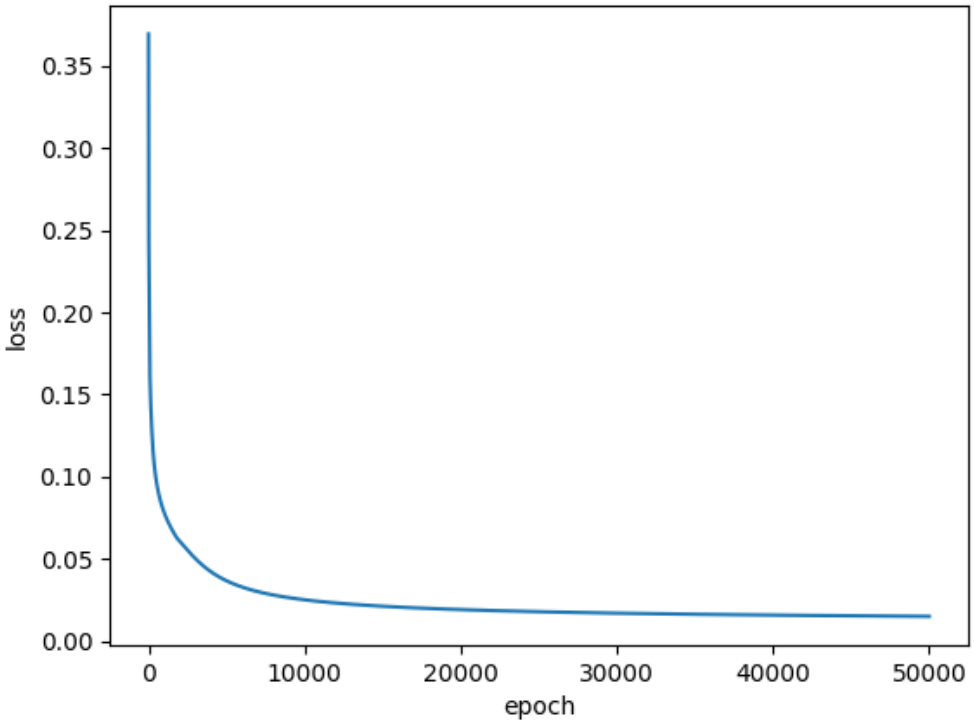
Linear data




0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
loss
0
2000
4000
6000
8000
10000
epoch

XOR data


Ground truth
Predict result
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.25
0.50
0.75
1.00


0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
loss
0
10000
20000
30000
40000
50000
epoch