Lab1：Backpropagation

- Introduction：

  本次作業用 python 實作 NN with Backpropagation：

  A. Neural Network：

$$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$$

$$a^{(L)} = \sigma(z^{(L)})$$

  a：the inputs of each layers

  L：the layer number

  $a^{(L)}$：output of the layer

  w：weights

  b：bias

  σ：activation function

  B. Backpropagation：

Backpropagation is used to calculate the gradients, starting from the output layer and propagating backwards, and updating weights and biases for each layer. The idea is that we nudge the weights and biases to get the desired output and minimize the cost function. The cost function is defined below:

$$C_0 = (a^{(L)} - y)^2$$

Then use partial derivatives and chain rule to calculate the relationship between the neural network components and the cost function from last layer to first layer. When we know what affects it, we can effectively change the relevant weights and biases to minimize the cost function.

Weights：

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C0}{\partial a^{(L)}} = a^{(L-1)} \sigma'(z^{(L)}) 2(a^{(L)} - y)$$

Biases：

$$\frac{\partial C_0}{\partial b^{(L)}} = \frac{\partial z^{(L)}}{\partial b^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C0}{\partial a^{(L)}} = \sigma'(z^{(L)}) 2(a^{(L)} - y)$$

Gradient：

$$\nabla C \leftarrow \begin{cases} \dfrac{\partial C}{\partial w_{jk}^{(l)}} = a_k^{(l-1)} \sigma'(z_j^{(l)}) \dfrac{\partial C}{\partial a_j^{(l)}} \\ \sum_{j=0}^{n_{l+1}-1} w_{jk}^{(l+1)} \sigma'(z_j^{(l+1)}) \dfrac{\partial C}{\partial a_j^{(l+1)}} \\ \text{or} \\ 2(a_j^{(L)} - y_j) \end{cases}$$

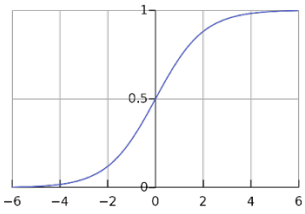
In the end, we subtract the weights from the weights multiply by the learning rate to update the weights as below:

$$w^{(L)} = w^{(L)} - learning\ rate \times \frac{\partial C}{\partial w^{(L)}}$$

- Experiment setups：
  A. Sigmoid functions：

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

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

$$S(t) = \frac{1}{1 + e^{-t}}$$



B. Neural networks：

1. Init：

設定好要訓練的 epoch 數，以及每一層 Neurons 的數目

```
def Net(x, y, lr, epoch=2000, HLsize=100, beta=0.9, beta2=0.1, n=100, epsllon=1e-8
    InputL, FirstL, SecondL, OutputL = x.shape[1], HLsize, HLsize, y.shape[1]

    W1, b1, Wv1, bv1, Wn1, bn1 = HL(InputL, FirstL)
    W2, b2, Wv2, bv2, Wn2, bn2 = HL(FirstL, SecondL)
    Wout, bout, Wvout, bvout, Wnout, bnout = HL(SecondL, OutputL)
```

隨機給定初始 weight & bias 的值

```
def HL(InputLayerSize, OuputLayerSize):
    W = np.random.uniform(-1, 1, size=(InputLayerSize, OuputLayerSize))
    b = np.random.uniform(-1, 1, size=(1, OuputLayerSize))
    Wv = 0
    bv = 0
    Wn = 0
    bn = 0
    return W, b, Wv, bv, Wn, bn
```

2. Forward：

$$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$$

$$a^{(L)} = \sigma(z^{(L)})$$

重複直到最後一層

```
# forward
z1 = GetActF(x , W1, b1, l, Activation)
z2 = GetActF(z1, W2, b2, l, Activation)
pred_y = GetActF(z2, Wout, bout, l, Activation=sigmoid)
```

C. Backpropagation：

1. 計算每次結果的誤差

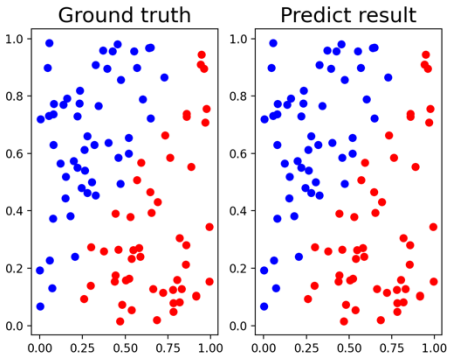
```
# backpropagation
error = pred_y - y
d_pred_y = 2*error * derivative_sigmoid(pred_y, l)

error_a2 = d_pred_y.dot(Wout.T)
d_z2 = error_a2 * Derivative(z2, l)

error_a1 = d_z2.dot(W2.T)
d_z1 = error_a1 * Derivative(z1, l)
```

2. 更新 weights & biases

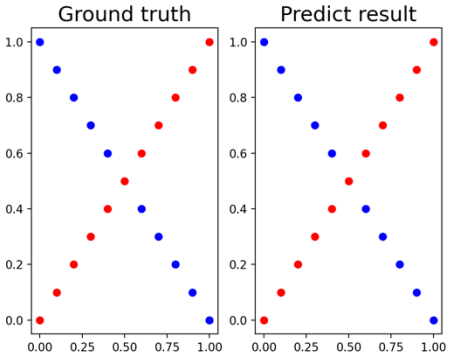
```
# update weights & biases
Wout, bout, Wnout, bnout = Upgrade(Wvout, bvout, beta, Wnout, bnout, beta2, epsllon, z2, d_pred_y, Wout, bout, lr, e)
W2, b2, Wn2, bn2 = Upgrade(Wv2, bv2, beta, Wn2, bn2, beta2, epsllon, z1, d_z2, W2, b2, lr, e)
W1, b1, Wn1, bn1 = Upgrade(Wv2, bv2, beta, Wn1, bn1, beta2, epsllon, x, d_z1, W1, b1, lr, e)
```

- Results of testing：

  A. Screenshot and comparison figure



Linear：lr=0.1, epoch=5000, layers=2,4,4,1, activation=sigmoid, optimizer=GD



XOR：lr=0.1, epoch=5000, layers=2,4,4,1, activation=sigmoid, optimizer=GD

  B. Show the accuracy of your prediction

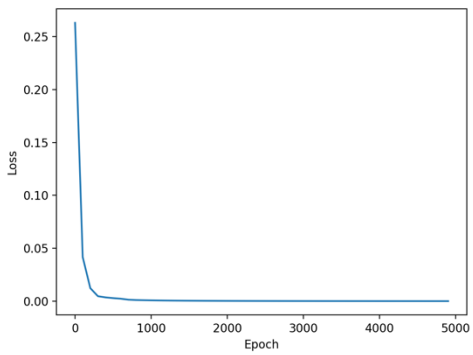
```
epoch: 0, loss: 26.295549488990762, acc: 0.49
epoch: 100, loss: 4.179391723613973, acc: 0.94
epoch: 200, loss: 1.2337181857404111, acc: 0.98
epoch: 300, loss: 0.472165501080743, acc: 0.99
epoch: 400, loss: 0.3612378940123535, acc: 0.99
epoch: 500, loss: 0.29195424566178446, acc: 1.0
epoch: 600, loss: 0.23653535344988483, acc: 1.0
epoch: 700, loss: 0.14397457579691228, acc: 1.0
epoch: 800, loss: 0.1108036837457237, acc: 1.0
epoch: 900, loss: 0.094875486645686 94, acc: 1.0
epoch: 1000, loss: 0.08224419671071671, acc: 1.0
```

Linear：lr=0.1, epoch=5000, layers=2,4,4,1, activation=sigmoid, optimizer=GD

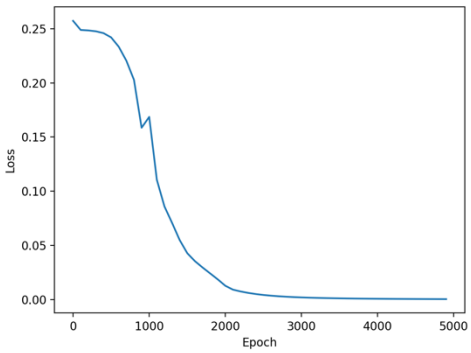
```
epoch: 0, loss: 5.402211466732375, acc: 0.47619047619047616
epoch: 100, loss: 5.225421853460363, acc: 0.5238095238095238
epoch: 200, loss: 5.216558712907537, acc: 0.5238095238095238
epoch: 300, loss: 5.199505617264359, acc: 0.5238095238095238
epoch: 400, loss: 5.163699991542498, acc: 0.42857142857142855
epoch: 500, loss: 5.081061761891651, acc: 0.42857142857142855
epoch: 600, loss: 4.899875598469338, acc: 0.47619047619047616
epoch: 700, loss: 4.629070759264108, acc: 0.5714285714285714
epoch: 800, loss: 4.25775160242503, acc: 0.5714285714285714
epoch: 900, loss: 3.330572398285446, acc: 0.8571428571428571
epoch: 1000, loss: 3.5407165318541876, acc: 0.7142857142857143
epoch: 1100, loss: 2.3228248635548767, acc: 0.8571428571428571
epoch: 1200, loss: 1.8059469770618717, acc: 0.8571428571428571
epoch: 1300, loss: 1.4877639682567299, acc: 0.8571428571428571
epoch: 1400, loss: 1.1542629925505383, acc: 0.9047619047619048
epoch: 1500, loss: 0.89912576640872, acc: 0.9523809523809523
epoch: 1600, loss: 0.7460893759464706, acc: 0.9523809523809523
epoch: 1700, loss: 0.6233087670623173, acc: 0.9523809523809523
epoch: 1800, loss: 0.5082656741989541, acc: 0.9523809523809523
epoch: 1900, loss: 0.39259668600042297, acc: 0.9523809523809523
epoch: 2000, loss: 0.2680291410294914, acc: 1.0
epoch: 2100, loss: 0.19319089234116088, acc: 1.0
epoch: 2200, loss: 0.15814676951607803, acc: 1.0
epoch: 2300, loss: 0.13019143019287455, acc: 1.0
epoch: 2400, loss: 0.10797416507486471, acc: 1.0
epoch: 2500, loss: 0.09034991219081287, acc: 1.0
epoch: 2600, loss: 0.07634078426449326, acc: 1.0
epoch: 2700, loss: 0.06514873790499831, acc: 1.0
epoch: 2800, loss: 0.05614336229467288, acc: 1.0
epoch: 2900, loss: 0.048836612867860396, acc: 1.0
epoch: 3000, loss: 0.04285469013366596, acc: 1.0
```

XOR：lr=0.1, epoch=5000, layers=2,4,4,1, activation=sigmoid, optimizer=GD

C. Learning curve (loss, epoch curve)

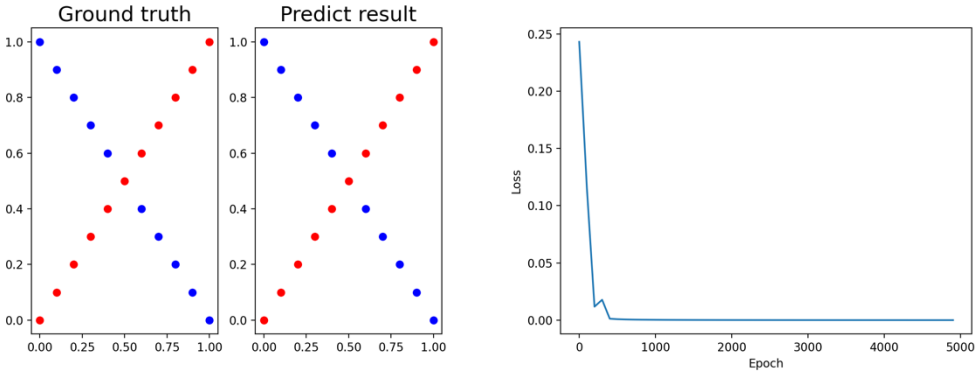


Linear：lr=0.1, epoch=5000, layers=2,4,4,1, activation=sigmoid, optimizer=GD

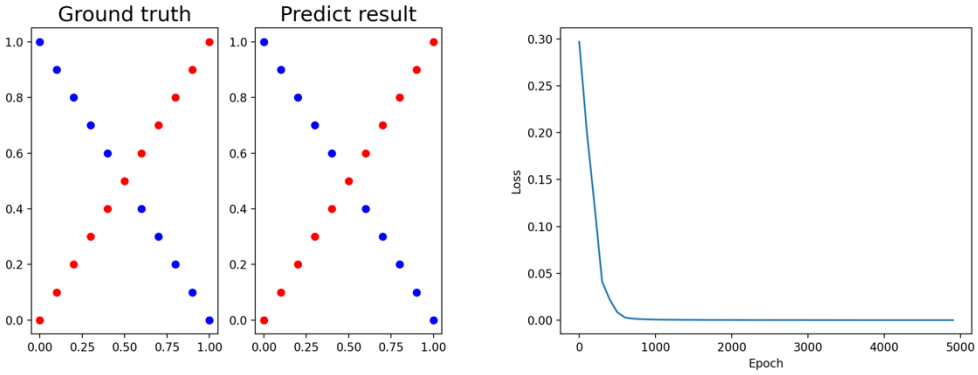


XOR：lr=0.1, epoch=5000, layers=2,4,4,1, activation=sigmoid, optimizer=GD

D. Other activation：relu, tanh



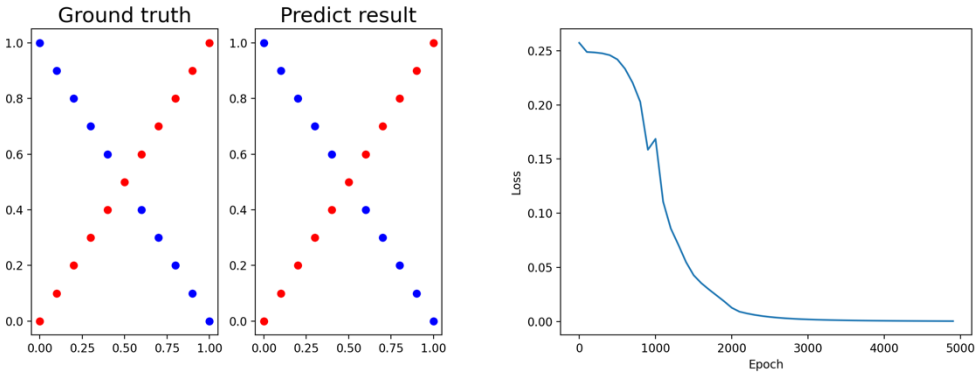
XOR：lr=0.1, epoch=5000, layers=2,4,4,1, activation=relu, optimizer=GD



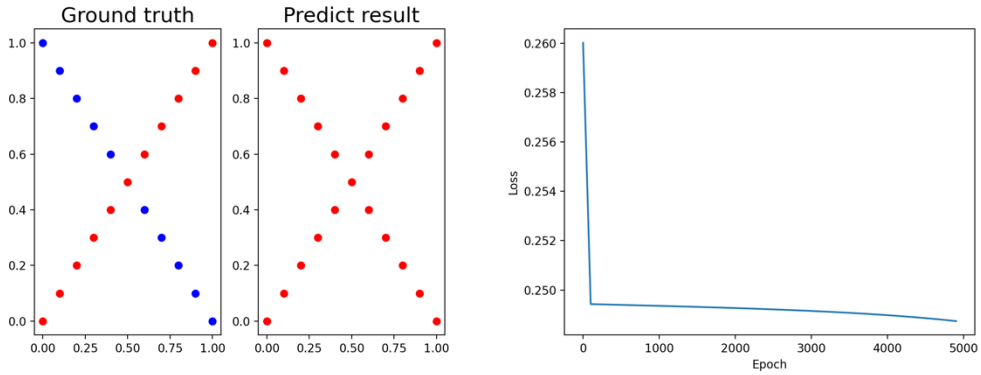
XOR：lr=0.1, epoch=5000, layers=2,4,4,1, activation=tanh, optimizer=GD

- Discussion：

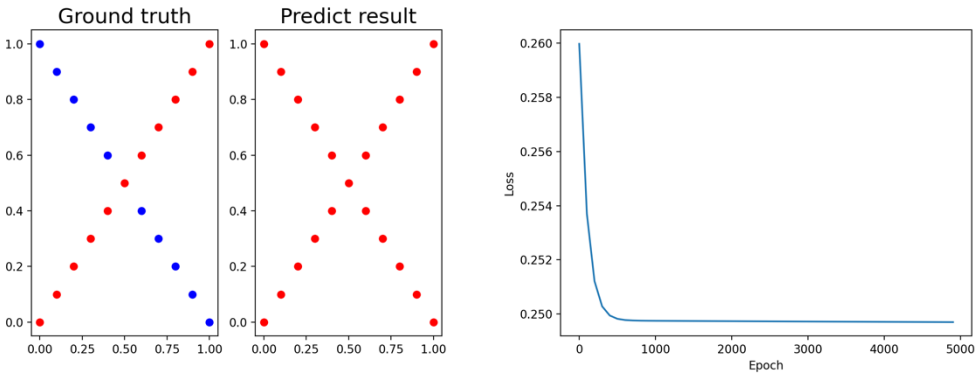
A. Try different learning rates：



XOR：lr=0.1, epoch=5000, layers=2,4,4,1, activation=sigmoid, optimizer=GD



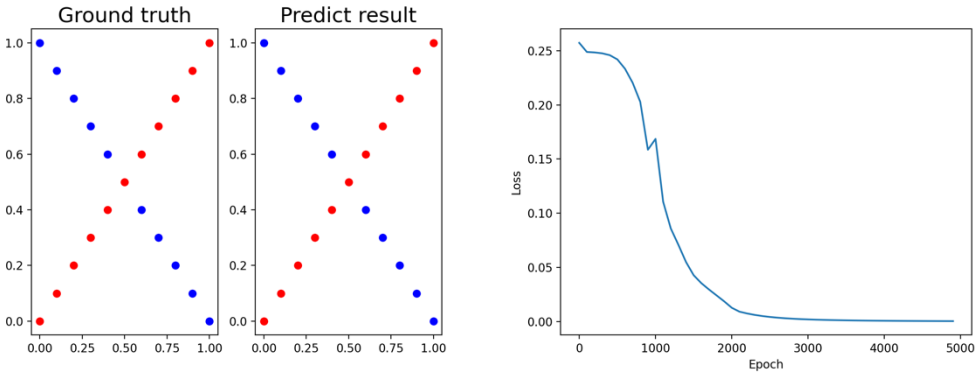
XOR：lr=0.01, epoch=5000, layers=2,4,4,1, activation=sigmoid, optimizer=GD



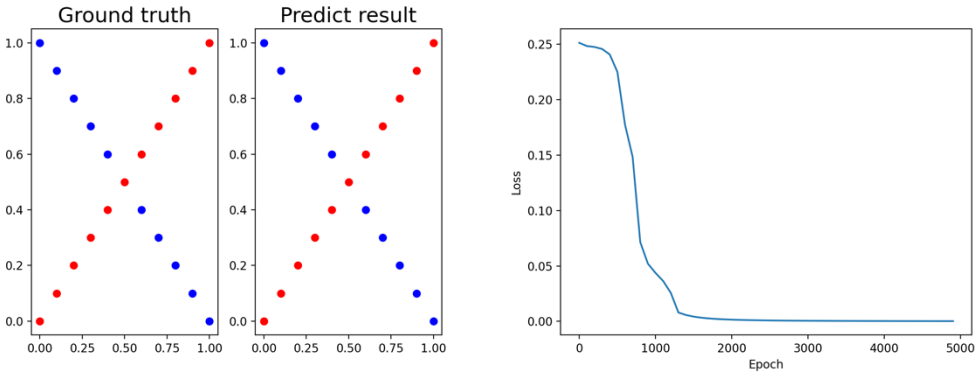
XOR：lr=0.001, epoch=5000, layers=2,4,4,1, activation=sigmoid, optimizer=GD

當使用 sigmoid 作為 activation function 時，learning rate 越低，反而容易造成 gradient 卡住，準確率剩一半，loss 卡在 0.25。

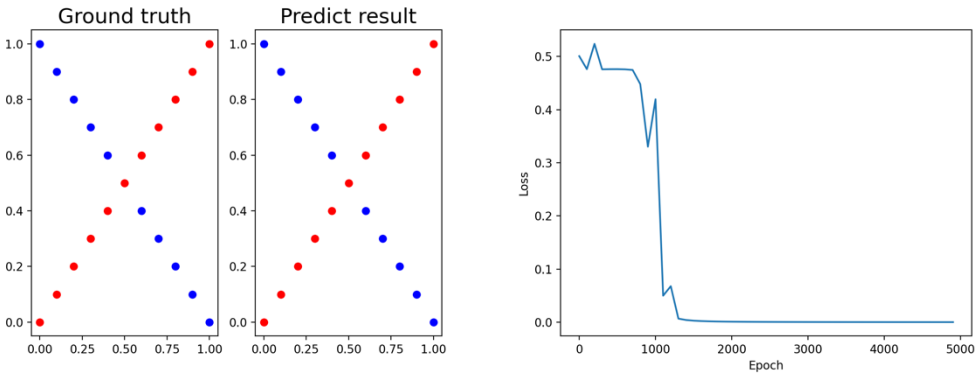
B. Try different numbers of hidden units：



XOR：lr=0.1, epoch=5000, layers=2,4,4,1, activation=sigmoid, optimizer=GD



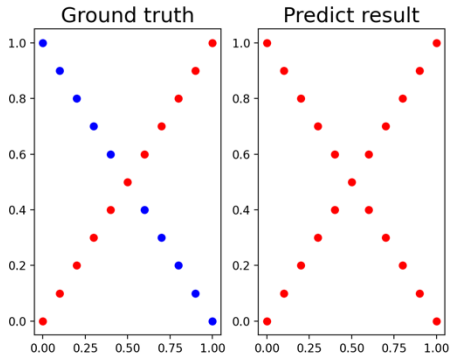
XOR：lr=0.1, epoch=5000, layers=2,8,8,1, activation=sigmoid, optimizer=GD

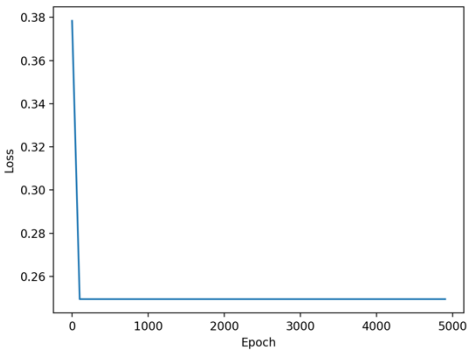


XOR：lr=0.1, epoch=5000, layers=2,100,100,1, activation=sigmoid, optimizer=GD

在 Neuron=4, 8, 100 中，模型的預測能力都很好，不過在 Neuron=100，epoch=1000 以下的 loss 跳動很大，可能 learning rate 在 neuron=100 時太大了。

C. Try without activation functions：



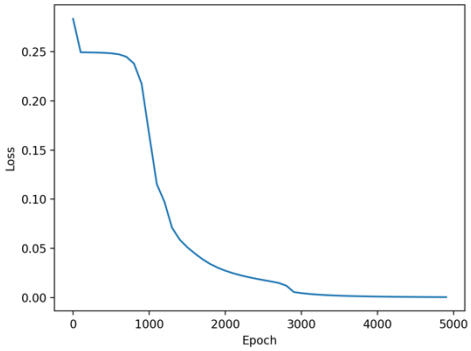


XOR：lr=0.1, epoch=5000, layers=2,4,4,1, activation=none, optimizer=GD

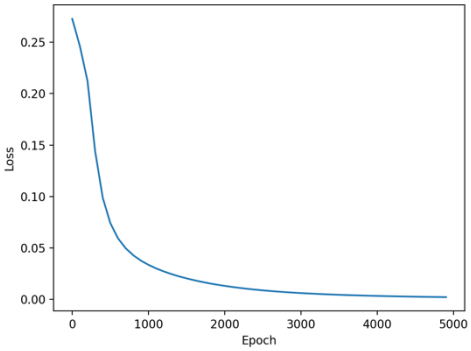
沒有 activation function 等同於找一個切平面切出最佳解，在 XOR 中人眼預期切出 0.75 的準確度，但觀察模型可以發現 loss 卡在 0.25 附近，也就是準確度只會有 0.5，模型無法切出 0.75 的平面。

D. Other optimizer：momentum, adagram, adam

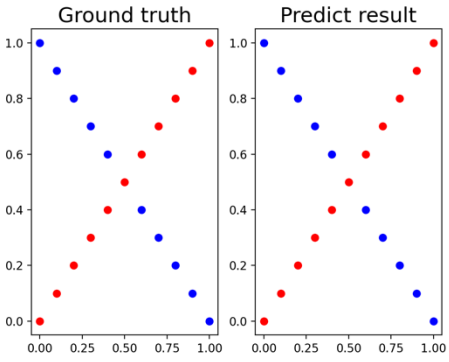
XOR：lr=0.01, epoch=5000, layers=2,4,4,1, activation=sigmoid

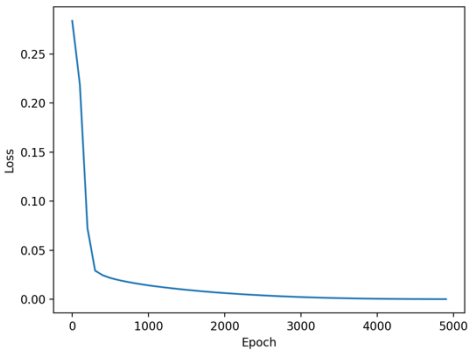


optimizer=momentum



optimizer=adagrad





XOR：lr=0.01, epoch=5000, layers=2,4,4,1, activation=sigmoid, optimizer=adam

- Extra：

  A. Implement different optimizers：

  1. momentum：

```
# momentum
def UpgradeM(Wv, bv, beta, Wn, bn, beta2, epsllon, z, dz, W, b, lr, e):
    g = z.T.dot(dz)
    Wv = beta*Wv + g*lr
    W -= Wv

    g = np.sum(dz, axis=0, keepdims=True)
    bv = beta*bv + g*lr
    b -= bv

    return W, b, Wn, bn
```

Repeat Until Convergence {

$$\nu_j \leftarrow \eta * \nu_j - \alpha * \nabla_w \sum_1^m L_m(w)$$

$$\omega_j \leftarrow \nu_j + \omega_j$$

}

  2. adagrad：

```
# Adagrad
def UpgradeAda(Wv, bv, beta, Wn, bn, beta2, epsllon, z, dz, W, b, lr, e):
    g = z.T.dot(dz)
    Wn += np.sum(np.square(g), axis=0)
    coef = 1/pow(Wn+epsllon, 1/2)
    W -= lr*coef*g

    g = np.sum(dz, axis=0, keepdims=True)
    bn += np.sum(np.square(g), axis=0)
    coef = 1/pow(bn+epsllon, 1/2)
    b -= lr*coef*g

    return W, b, Wn, bn
```

Adagrad

$$v_t = v_{t-1} + (\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{(v_t)} + \epsilon} \nabla w_t$$

  3. adam：

```python
# Adam
def UpgradeAdam(Wv, bv, beta, Wn, bn, beta2, epsllon, z, dz, W, b, lr, e):
    e = e+1

    g = z.T.dot(dz)
    Wv = beta*Wv + (1-beta)*(g**2)
    Wn = beta2*Wn + (1-beta2)*g
    v_hat = Wv / (1-beta**e)
    n_hat = Wn / (1-beta2**e)
    coef = n_hat/(np.sqrt(v_hat)+epsllon)
    W -= lr*coef

    g = np.sum(dz, axis=0, keepdims=True)
    bv = beta*bv + (1-beta)*(g**2)
    bn = beta2*bn + (1-beta2)*g
    v_hat = bv / (1-beta**e)
    n_hat = bn / (1-beta2**e)
    coef = n_hat/(np.sqrt(v_hat)+epsllon)
    b -= lr*coef

    return W, b, Wn, bn
```

First moment & second moment (mean, variance)

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

Bias correction

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Update parameters

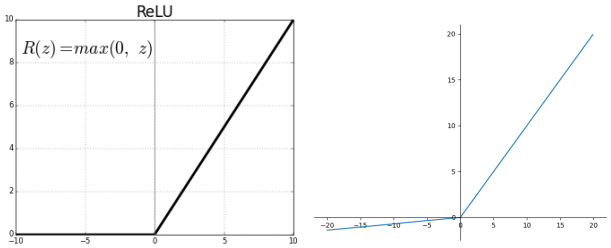
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

B. Implement different activation functions：

1. leaky relu, relu (when l=0)：

```python
#leaky relu
def leaky_relu(x, l):
    return np.where(x > 0, x, x*l)

def derivative_leaky_relu(x, l):
    return np.where(x > 0, 1, l)
```

Leaky ReLu



2. tanh：

```python
# tanh
def tanh(x, l):
    return np.tanh(x)

def derivative_tanh(x, l):
    return 1 - np.square(np.tanh(x))
```