Lab1 Backpropagation

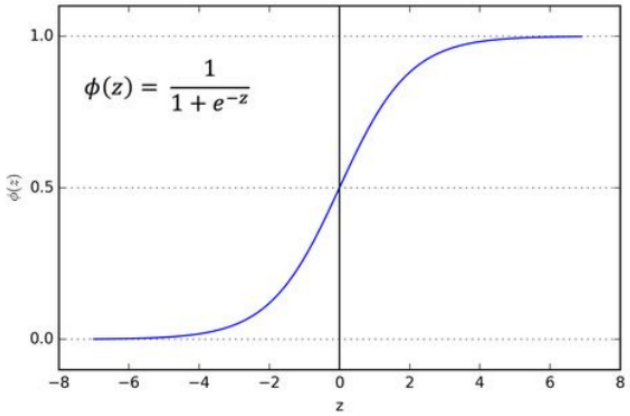
1. Introduction

In this lab, I implement a simple neural network with 2 hidden layers and also the backpropagation function. Through this simple example, I understand how a neural network learns from data with both forward and backward paths.

2. Experiment setups

a. Sigmoid function

The sigmoid function is a mathematical function that maps any input value to a value between 0 and 1. It is commonly used in machine learning and neural networks as an activation function to introduce non-linearity in the output of a neural network.



The output sigmoid function is always between 0 and 1, and the function is symmetric around x=0.

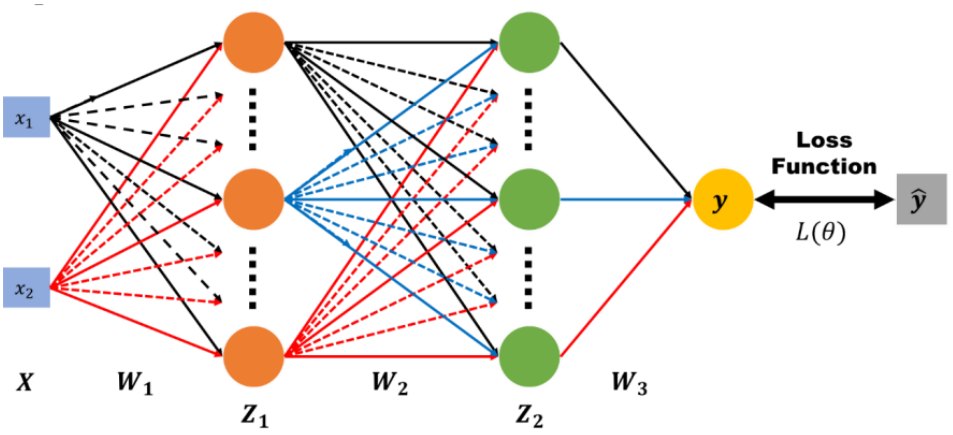
The logistic function is S-shaped and has a steep slope near the origin, which makes it useful for modeling binary classification problems, where the output is either 0 or 1. The sigmoid function is also used in other applications, such as in statistics to model probability distributions or in signal processing to normalize signals.

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

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

b. Neural networks

In this lab, I only need to construct a simple neural network with 2 hidden layers and a few neurons in each.



Define a simple neural network with 2 hidden layers and its forward path.

```
class Network():
    def __init__(self, x_in, h1_neuron, h2_neuron, y_out):
        self.w1 = np.random.randn(x_in, h1_neuron)
        self.w2 = np.random.randn(h1_neuron, h2_neuron)
        self.w3 = np.random.randn(h2_neuron, y_out)

    def forward(self, x):
        self.x = x
        self.z1 = sigmoid(x @ self.w1)
        self.z2 = sigmoid(self.z1 @ self.w2)
        self.z3 = sigmoid(self.z2 @ self.w3)
        self.pred_y = self.z3

        return self.pred_y
```

The loss is the MSE between ground truth and predicted results

```
loss = np.mean((y-pred_y) ** 2)
```

c. Backpropagation

Backpropagation is an algorithm used for training neural networks that adjust the weights of a neural network in order to minimize the difference between the predicted and actual outputs. The basic idea of backpropagation is to propagate the error backwards through the network from the output layer to the input layer, and use it to update the weights of the network. This is done by calculating the partial derivatives of the error with respect to the weights at each layer, and using these derivatives to adjust the weights in the opposite direction of the gradient of the error.

```
def backpropagation(self, y):
    dy = derivative_MSE(y, self.pred_y)
    dz3 = derivative_sigmoid(self.z3)
    dz2 = derivative_sigmoid(self.z2)
    dz1 = derivative_sigmoid(self.z1)

    self.d_l3 = self.z2.T @ (dz3 * dy)
    self.d_l2 = self.z1.T @ (dz2  * ((dz3 * dy) @ self.w3.T))
    self.d_l1 = self.x.T @ (dz1 * ((dz2  * ((dz3 * dy) @ self.w3.T)) @ self.w2.T))
```

The derivative of each layer is calculated recursively from output layer to input layer.

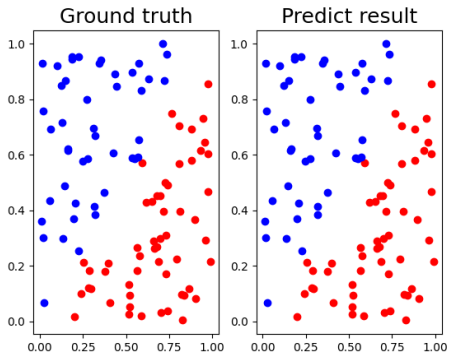
3. Results

a. Comparison figures

Task1 - Linear

```
epoch 1000 loss : 0.13396498238975865 acc(%) : 96.0
epoch 2000 loss : 0.0582528238241668 acc(%) : 100.0
epoch 3000 loss : 0.037296932029524145 acc(%) : 100.0
epoch 4000 loss : 0.028526341207265368 acc(%) : 100.0
epoch 5000 loss : 0.023683424921394018 acc(%) : 100.0
epoch 6000 loss : 0.020550836533876377 acc(%) : 100.0
epoch 7000 loss : 0.018310676890999 acc(%) : 100.0
epoch 8000 loss : 0.016594717580902456 acc(%) : 100.0
epoch 9000 loss : 0.015214132128105023 acc(%) : 100.0
epoch 10000 loss : 0.014062717521979922 acc(%) : 100.0
epoch 11000 loss : 0.013076392539013867 acc(%) : 100.0
epoch 12000 loss : 0.012214260621198738 acc(%) : 100.0
epoch 13000 loss : 0.011448985427724958 acc(%) : 100.0
epoch 14000 loss : 0.010761565485350098 acc(%) : 100.0
epoch 15000 loss : 0.010138339185139275 acc(%) : 100.0
epoch 16000 loss : 0.009569188226590279 acc(%) : 100.0
epoch 17000 loss : 0.009046417426821152 acc(%) : 100.0
epoch 18000 loss : 0.008564032688121738 acc(%) : 100.0
epoch 19000 loss : 0.008117262089078207 acc(%) : 100.0
epoch 20000 loss : 0.0077022302560196656 acc(%) : 100.0
```

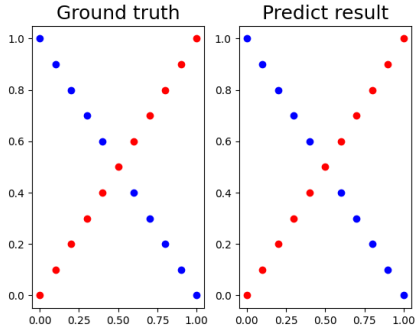
```
pred_y:
[[9.95038352e-01]
 [3.17563998e-04]
 [3.11766771e-04]
 [4.13547493e-04]
 [6.31936875e-04]
 [3.30095977e-04]
 [9.44380923e-03]
 [6.85395275e-03]
 [9.75513168e-01]
 [9.99277439e-01]
 [1.15168606e-03]
 [5.87456193e-04]
 [9.98674011e-01]
 [9.98737655e-01]
 [9.99246574e-01]
 [3.95994150e-02]
 [4.07062554e-04]
 [2.95168161e-04]
 [6.05653690e-04]
 [9.96457621e-01]
```



Task2 - XOR

```
epoch 1000 loss : 0.2473680486092798 acc(%) : 57.14285714285714
epoch 2000 loss : 0.24518797209022652 acc(%) : 61.904761904761905
epoch 3000 loss : 0.24185191980812867 acc(%) : 71.42857142857143
epoch 4000 loss : 0.23661062225489507 acc(%) : 71.42857142857143
epoch 5000 loss : 0.2284387921881568 acc(%) : 76.19047619047619
epoch 6000 loss : 0.21653859155322305 acc(%) : 76.19047619047619
epoch 7000 loss : 0.2015212628824377 acc(%) : 76.19047619047619
epoch 8000 loss : 0.18110273756797504 acc(%) : 71.42857142857143
epoch 9000 loss : 0.14225826909001366 acc(%) : 85.71428571428571
epoch 10000 loss : 0.10256335218848904 acc(%) : 90.47619047619048
epoch 11000 loss : 0.07700558190340341 acc(%) : 90.47619047619048
epoch 12000 loss : 0.06072731505513692 acc(%) : 95.23809523809523
epoch 13000 loss : 0.04934521315799968 acc(%) : 95.23809523809523
epoch 14000 loss : 0.04075477113977421 acc(%) : 95.23809523809523
epoch 15000 loss : 0.033945735424733926 acc(%) : 100.0
epoch 16000 loss : 0.028400862394263315 acc(%) : 100.0
epoch 17000 loss : 0.023838171520206418 acc(%) : 100.0
epoch 18000 loss : 0.020081679146769862 acc(%) : 100.0
epoch 19000 loss : 0.016998682108550718 acc(%) : 100.0
epoch 20000 loss : 0.014475715753704346 acc(%) : 100.0
```

```
pred_y:
[[0.00453375]
 [0.98297559]
 [0.01852049]
 [0.98277596]
 [0.06565587]
 [0.97801242]
 [0.14325061]
 [0.95449006]
 [0.19204493]
 [0.74073214]
 [0.18391452]
 [0.14182713]
 [0.686605  ]
 [0.09614804]
 [0.94899539]
 [0.06171503]
 [0.97462424]
 [0.03967286]
 [0.97954042]
 [0.0264174 ]
 [0.98102308]]
```



b. Accuracy

Linear:

```
====================
Accuracy(%): 100.0
```

XOR:

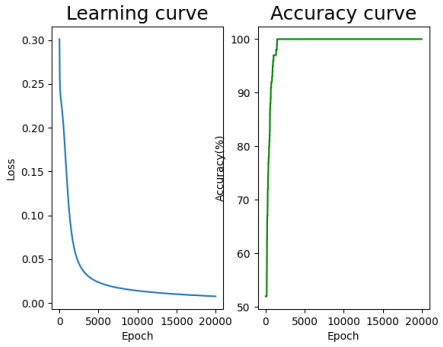
```
====================
Accuracy(%): 100.0
```

c. Learning curves

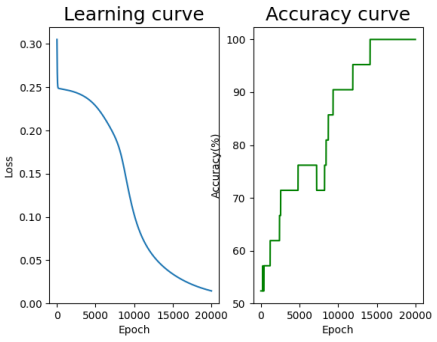
epoch: 20000

lr: 0.1

linear



XOR

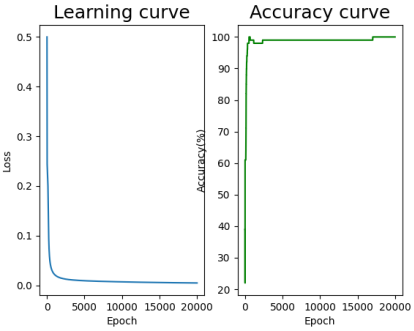


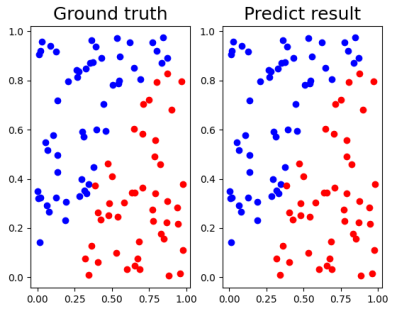
4. Discussion

a. Different learning rates

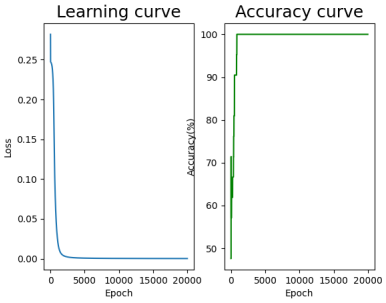
lr == 1:

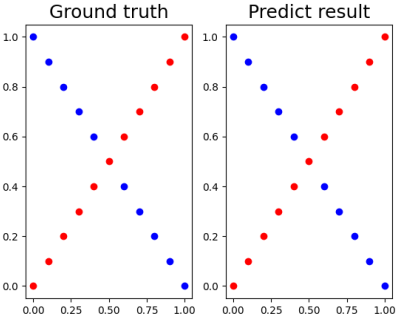
Linear:





XOR:

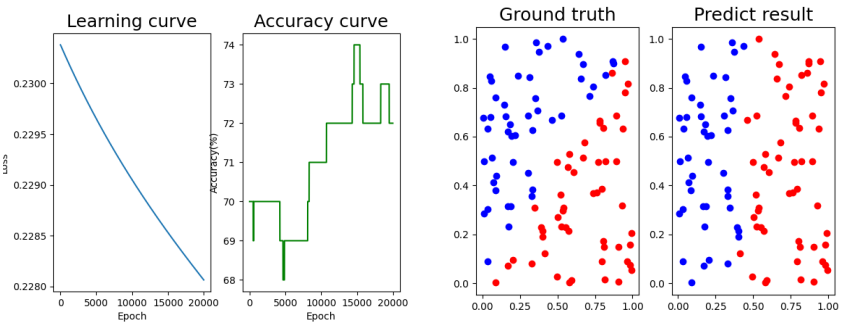




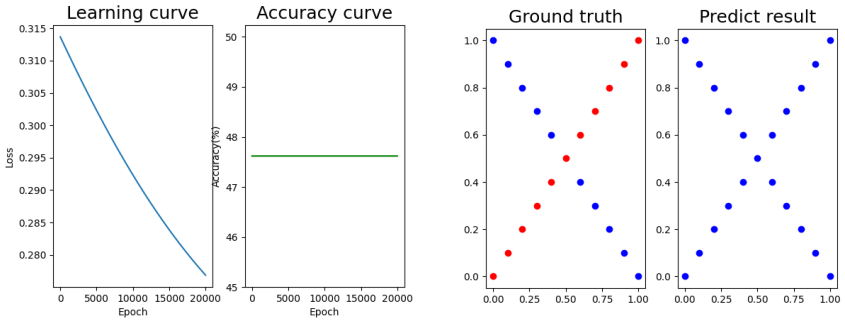
Converge faster especially on xor task.

lr == 0.0001

Linear:



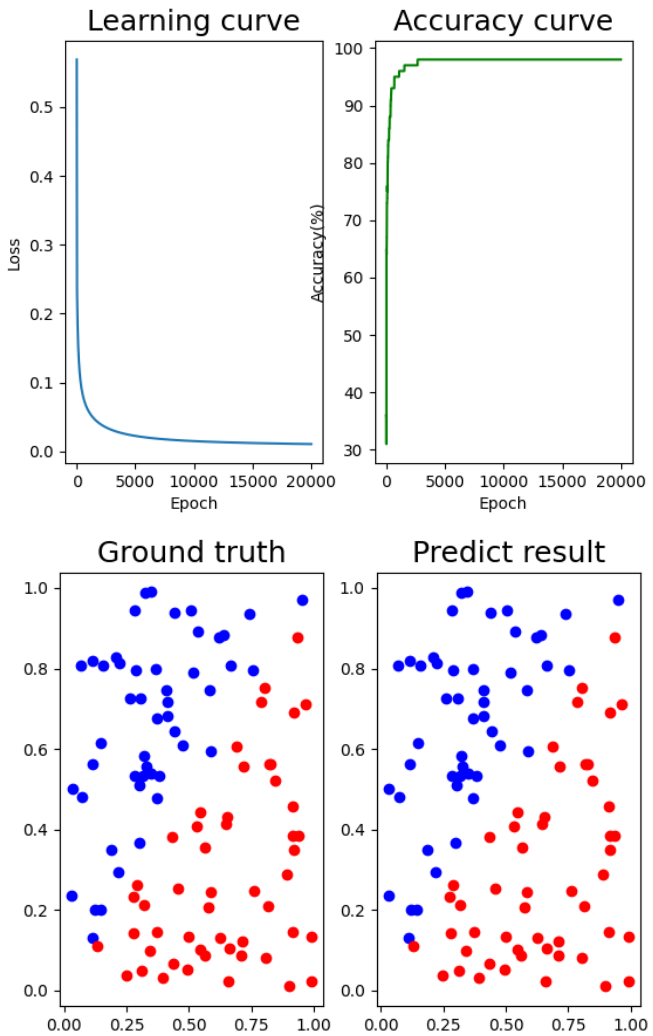
XOR:

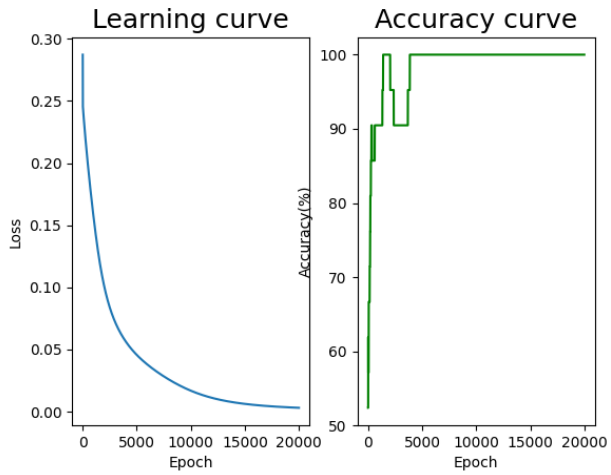


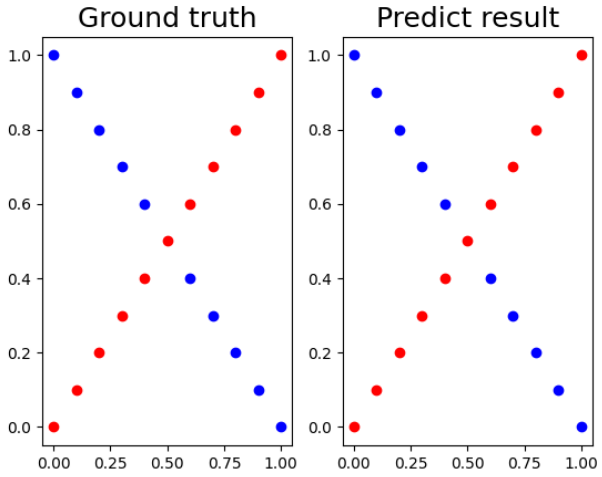
Didn't converge on both tasks and has only 81.0% and 47.6% accuracy respectively.

b. Different hidden units

32 X 32:

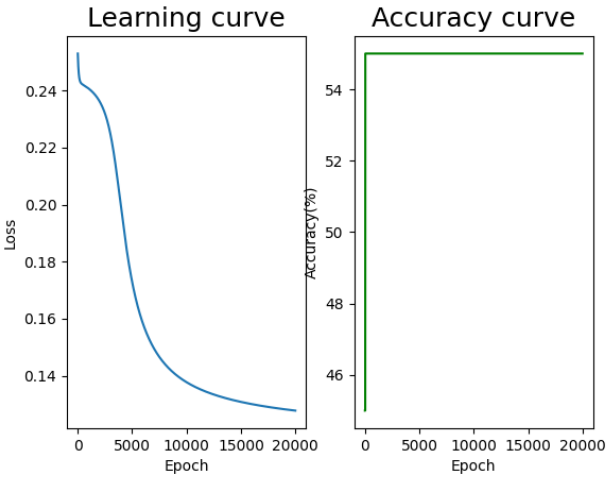


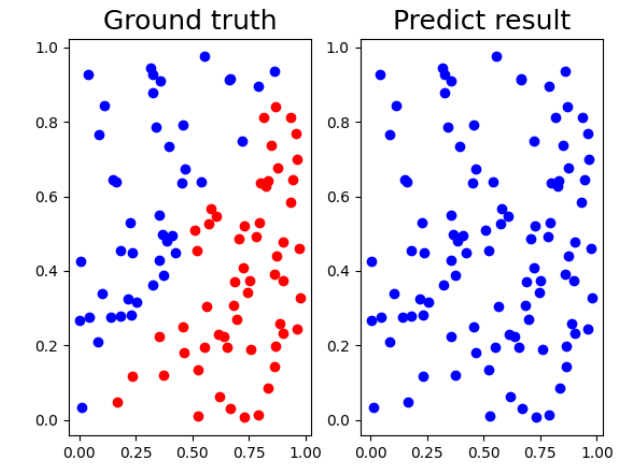


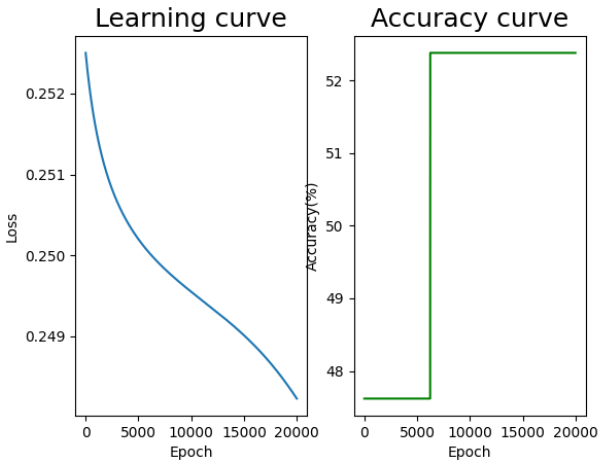


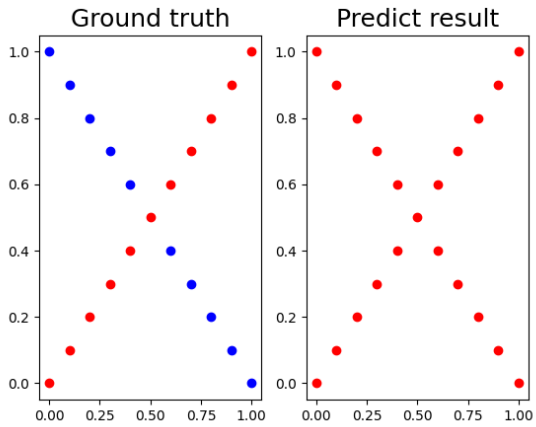
Still perform well on both tasks.

2 X 1:



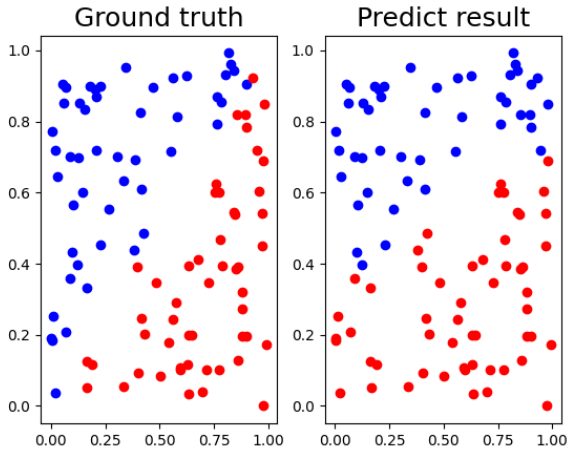




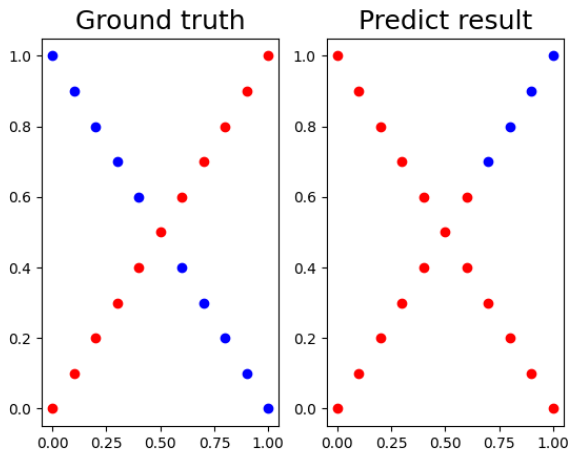


Can not converge within 20000 epochs.

c. Without activation function



The accuracy drop to 85% but the network still learn something since the function it tried to learn is a linear function.



However it can not deal with xor task because of its non-linearity.