Lab1

學號：311512049 姓名：陳緯翰

Introduction (20%)

實作兩層hidden layer的神經網路並不能使用任何插件。透過forward propagation預測答案和backpropagation對權重進行更新，並利用groud truth和predict出來的結果去計算loss，同時調整不同的learning_rate，activate function和隱藏層神經元，以探討對學習過程和準確度的影響。

Experiment setups (30%):

a. Sigmoid functions

sigmoid的公式為 $\sigma(x) = \frac{1}{1+e^{-x}}$，而其微分為 $\frac{d}{dx}\sigma(x) = \sigma(x)(1-\sigma(x))$ 而 sigmoid function在 forward propagation時能將數據分布變成0到1，以縮小數據保證數據幅度不會有太 大問題，但也會有容易出現梯度消失及運算耗時較久等問題。

```
def sigmoid(self, x):   # 使NN分布變成0~1
    return 1.0 / (1.0 + np.exp(-x))

def derivative_sigmoid(self, x):   # derivative_sigmoid為一個常數，因為x在sigmoid就已經決定了，backward只是後續的步驟
    return np.multiply(x, 1.0 - x)   # 有推導
```

b. Neural network

- 流程如下:

1. 初始化神經網路所有權重病加上Bias (此任務bias可加可不加)
2. 將資料由 input layer 往 output layer 向前傳遞 (forward pass) 並計算出所有神經元的 output
3. 誤差由 output layer 往 input layer 向後傳遞 (backward pass)並算出每個神經元對誤差的影響
4. 用誤差影響去更新權重 (weights)
5. 重複步驟 (2)~(4)直到誤差收斂夠小

```
class NeuralNetwork:
    def __init__(self, input_size, hidden_size_1, hidden_size_2, output_size, learning_rate):
        self.input_size = input_size
        self.hidden_size_1 = hidden_size_1
        self.hidden_size_2 = hidden_size_2
        self.output_size = output_size
        self.learning_rate = learning_rate

        # 初始化權重和偏置  y = Wx + b
        # 生成一個或多個符合標準正態分佈的隨機數，其均值為0，標準差為1
        self.weights_input_hidden_1 = np.random.randn(self.input_size, self.hidden_size_1)
        self.weights_hidden_1_hidden_2 = np.random.randn(self.hidden_size_1, self.hidden_size_2)
        self.weights_hidden_2_output = np.random.randn(self.hidden_size_2, self.output_size)

        self.bias_input_hidden_1 = np.random.randn(self.hidden_size_1)
        self.bias_hidden_1_hidden_2 = np.random.randn(self.hidden_size_2)
        self.bias_hidden_2_output = np.random.randn(self.output_size)        Henry Chen, 2 days ago
```

- weights可用np.random.randn和np.random.normal去生成兩個的差別如下：

	np.random.randn	np.random.normal
函數原型	numpy.random.randn(d0, d1, ..., dn)	numpy.random.normal(loc=0.0, scale=1.0, size=None)
返回值	符合標準正態分佈的隨機數	符合標準正態分佈的隨機數
參數 loc	No	正態分佈的均值
參數 scale	No	正態分佈的標準差
參數 size	返回隨機數的形狀	返回隨機數的形狀

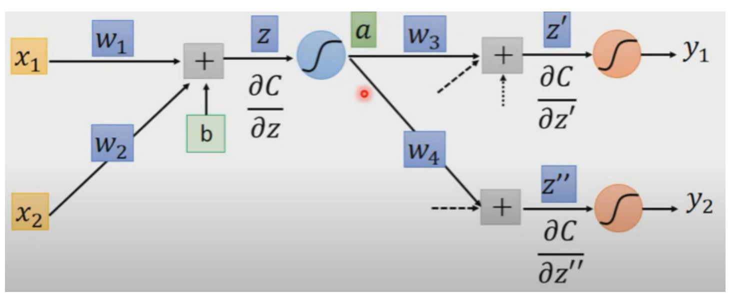
- 初始化神經網路的基本特性：
  input層之神經元數，各個 Hidden layer的神經元及 learning_rate等，並將所需要的 weight矩陣透過**np.random.randnt**產生平均值為 0標準差為 1符合標準正態分佈的隨機數，而我們預設 hidden layer1 及 hidden layer2其neurons數都為 4，learning rate則是 0.01
- 定義forward_propagation：

```
        You, 26 minutes ago • Uncommitted changes
def forward(self, x, mode):
    if mode == "sigmoid":
        # 輸入層到隱藏層1
        hidden_input_1 = np.dot(x, self.weights_input_hidden_1) + self.bias_input_hidden_1   # (n, 2)* (2, 4) + (4, ) =
        self.hidden_output_1 = self.sigmoid(hidden_input_1)
        # 隱藏層1到隱藏層2
        hidden_input_2 = np.dot(self.hidden_output_1, self.weights_hidden_1_hidden_2) + self.bias_hidden_1_hidden_2  #
        self.hidden_output_2 = self.sigmoid(hidden_input_2)
        # 隱藏層2到輸出層
        output_input = np.dot(self.hidden_output_2, self.weights_hidden_2_output) + self.bias_hidden_2_output # (n, 4)*
        output = self.sigmoid(output_input)
        return output

    elif mode == "relu":
        # 輸入層到隱藏層1
        hidden_input_1 = np.dot(x, self.weights_input_hidden_1) + self.bias_input_hidden_1
        self.hidden_output_1 = self.relu(hidden_input_1)
        # 隱藏層1到隱藏層2
        hidden_input_2 = np.dot(self.hidden_output_1, self.weights_hidden_1_hidden_2) + self.bias_hidden_1_hidden_2
        self.hidden_output_2 = self.relu(hidden_input_2)
        # 隱藏層2到輸出層
        output_input = np.dot(self.hidden_output_2, self.weights_hidden_2_output) + self.bias_hidden_2_output
        output = self.relu(output_input)
        return output
```

c. Backpropagation

神經網路主要就是要求取weight 和輸出的關係以便更新權重 $\frac{\partial C}{\partial W} = \frac{\partial Z}{\partial W}\frac{\partial C}{\partial Z}$，$\frac{\partial Z}{\partial W}$(可以輕易在forward求出)* $\frac{\partial C}{\partial Z}$(backward在求此)



整個loss對weight的偏為可以寫成：

$$\frac{\partial L(\theta)}{\partial w_1} = \frac{\partial y}{\partial w_1}\frac{\partial L(\theta)}{\partial y} = \frac{\partial x''}{\partial w_1}\frac{\partial z}{\partial x''}\frac{\partial L(\theta)}{\partial z} = \frac{\partial z}{\partial w_1}\frac{\partial x''}{\partial z}\frac{\partial L(\theta)}{\partial y} = \frac{\partial x'}{\partial w_1}\frac{\partial z}{\partial x'}\frac{\partial x''}{\partial z}\frac{\partial L(\theta)}{\partial y} = \frac{\partial x'}{\partial w_1}\frac{\partial z}{\partial x'}\frac{\partial x''}{\partial z}\frac{\partial L(\theta)}{\partial x''}\frac{\partial x''}{\partial y}$$

```python
def backward(self, x, y_true, y_pred, mode):  # output = y_pred

    if mode == "sigmoid":
        # 計算輸出層的梯度
        output_error = y_pred - y_true  # (n, 1)
        output_delta = output_error * self.derivative_sigmoid(y_pred)  # (n, 1)

        # 計算第二個隱藏層的梯度
        hidden_2_error = np.dot(output_delta, self.weights_hidden_2_output.T)  # (n, 1)* (4, 1).T = (n, 4)
        hidden_2_delta = hidden_2_error * self.derivative_sigmoid(self.hidden_output_2)
              Henry Chen, 2 days ago • first commit …
        # 計算第一個隱藏層的梯度
        hidden_1_error = np.dot(hidden_2_delta, self.weights_hidden_1_hidden_2.T)   # (n, 4)* (4, 4).t = (n, 4)
        hidden_1_delta = hidden_1_error * self.derivative_sigmoid(self.hidden_output_1)

    elif mode == "relu":
        # 計算輸出層的梯度
        output_error = y_pred - y_true
        output_delta = output_error * self.derivative_relu(y_pred)

        # 計算第二個隱藏層的梯度
        hidden_2_error = np.dot(output_delta, self.weights_hidden_2_output.T)
        hidden_2_delta = hidden_2_error * self.derivative_relu(self.hidden_output_2)

        # 計算第一個隱藏層的梯度
        hidden_1_error = np.dot(hidden_2_delta, self.weights_hidden_1_hidden_2.T)
        hidden_1_delta = hidden_1_error * self.derivative_relu(self.hidden_output_1)
```

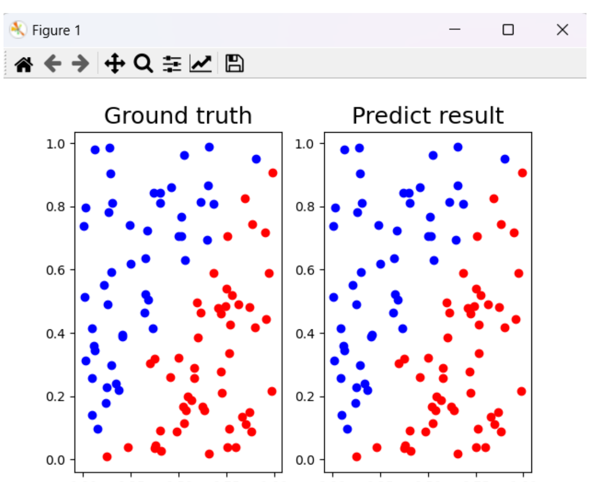
- 更新權重

```python
# 更新權重和偏置  求delta_C/delta_W = delta_Z/delta_W(可以輕易在forward求出)* delta_C/delta_Z(backward在求此)
self.weights_hidden_2_output -= self.learning_rate * np.dot(self.hidden_output_2.T, output_delta) # (n, 4).T* (n, 1) = (4, 1)
self.bias_hidden_2_output -= self.learning_rate * np.sum(output_delta, axis=0)
self.weights_hidden_1_hidden_2 -= self.learning_rate * np.dot(self.hidden_output_1.T, hidden_2_delta) # (n , 4).T *(n, 4) = (4, 4)
self.bias_hidden_1_hidden_2 -= self.learning_rate * np.sum(hidden_2_delta, axis=0)
self.weights_input_hidden_1 -= self.learning_rate * np.dot(x.T, hidden_1_delta) # (n, 2).T* (n, 4) = (2, 4)
self.bias_input_hidden_1 -= self.learning_rate * np.sum(hidden_1_delta, axis=0)
```

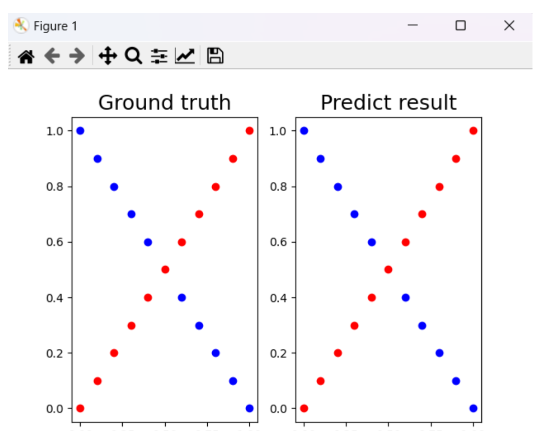
Results of your testing (20%)

a. Screenshot and comparison figure

- Linear



- XOR



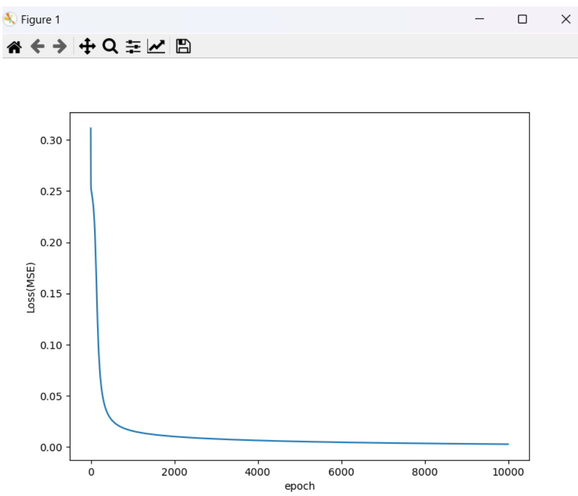
b. Show the accuracy of your prediction

下圖

c. Learning curve (loss, epoch curve)

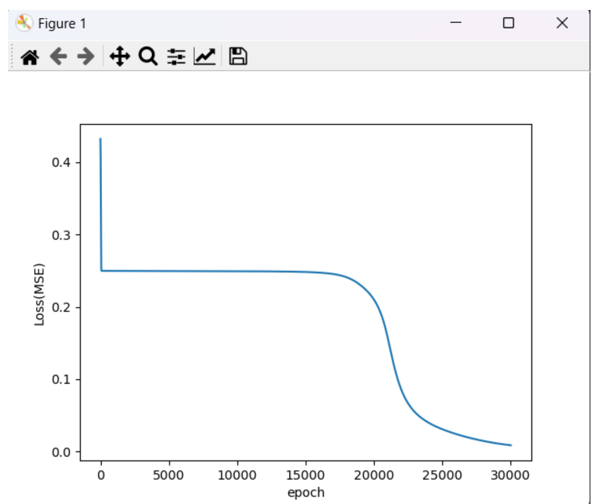
- Linear

```
(base) PS C:\Users\USER\Desktop\交大實
Epoch 0, Loss: 0.3113, Acc: 0.4900
Epoch 2000, Loss: 0.0103, Acc: 1.0000
Epoch 4000, Loss: 0.0065, Acc: 1.0000
Epoch 6000, Loss: 0.0046, Acc: 1.0000
Epoch 8000, Loss: 0.0035, Acc: 1.0000
End training!!!
```



- XOR

```
Epoch 0, Loss: 0.4322, Acc: 0.5238
Epoch 2000, Loss: 0.2496, Acc: 0.5238
Epoch 4000, Loss: 0.2495, Acc: 0.5238
Epoch 6000, Loss: 0.2494, Acc: 0.5238
Epoch 8000, Loss: 0.2494, Acc: 0.5238
Epoch 10000, Loss: 0.2493, Acc: 0.5238
Epoch 12000, Loss: 0.2491, Acc: 0.5238
Epoch 14000, Loss: 0.2487, Acc: 0.5238
Epoch 16000, Loss: 0.2473, Acc: 0.5238
Epoch 18000, Loss: 0.2409, Acc: 0.4762
Epoch 20000, Loss: 0.2098, Acc: 0.5238
Epoch 22000, Loss: 0.0851, Acc: 0.9524
Epoch 24000, Loss: 0.0396, Acc: 1.0000
Epoch 26000, Loss: 0.0241, Acc: 1.0000
Epoch 28000, Loss: 0.0145, Acc: 1.0000
End training!!!
```



可以看出xor在一開始有一小段 訓練不太下去，可能還停留在local minimun，因此較晚收斂到 全域最小值上 ，而 linear在整體訓練的較快最後 loss也更低 。

Discussion (30%)

a. Try different learning rates

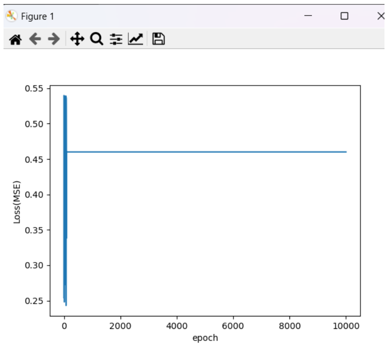
- Linear

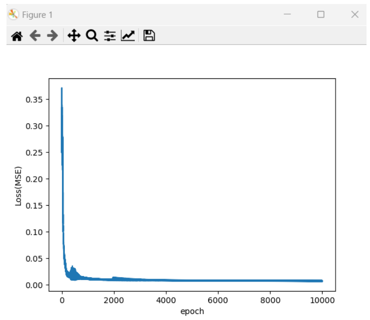
learning_rate = 1　　learning_rate = 0.1　　learning_rate = 0.01

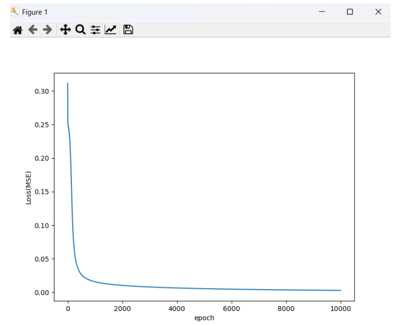
```
Epoch 0, Loss: 0.2540, Acc: 0.5400
Epoch 2000, Loss: 0.4600, Acc: 0.5400
Epoch 4000, Loss: 0.4600, Acc: 0.5400
Epoch 6000, Loss: 0.4600, Acc: 0.5400
Epoch 8000, Loss: 0.4600, Acc: 0.5400
End training!!!
```

```
Epoch 0, Loss: 0.2504, Acc: 0.4500
Epoch 2000, Loss: 0.0084, Acc: 0.9900
Epoch 4000, Loss: 0.0092, Acc: 0.9900
Epoch 6000, Loss: 0.0086, Acc: 0.9900
Epoch 8000, Loss: 0.0082, Acc: 0.9900
End training!!!
```

```
(base) PS C:\Users\USER\Desktop\交大實
Epoch 0, Loss: 0.3113, Acc: 0.4900
Epoch 2000, Loss: 0.0103, Acc: 1.0000
Epoch 4000, Loss: 0.0065, Acc: 1.0000
Epoch 6000, Loss: 0.0046, Acc: 1.0000
Epoch 8000, Loss: 0.0035, Acc: 1.0000
End training!!!
```







- XOR

learning_rate = 1

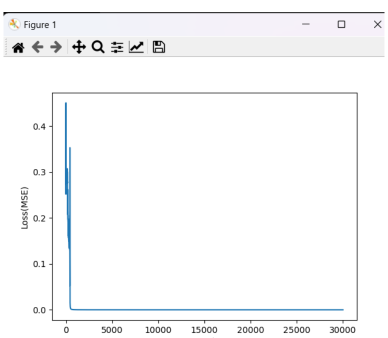
learning_rate = 0.1

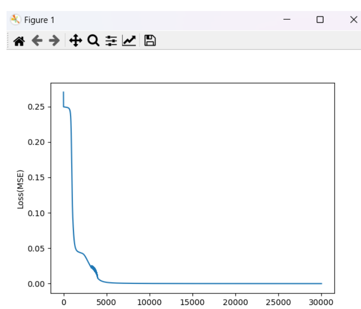
learning_rate = 0.01

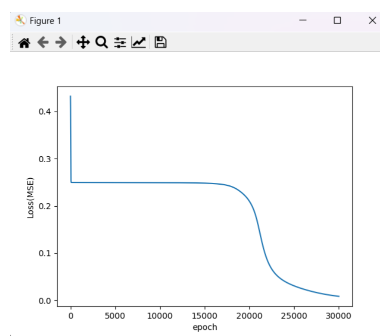
```
Epoch 0, Loss: 0.2801, Acc: 0.5238
Epoch 2000, Loss: 0.0001, Acc: 1.0000
Epoch 4000, Loss: 0.0000, Acc: 1.0000
Epoch 6000, Loss: 0.0000, Acc: 1.0000
Epoch 8000, Loss: 0.0000, Acc: 1.0000
Epoch 10000, Loss: 0.0000, Acc: 1.0000
Epoch 12000, Loss: 0.0000, Acc: 1.0000
Epoch 14000, Loss: 0.0000, Acc: 1.0000
Epoch 16000, Loss: 0.0000, Acc: 1.0000
Epoch 18000, Loss: 0.0000, Acc: 1.0000
Epoch 20000, Loss: 0.0000, Acc: 1.0000
Epoch 22000, Loss: 0.0000, Acc: 1.0000
Epoch 24000, Loss: 0.0000, Acc: 1.0000
Epoch 26000, Loss: 0.0000, Acc: 1.0000
Epoch 28000, Loss: 0.0000, Acc: 1.0000
End training!!!
```

```
Epoch 0, Loss: 0.2703, Acc: 0.5238
Epoch 2000, Loss: 0.0436, Acc: 0.9524
Epoch 4000, Loss: 0.0076, Acc: 1.0000
Epoch 6000, Loss: 0.0009, Acc: 1.0000
Epoch 8000, Loss: 0.0004, Acc: 1.0000
Epoch 10000, Loss: 0.0002, Acc: 1.0000
Epoch 12000, Loss: 0.0002, Acc: 1.0000
Epoch 14000, Loss: 0.0001, Acc: 1.0000
Epoch 16000, Loss: 0.0001, Acc: 1.0000
Epoch 18000, Loss: 0.0001, Acc: 1.0000
Epoch 20000, Loss: 0.0001, Acc: 1.0000
Epoch 22000, Loss: 0.0001, Acc: 1.0000
Epoch 24000, Loss: 0.0001, Acc: 1.0000
Epoch 26000, Loss: 0.0001, Acc: 1.0000
Epoch 28000, Loss: 0.0001, Acc: 1.0000
End training!!!
```

```
Epoch 0, Loss: 0.4322, Acc: 0.5238
Epoch 2000, Loss: 0.2496, Acc: 0.5238
Epoch 4000, Loss: 0.2495, Acc: 0.5238
Epoch 6000, Loss: 0.2494, Acc: 0.5238
Epoch 8000, Loss: 0.2494, Acc: 0.5238
Epoch 10000, Loss: 0.2493, Acc: 0.5238
Epoch 12000, Loss: 0.2491, Acc: 0.5238
Epoch 14000, Loss: 0.2487, Acc: 0.5238
Epoch 16000, Loss: 0.2473, Acc: 0.5238
Epoch 18000, Loss: 0.2409, Acc: 0.4762
Epoch 20000, Loss: 0.2098, Acc: 0.5238
Epoch 22000, Loss: 0.0851, Acc: 0.9524
Epoch 24000, Loss: 0.0396, Acc: 1.0000
Epoch 26000, Loss: 0.0241, Acc: 1.0000
Epoch 28000, Loss: 0.0145, Acc: 1.0000
End training!!!
```







從圖可以看出learning rate 較小時當他找到合適的梯度時他會往該方向去進行更新， 而當 learning rate 較高時震盪也較明顯，當他訓練到一定程度時也會更難降下去甚至訓練不出來。

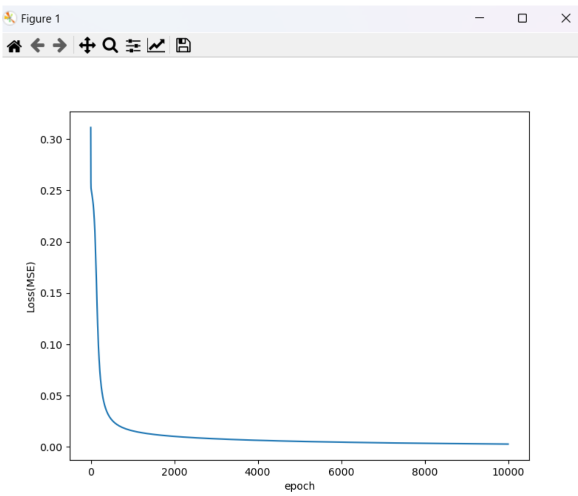
b. Try different numbers of hidden units

- Linear

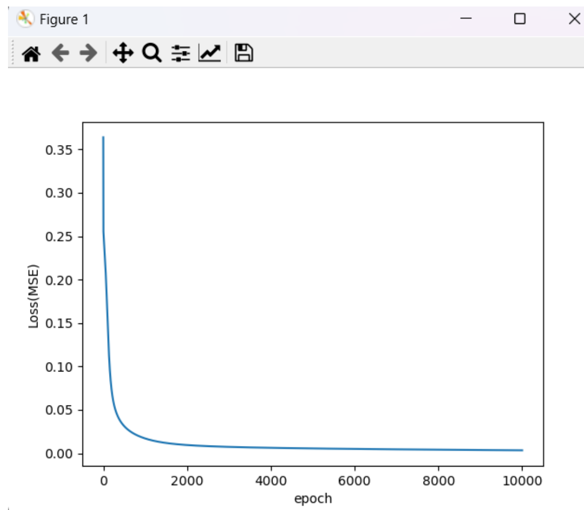
hidden_layer皆為4層

hidden_layer皆為10層

```
(base) PS C:\Users\USER\Desktop\交大實
Epoch 0, Loss: 0.3113, Acc: 0.4900
Epoch 2000, Loss: 0.0103, Acc: 1.0000
Epoch 4000, Loss: 0.0065, Acc: 1.0000
Epoch 6000, Loss: 0.0046, Acc: 1.0000
Epoch 8000, Loss: 0.0035, Acc: 1.0000
End training!!!
```




```
Epoch 0, Loss: 0.3639, Acc: 0.4600
Epoch 2000, Loss: 0.0094, Acc: 0.9900
Epoch 4000, Loss: 0.0062, Acc: 0.9900
Epoch 6000, Loss: 0.0049, Acc: 0.9900
Epoch 8000, Loss: 0.0041, Acc: 1.0000
```



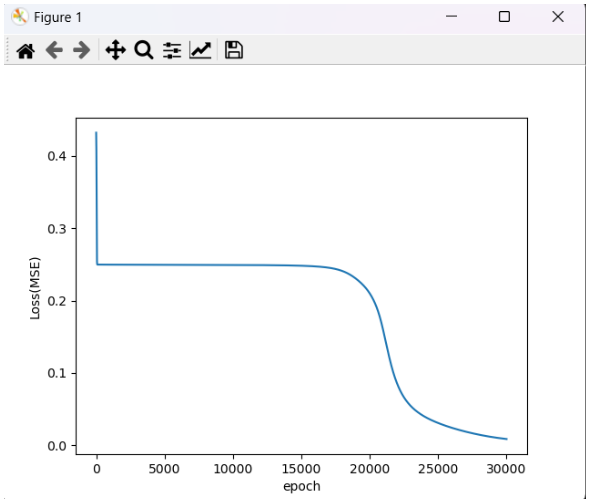
- XOR

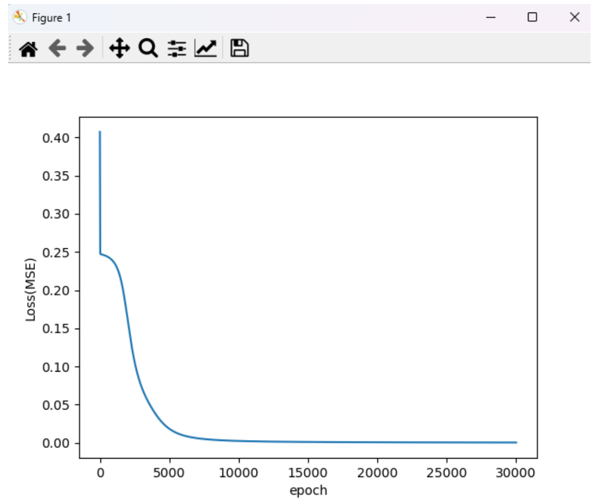
hidden_layer皆為4層

```
Epoch 0, Loss: 0.4322, Acc: 0.5238
Epoch 2000, Loss: 0.2496, Acc: 0.5238
Epoch 4000, Loss: 0.2495, Acc: 0.5238
Epoch 6000, Loss: 0.2494, Acc: 0.5238
Epoch 8000, Loss: 0.2494, Acc: 0.5238
Epoch 10000, Loss: 0.2493, Acc: 0.5238
Epoch 12000, Loss: 0.2491, Acc: 0.5238
Epoch 14000, Loss: 0.2487, Acc: 0.5238
Epoch 16000, Loss: 0.2473, Acc: 0.5238
Epoch 18000, Loss: 0.2409, Acc: 0.4762
Epoch 20000, Loss: 0.2098, Acc: 0.5238
Epoch 22000, Loss: 0.0851, Acc: 0.9524
Epoch 24000, Loss: 0.0396, Acc: 1.0000
Epoch 26000, Loss: 0.0241, Acc: 1.0000
Epoch 28000, Loss: 0.0145, Acc: 1.0000
End training!!!
```

hidden_layer皆為10層

```
Epoch 0, Loss: 0.4072, Acc: 0.4762
Epoch 2000, Loss: 0.1613, Acc: 0.8571
Epoch 4000, Loss: 0.0380, Acc: 1.0000
Epoch 6000, Loss: 0.0098, Acc: 1.0000
Epoch 8000, Loss: 0.0041, Acc: 1.0000
Epoch 10000, Loss: 0.0024, Acc: 1.0000
Epoch 12000, Loss: 0.0016, Acc: 1.0000
Epoch 14000, Loss: 0.0012, Acc: 1.0000
Epoch 16000, Loss: 0.0010, Acc: 1.0000
Epoch 18000, Loss: 0.0008, Acc: 1.0000
Epoch 20000, Loss: 0.0007, Acc: 1.0000
Epoch 22000, Loss: 0.0006, Acc: 1.0000
Epoch 24000, Loss: 0.0005, Acc: 1.0000
Epoch 26000, Loss: 0.0005, Acc: 1.0000
Epoch 28000, Loss: 0.0004, Acc: 1.0000
End training!!!
```





hidden unit越大其下降的速率也稍微更快一點。

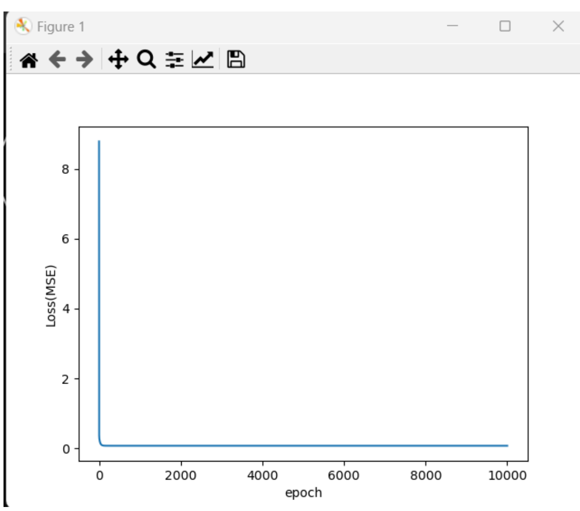
c. Try without activation functions

因為沒有acitve function，所以等於沒有了non-linear的特徵，但是因為

- Linear

此資料是linear分布，所以把leaning_rate調低的情況下還是train得出來。

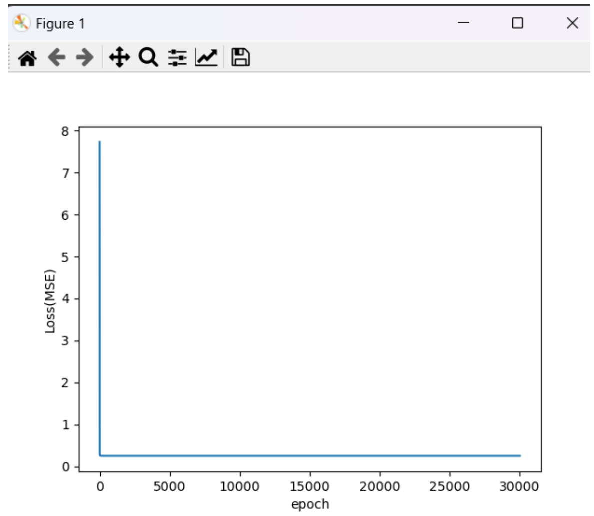
```
Epoch 0, Loss: 8.7840, Acc: 0.5000
Epoch 2000, Loss: 0.0758, Acc: 0.9900
Epoch 4000, Loss: 0.0758, Acc: 0.9900
Epoch 6000, Loss: 0.0758, Acc: 0.9900
Epoch 8000, Loss: 0.0758, Acc: 0.9900
End training!!!
```



- XOR

此資料為非線性分布所以不可能train出來。

```
Epoch 0, Loss: 7.7320, Acc: 0.4762
Epoch 2000, Loss: 0.2494, Acc: 0.5238
Epoch 4000, Loss: 0.2494, Acc: 0.5238
Epoch 6000, Loss: 0.2494, Acc: 0.5238
Epoch 8000, Loss: 0.2494, Acc: 0.5238
Epoch 10000, Loss: 0.2494, Acc: 0.5238
Epoch 12000, Loss: 0.2494, Acc: 0.5238
Epoch 14000, Loss: 0.2494, Acc: 0.5238
Epoch 16000, Loss: 0.2494, Acc: 0.5238
Epoch 18000, Loss: 0.2494, Acc: 0.5238
Epoch 20000, Loss: 0.2494, Acc: 0.5238
Epoch 22000, Loss: 0.2494, Acc: 0.5238
Epoch 24000, Loss: 0.2494, Acc: 0.5238
Epoch 26000, Loss: 0.2494, Acc: 0.5238
Epoch 28000, Loss: 0.2494, Acc: 0.5238
End training!!!
```



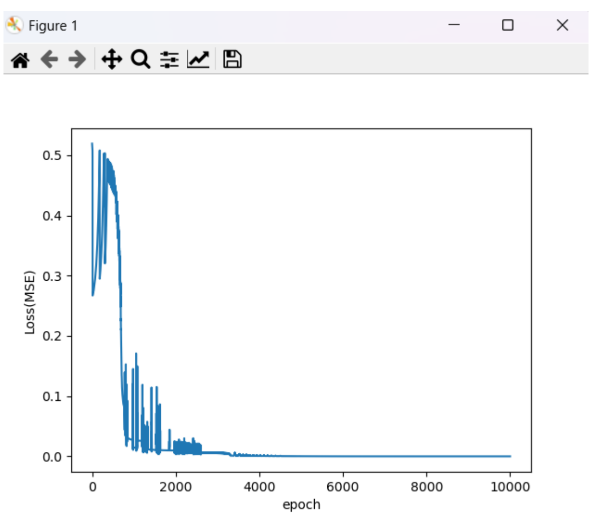
Extra (10%)

B. Implement different activation functions. (3%)

使用leaky_relu去進行測試比對， 其 LOSS與分類結果如下：relu對訓練較快!

- Linear

```
Epoch 0, Loss: 0.5193, Acc: 0.4900
Epoch 2000, Loss: 0.0240, Acc: 0.9700
Epoch 4000, Loss: 0.0007, Acc: 1.0000
Epoch 6000, Loss: 0.0001, Acc: 1.0000
Epoch 8000, Loss: 0.0000, Acc: 1.0000
End training!!!
```



- XOR

```
Epoch 0, Loss: 0.3055, Acc: 0.4762
Epoch 2000, Loss: 0.0434, Acc: 0.9524
Epoch 4000, Loss: 0.0434, Acc: 0.9524
Epoch 6000, Loss: 0.0434, Acc: 0.9524
Epoch 8000, Loss: 0.0434, Acc: 0.9524
Epoch 10000, Loss: 0.0434, Acc: 0.9524
Epoch 12000, Loss: 0.0434, Acc: 0.9524
Epoch 14000, Loss: 0.0434, Acc: 0.9524
Epoch 16000, Loss: 0.0434, Acc: 0.9524
Epoch 18000, Loss: 0.0434, Acc: 0.9524
Epoch 20000, Loss: 0.0434, Acc: 0.9524
Epoch 22000, Loss: 0.0434, Acc: 0.9524
Epoch 24000, Loss: 0.0434, Acc: 0.9524
Epoch 26000, Loss: 0.0434, Acc: 0.9524
Epoch 28000, Loss: 0.0434, Acc: 0.9524
End training!!!
```

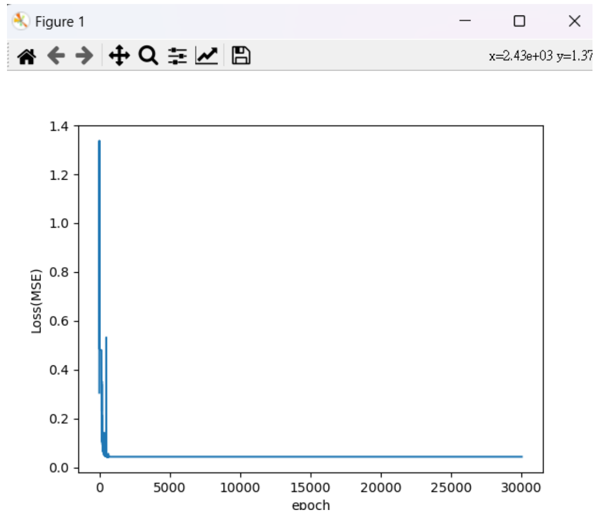


Figure 1
x=2.43e+03 y=1.37
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Loss(MSE)
0
5000
10000
15000
20000
25000
30000
epoch