1. Introduction

   In this lab, there are two hidden layers to form the neural network. Using numpy to do the forwarding and process backpropagation by the value of loss function, and update the weight between input and hidden layer 1, hidden layer 1 and hidden layer 2, hidden layer2 and output.

2. Experiment setups

   A. Sigmoid functions

   In this lab, I use sigmoid function which is

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

   as the activation function between every layer, and its deviation is

$$f(x)\big(1 - f(x)\big)$$

   B. Neural network

   There are layers including input layer, hidden layer 1, hidden layer 2, output layer, they are all fully-connected. Weight initialization using

```
np.random.seed(99999)
np.random.seed(seed)

w1 = np.random.rand(x.shape[1], h1)
w2 = np.random.rand(h1, h2)
w3 = np.random.rand(h2, 1)
```

   In addition, I wrap the neural network as a function like

```
network(epoch=10000, lr=0.1, h1=3, h2=3, distribution='XOR', seed=0)
```

   epoch: the max iteration to do forwarding and backpropagation, but if the condition fulfilled, it will stop no matter the iteration equal epoch or not. (the condition is accuracy = 100%)
   lr: the learning rate.
   h1: the hidden units in hidden layer 1.
   h2: the hidden units in hidden layer 2.
   distribution: there are two options: 'XOR' or 'linear'.
   seed: set the seed of the random numpy.

   C. Backpropagation

   I use the loss function called *binary cross entropy*, and all gradient of weights can be computed as below

$$X - W_1 \xrightarrow{z_1} \sigma \xrightarrow{a_1} W_2 \xrightarrow{z_2} \sigma \xrightarrow{a_2} W_3 \xrightarrow{z_3} \sigma - \hat{y} \xleftrightarrow{L} y$$

$\times$: element wise multiply

$\cdot$: matrix multiply

$$L = -y\log\hat{y} - (1-y)\log(1-\hat{y})$$

$$\frac{\partial L}{\partial \hat{y}} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})}$$

$$\frac{\partial L}{\partial z_3} = \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z_3} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})} \times (\hat{y}(1-\hat{y}))$$

$$\frac{\partial L}{\partial W_3} = \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial z_3}\frac{\partial z_3}{\partial W_2} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})} \times (\hat{y}(1-\hat{y})) \times \frac{\partial a_2 \cdot W_3}{\partial W_3} = a_2^T \cdot \left(\frac{\hat{y}-y}{\hat{y}(1-\hat{y})} \times (\hat{y}(1-\hat{y}))\right)$$

$$\frac{\partial L}{\partial a_2} = \frac{\partial L}{\partial z_3}\frac{\partial z_3}{\partial a_2} = \frac{\partial L}{\partial z_3}\frac{\partial a_2 \cdot W_3}{\partial a_2} = \frac{\partial L}{\partial z_3} \cdot W_3^T$$

$$\frac{\partial L}{\partial z_2} = \frac{\partial L}{\partial a_2}\frac{\partial a_2}{\partial z_2} = \frac{\partial L}{\partial a_2} \times (a_2(1-a_2))$$

$$\frac{\partial L}{\partial W_2} = a_1^T \cdot \frac{\partial L}{\partial z_2}$$

$$\frac{\partial L}{\partial a_1} = \frac{\partial L}{\partial z_2} \cdot W_2^T$$

$$\frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial a_1} \times (a_1(1-a_1))$$

$$\frac{\partial L}{\partial W_1} = X^T \cdot \frac{\partial L}{\partial z_1}$$

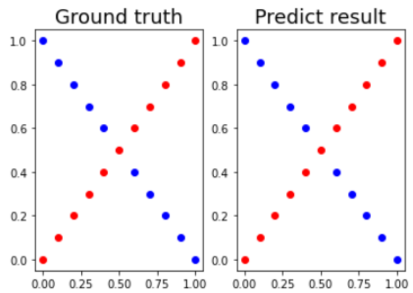
update weight using wi -= lr * grad_wi (i = 1, 2, 3)

3. Results of your testing

   A. Screenshot and comparison figure

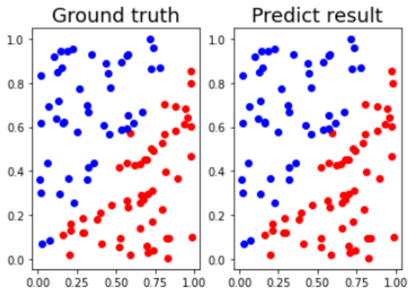
```
network(epoch=10000, lr=0.1, h1=3, h2=3, distribution='XOR', seed=0)
```

```
epoch: 100 accuracy: 0.5238095238095238 loss: 14.535359703924756
epoch: 200 accuracy: 0.5238095238095238 loss: 14.534297101623464
epoch: 300 accuracy: 0.5238095238095238 loss: 14.533385016210135
epoch: 400 accuracy: 0.5238095238095238 loss: 14.532525558280865
epoch: 500 accuracy: 0.5238095238095238 loss: 14.531629213192884
epoch: 600 accuracy: 0.5238095238095238 loss: 14.53059186751609
epoch: 700 accuracy: 0.5238095238095238 loss: 14.529264169943021
epoch: 800 accuracy: 0.5238095238095238 loss: 14.527394749015413
epoch: 900 accuracy: 0.5238095238095238 loss: 14.524501616940366
epoch: 1000 accuracy: 0.5238095238095238 loss: 14.519527797093339
epoch: 1100 accuracy: 0.5238095238095238 loss: 14.509718671616021
epoch: 1200 accuracy: 0.5238095238095238 loss: 14.48590256084938
epoch: 1300 accuracy: 0.5238095238095238 loss: 14.407008548046027
epoch: 1400 accuracy: 0.5238095238095238 loss: 14.110971988988853
epoch: 1500 accuracy: 0.8571428571428571 loss: 13.30980709065917
epoch: 1600 accuracy: 0.5238095238095238 loss: 13.018915242873533
epoch: 1700 accuracy: 0.8095238095238095 loss: 8.684009429198237
epoch: 1769 accuracy: 1.0 loss: 4.3954408074595275
```



```
network(epoch=10000, lr=0.01, h1=2, h2=2, distribution='linear', seed=0)
```

```
epoch: 100 accuracy: 0.54 loss: 68.94822881602369
epoch: 200 accuracy: 0.54 loss: 68.8932643717875
epoch: 300 accuracy: 0.54 loss: 68.78442657873096
epoch: 400 accuracy: 0.54 loss: 68.41399532999709
epoch: 500 accuracy: 0.54 loss: 65.13126933511963
epoch: 600 accuracy: 0.96 loss: 29.271026550078194
epoch: 700 accuracy: 0.97 loss: 11.238978201183238
epoch: 800 accuracy: 0.99 loss: 6.552449075272853
epoch: 900 accuracy: 0.99 loss: 4.663473495313786
epoch: 1000 accuracy: 0.99 loss: 3.7005126106862307
epoch: 1100 accuracy: 0.99 loss: 3.1131512911497796
epoch: 1200 accuracy: 0.99 loss: 2.712696571961642
epoch: 1300 accuracy: 0.99 loss: 2.418734065460382
epoch: 1400 accuracy: 0.99 loss: 2.191445710032903
epoch: 1479 accuracy: 1.0 loss: 2.044271556081318
```



B. Show the accuracy of your prediction

XOR

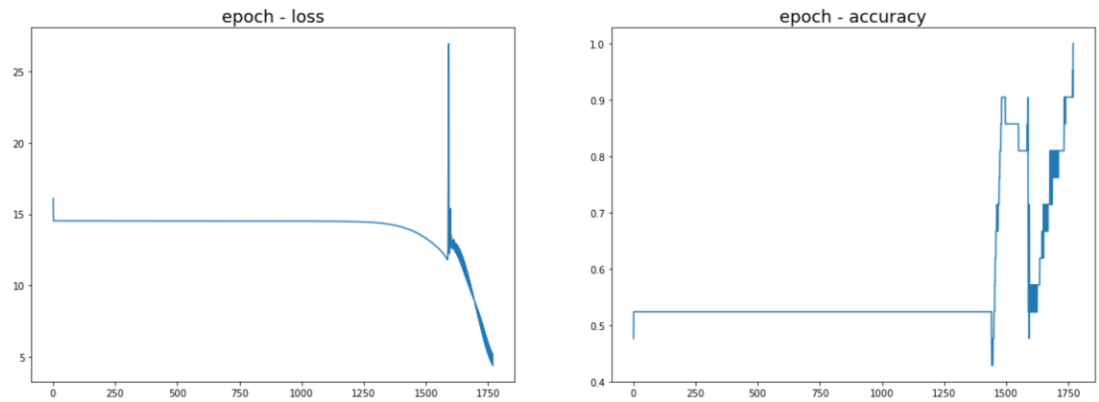
```
[[0.0553829 ]
 [0.98043249]
 [0.18155334]
 [0.97884826]
 [0.38342867]
 [0.97284542]
 [0.49956534]
 [0.94096973]
 [0.4949563 ]
 [0.71225017]
 [0.40918925]
 [0.2914546 ]
 [0.71158079]
 [0.18829961]
 [0.94082852]
 [0.11866489]
 [0.97303542]
 [0.07728232]
 [0.9791137 ]
 [0.05347794]
 [0.98071857]]
```

linear

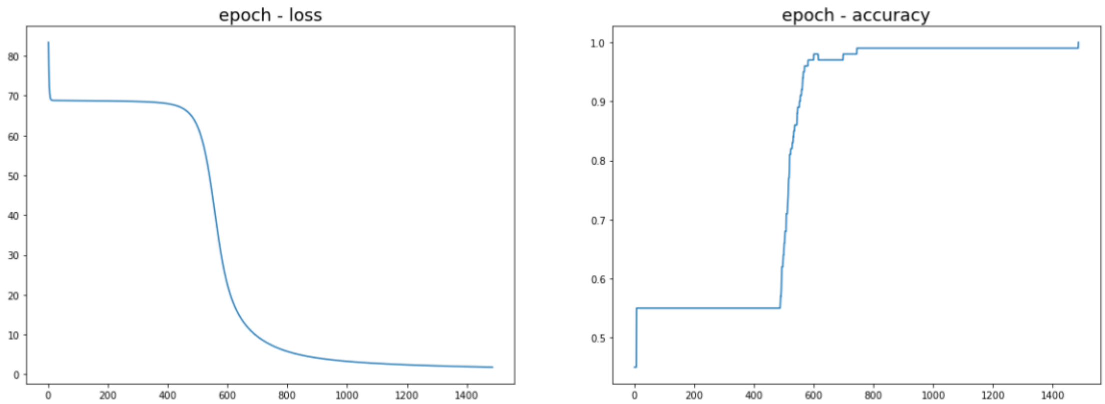
```
[[9.97485561e-01]
 [3.44261283e-03]
 [9.99863041e-01]
 [9.99905511e-01]
 [9.99917745e-01]
 [1.33101215e-02]
 [9.99905096e-01]
 [9.98001494e-01]
 [9.99910669e-01]
 [8.17466009e-01]
 [1.26129347e-03]
 [9.91559797e-01]
 [6.10162345e-04]
 [8.28201246e-01]
 [2.09453452e-02]
 [9.63104952e-01]
 [6.85656666e-03]
 [5.82219731e-04]
 [8.54934181e-02]
[7.31563501e-04]
[1.47431547e-03]
[7.45315863e-02]
[6.45347501e-04]
[9.99403192e-01]
[5.96900422e-04]
[5.94711189e-04]
[6.57974190e-04]
[7.96124726e-04]
[5.99626580e-04]
[2.77814798e-03]
[2.24114106e-03]
[9.95773155e-01]
[9.99912308e-01]
[9.34328222e-04]
[6.95601789e-04]
[9.99879692e-01]
[9.99873692e-01]
[9.99910184e-01]
[1.01387155e-02]
[6.26471484e-04]
[5.86229428e-04]
[7.23715104e-04]
[9.99782397e-01]
[9.99862294e-01]
[4.99937498e-01]
[9.99916210e-01]
[9.99890443e-01]
[7.31525635e-04]
[7.20422242e-04]
[9.97744867e-04]
[2.71302196e-03]
[1.60558682e-03]
[9.59384849e-04]
[9.99576129e-01]
[9.99817635e-01]
[7.73589414e-04]
[2.29583210e-03]
[9.99241378e-01]
[9.11207610e-01]
[9.94479059e-01]
[1.60749382e-03]
[6.45925677e-04]
[9.99900065e-01]
[2.39421114e-02]
[9.99918039e-01]
[9.99848768e-01]
[9.99915795e-01]
[9.99897720e-01]
[9.99915911e-01]
[1.45226899e-03]
[8.11382641e-04]
[1.33267227e-03]
[9.99332980e-01]
[1.63696589e-02]
[9.99867283e-01]
[6.31418173e-04]
[6.68464727e-04]
[2.01274386e-03]
[6.28764036e-04]
[9.99910731e-01]
[6.03679512e-04]
[9.99898290e-01]
[8.25355642e-04]
[9.24978462e-04]
[9.82457757e-01]
[7.15293534e-04]
[2.05071963e-03]
[9.99916871e-01]
[1.29862011e-03]
[8.66974488e-01]
[9.99879260e-01]
[9.99885291e-01]
[1.80392044e-03]
[9.08595167e-01]
[1.54852735e-03]
[9.99881270e-01]
[5.92006966e-04]

[7.32199088e-04]
[9.99916127e-01]
[9.82744786e-01]]
```

C. Learning curve (loss, epoch curve)

XOR



linear



4. Discussion

A. Try different learning rates

XOR

```
for i in range(5):
    print(network_(epoch=50000, lr=0.5, h1=3, h2=3, distribution='XOR', seed=i),
          network_(epoch=50000, lr=0.1, h1=3, h2=3, distribution='XOR', seed=i),
          network_(epoch=50000, lr=0.05, h1=3, h2=3, distribution='XOR', seed=i),
          network_(epoch=50000, lr=0.01, h1=3, h2=3, distribution='XOR', seed=i),
          network_(epoch=50000, lr=0.005, h1=3, h2=3, distribution='XOR', seed=i))
```

```
50000 1769 3354 17327 34644
50000 9292 17510 50000 50000
50000 3520 6700 34021 50000
50000 1223 2413 12036 24066
50000 951 1909 9535 19067
```

the output is the epoch the neural network with the learning rate had achieved accuracy = 100%, if epoch = 50000, it means the neural network with the learning rate cannot achieve accuracy = 100%. We can observe that learning rate equal 0.1 has the less average epoch to achieve 100%, and if the learning rate is too big or too small the training would take long time or never converge as the worst case.

linear

```
for i in range(5):
    print(network_(epoch=50000, lr=0.1, h1=2, h2=2, distribution='linear', seed=i),
          network_(epoch=50000, lr=0.05, h1=2, h2=2, distribution='linear', seed=i),
          network_(epoch=50000, lr=0.01, h1=2, h2=2, distribution='linear', seed=i),
          network_(epoch=50000, lr=0.005, h1=2, h2=2, distribution='linear', seed=i),
          network_(epoch=50000, lr=0.001, h1=2, h2=2, distribution='linear', seed=i))
```

```
106 296 1479 2958 14783
277 243 1196 2388 11919
392 328 1669 3339 16696
97 159 789 1574 7860
116 2143 1583 3166 15827
```

learning rate equal 0.1 or 0.05 would be great.

B. Try different numbers of hidden units

XOR

```
for i in range(5):
    print(network_(epoch=50000, lr=0.1, h1=2, h2=2, distribution='XOR', seed=i),
          network_(epoch=50000, lr=0.1, h1=3, h2=3, distribution='XOR', seed=i),
          network_(epoch=50000, lr=0.1, h1=5, h2=5, distribution='XOR', seed=i),
          network_(epoch=50000, lr=0.1, h1=10, h2=10, distribution='XOR', seed=i))
```

```
50000 1769 1778 1175
50000 9292 2076 1646
10323 3520 2320 1725
50000 1223 1387 1407
50000 951 963 870
```

the greater numbers of hidden units, the less average epoch acquired.

linear

```
for i in range(5):
    print(network_(epoch=50000, lr=0.1, h1=2, h2=2, distribution='linear', seed=i),
          network_(epoch=50000, lr=0.1, h1=3, h2=3, distribution='linear', seed=i),
          network_(epoch=50000, lr=0.1, h1=5, h2=5, distribution='linear', seed=i),
          network_(epoch=50000, lr=0.1, h1=10, h2=10, distribution='linear', seed=i))
```

```
275 50000 50000 59
```

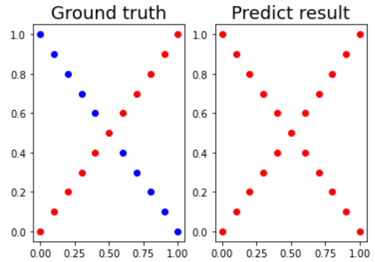
```
d:\miniconda3\lib\site-packages\ipykernel_launcher.py:84: RuntimeWarning: invalid value encountered in true_divide
d:\miniconda3\lib\site-packages\ipykernel_launcher.py:99: RuntimeWarning: divide by zero encountered in log
d:\miniconda3\lib\site-packages\ipykernel_launcher.py:99: RuntimeWarning: invalid value encountered in multiply
d:\miniconda3\lib\site-packages\ipykernel_launcher.py:82: RuntimeWarning: invalid value encountered in greater_equal
```

```
176 50000 50000 42
196 50000 50000 160
159 50000 65 39
103 50000 50000 66
```

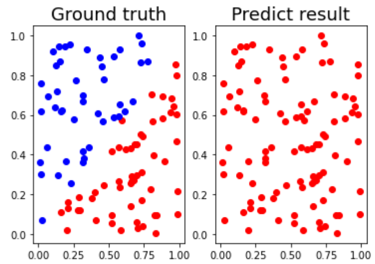
maybe the precision problem occurred, otherwise, the greater numbers of hidden units, the less average epoch acquired.

C. Try without activation functions

In [52]:
```
network_no_activation_func(epoch=10000, lr=0.1, h1=3, h2=3, distribution='XOR', seed=0)
```
```
epoch: 9800 accuracy: 0.5238095238095238 loss: nan
epoch: 9900 accuracy: 0.5238095238095238 loss: nan
epoch: 10000 accuracy: 0.5238095238095238 loss: nan
```



In [53]:
```
network_no_activation_func(epoch=10000, lr=0.01, h1=2, h2=2, distribution='linear', seed=0)
```
```
epoch: 9800 accuracy: 0.55 loss: nan
epoch: 9900 accuracy: 0.55 loss: nan
epoch: 10000 accuracy: 0.55 loss: nan
```



The problem occurred because my loss function used *binary cross entropy*, which deviation is

$$\frac{\partial L}{\partial \hat{y}} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})}$$

if the pred_y contains 0, the loss cannot be calculated.