Lab1

313605019 方敏

1. Introduction

本次實驗的目標是實作多層感知機（MLP）並使用 Backpropagation 進行訓練，並透過梯度下降（SGD）優化模型參數，以學習不同數據集的分類。模型需要包含兩層 hidden layers，並用不同優化器進行權重更新，以分析不同參數對訓練效果的影響。

實驗內容

(1) 實作前向與反向傳播：
- 建立可調整 hidden layers 數量的 MLP。
- 計算 Forward Propagation 以獲得模型輸出。
- 計算 Backward Propagation 以獲取梯度並更新權重。

(2) 模型訓練與優化：
- 使用不同優化器，並比較訓練收斂速度與準確度。
- 嘗試不同學習率，分析對模型學習的影響。
- 嘗試不同 hidden units 找出最佳結構。
- 測試不使用 Activation Functions 觀察影響。

(3) 測試不同數據集：
- 訓練 MLP 學習線性可分數據。
- 訓練 MLP 解決非線性分類（XOR）。

2. Implementation Details:

A. Sigmoid function

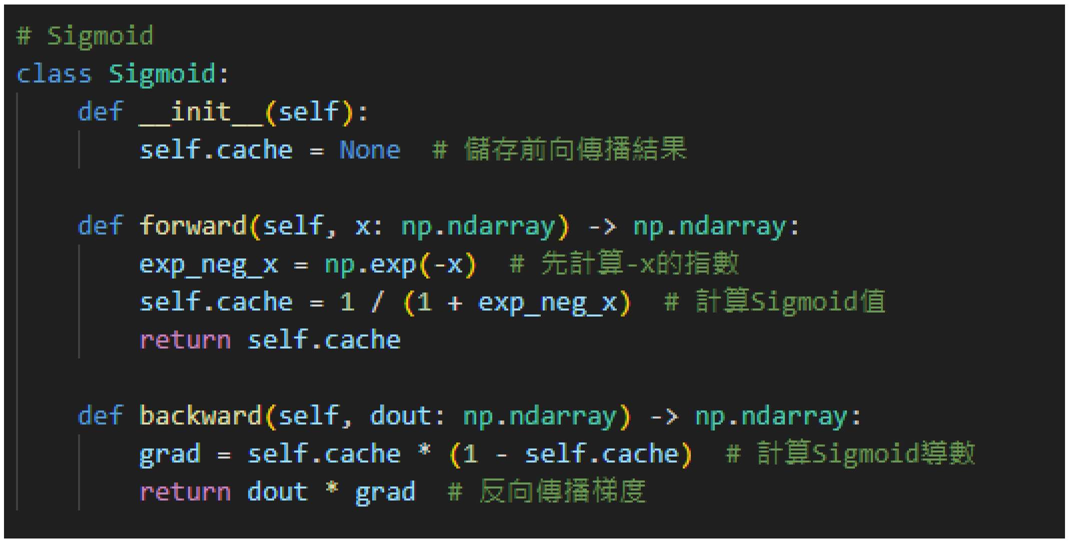
Sigmoid 函數是一種常見的 Activation Function，其數學表達式為：

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

Sigmoid 函數的輸出值介於 (0,1) 之間，適合做為二元分類問題，函數平滑且可微，適用於梯度下降法。

Sigmoid 導數為：

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

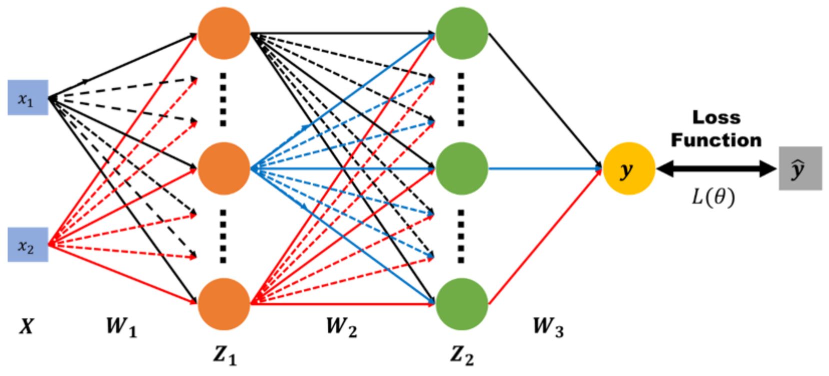


```python
# Sigmoid
class Sigmoid:
    def __init__(self):
        self.cache = None  # 儲存前向傳播結果

    def forward(self, x: np.ndarray) -> np.ndarray:
        exp_neg_x = np.exp(-x)  # 先計算-x的指數
        self.cache = 1 / (1 + exp_neg_x)  # 計算Sigmoid值
        return self.cache

    def backward(self, dout: np.ndarray) -> np.ndarray:
        grad = self.cache * (1 - self.cache)  # 計算Sigmoid導數
        return dout * grad  # 反向傳播梯度
```

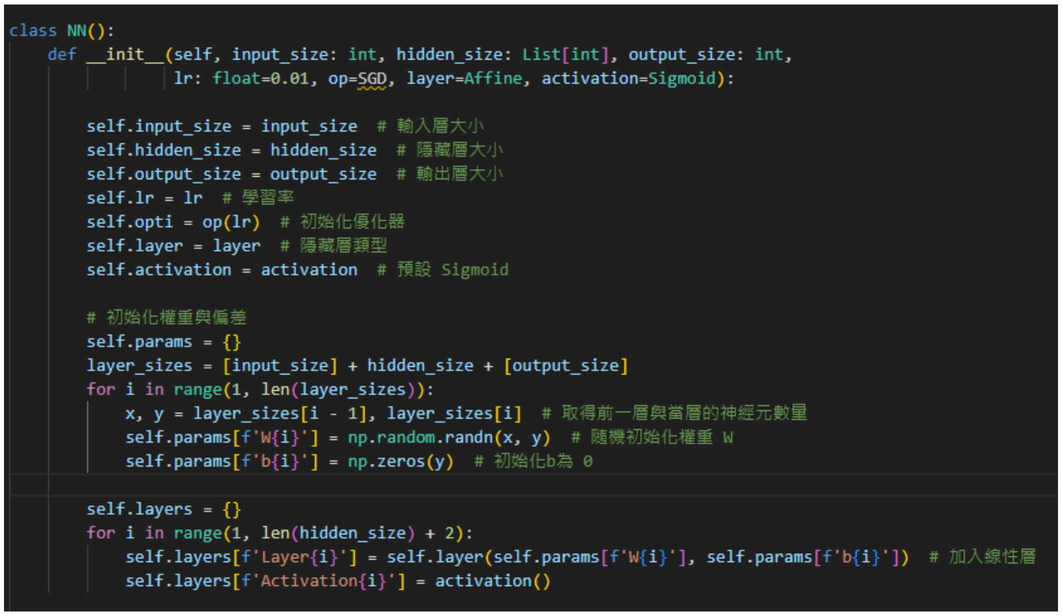
在實作中我定義了一個 Sigmoid 類別，self.cache 儲存 Sigmoid 的輸出值，forward()計算 Sigmoid 函數，backward()計算 Sigmoid 導數

B. Neural network architecture

實作了一個具有兩層隱藏層的多層感知機（MLP），架構如下：



Input Layer → Hidden Layer 1 → Hidden Layer 2 → Output Layer

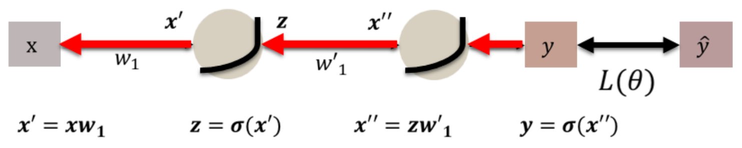


```python
class NN():
    def __init__(self, input_size: int, hidden_size: List[int], output_size: int,
                 lr: float=0.01, op=SGD, layer=Affine, activation=Sigmoid):

        self.input_size = input_size  # 輸入層大小
        self.hidden_size = hidden_size  # 隱藏層大小
        self.output_size = output_size  # 輸出層大小
        self.lr = lr  # 學習率
        self.opti = op(lr)  # 初始化優化器
        self.layer = layer  # 隱藏層類型
        self.activation = activation  # 預設 Sigmoid

        # 初始化權重與偏差
        self.params = {}
        layer_sizes = [input_size] + hidden_size + [output_size]
        for i in range(1, len(layer_sizes)):
            x, y = layer_sizes[i - 1], layer_sizes[i]  # 取得前一層與當層的神經元數量
            self.params[f'W{i}'] = np.random.randn(x, y)  # 隨機初始化權重 W
            self.params[f'b{i}'] = np.zeros(y)  # 初始化b為 0

        self.layers = {}
        for i in range(1, len(hidden_size) + 2):
            self.layers[f'Layer{i}'] = self.layer(self.params[f'W{i}'], self.params[f'b{i}'])  # 加入線性層
            self.layers[f'Activation{i}'] = activation()
```

在實作中定義一個可調整結構的多層感知機（MLP），包含輸入層、兩層隱藏層與輸出層，首先初始化權重跟偏差，權重使用標準正態分佈隨機初始化，偏差則設為 0。接著根據 hidden_size 建立每層 Affine（全連接層）與 Activation，並存入 self.layers。

C. Backpropagation

Backpropagation 用於最小化 Loss Function，主要目標是根據 Loss Function 的梯度來調整權重，改善模型的預測能力。



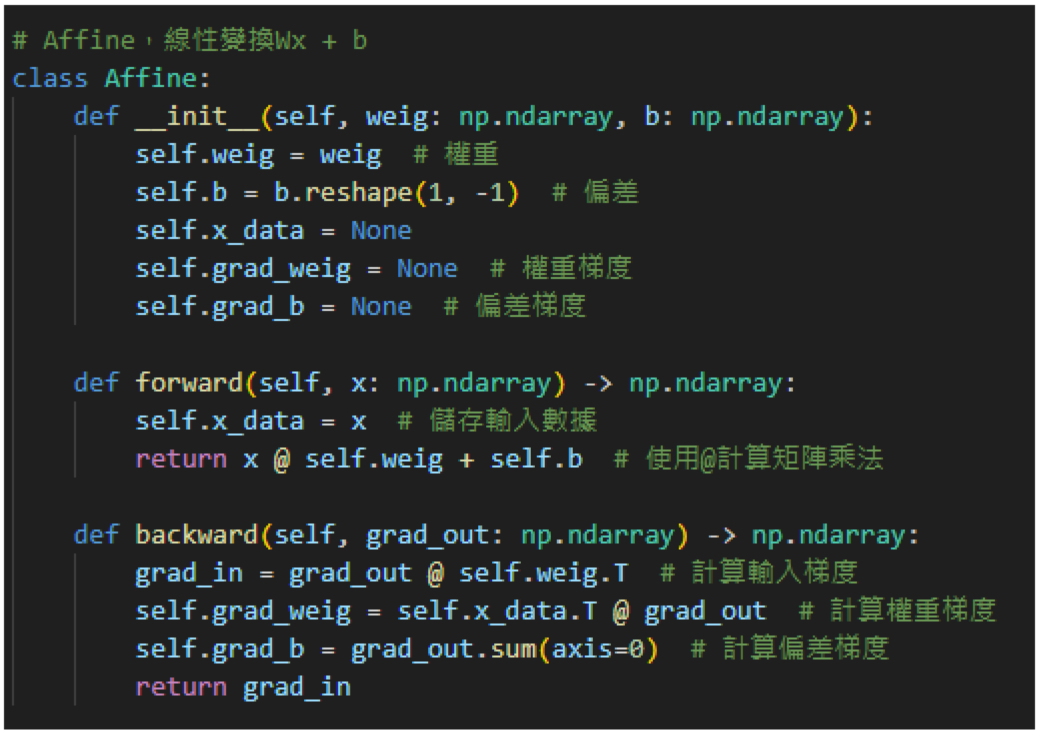
Chain rule

$y = g(x) \quad z = h(y)$

$$\mathbf{x} \overset{\mathbf{g}()}{\rightarrow} \boldsymbol{y} \overset{\boldsymbol{h}()}{\rightarrow} \mathbf{z} \qquad \frac{dz}{dx} = \frac{dz}{\cancel{dy}}\frac{\cancel{dy}}{dx}$$

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

先用 Forward Propagation 計算模型的輸出，再計算損失函數對輸出的梯度，再用 Chain Rule 從輸出層往前計算梯度，最後使用梯度下降法來更新權重。

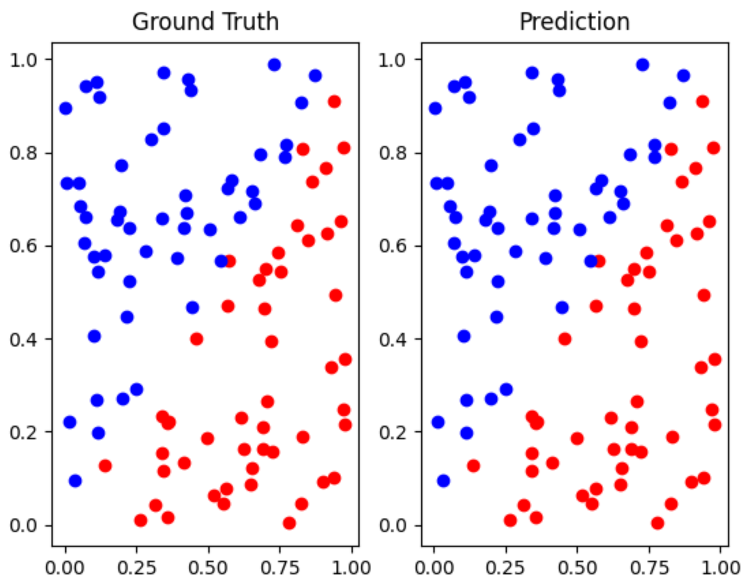


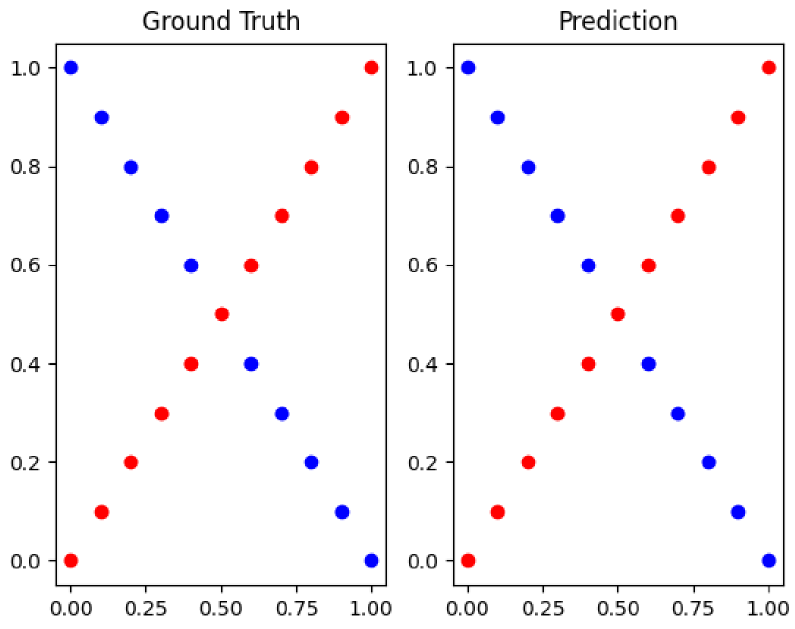
```python
# Affine，線性變換Wx + b
class Affine:
    def __init__(self, weig: np.ndarray, b: np.ndarray):
        self.weig = weig  # 權重
        self.b = b.reshape(1, -1)  # 偏差
        self.x_data = None
        self.grad_weig = None  # 權重梯度
        self.grad_b = None  # 偏差梯度

    def forward(self, x: np.ndarray) -> np.ndarray:
        self.x_data = x  # 儲存輸入數據
        return x @ self.weig + self.b  # 使用@計算矩陣乘法

    def backward(self, grad_out: np.ndarray) -> np.ndarray:
        grad_in = grad_out @ self.weig.T  # 計算輸入梯度
        self.grad_weig = self.x_data.T @ grad_out  # 計算權重梯度
        self.grad_b = grad_out.sum(axis=0)  # 計算偏差梯度
        return grad_in
```

在實作中用 Affine 類別實作全連接層，負責線性變換。Forward Propagation 時，計算輸入與權重的矩陣乘積，加上偏差 b 後即為 $Wx+b$，並儲存在 x_data 裡方便 Backpropagation 使用。在 Backpropagation 時，根據 Chain Rule 計算梯度，包括輸入梯度（grad_in）、權重梯度（grad_weig）與偏差梯度（grad_b），並將其回傳給前一層，以更新權重並找到最佳參數。

3. Experimental Results

A. Screenshot and comparison figure





B. Show the accuracy of your prediction
(achieve 90% accuracy)

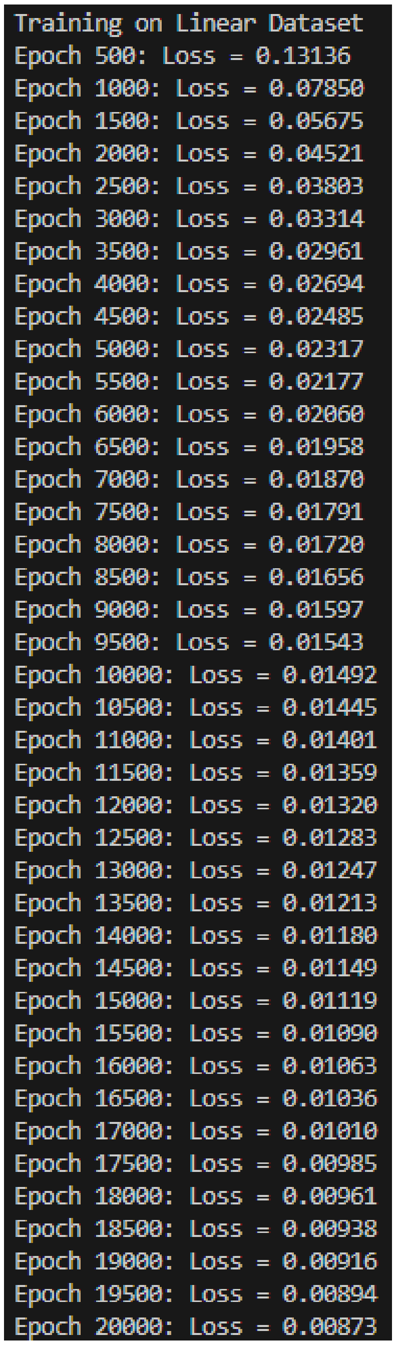
Parameters:

epochs = 20000 、learning rate = 0.1

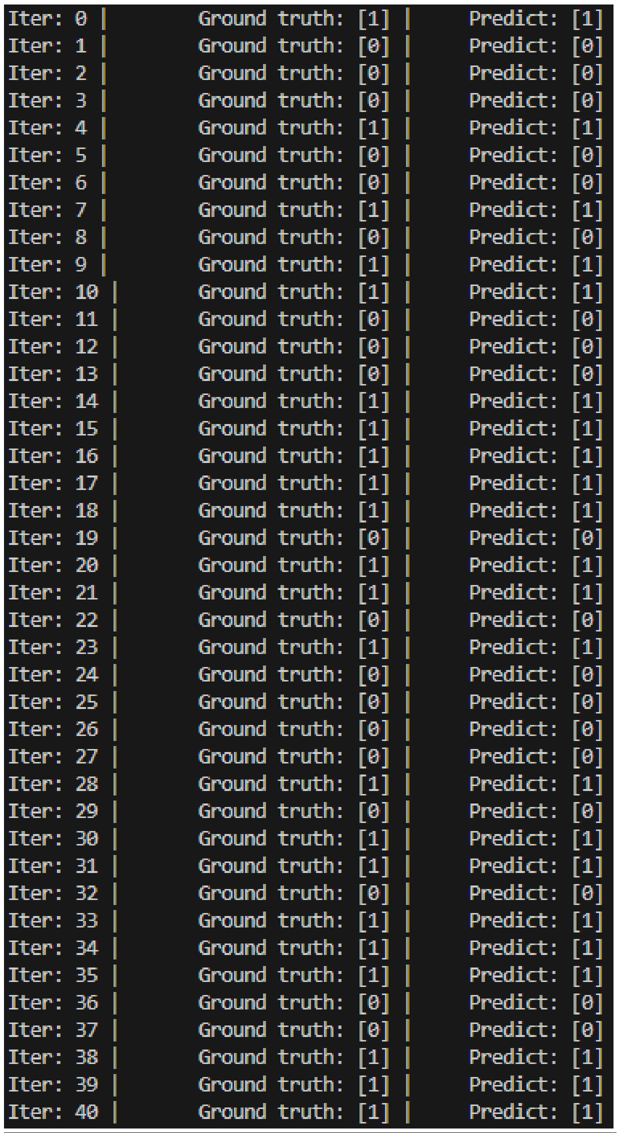
hidden layers = 2 、hidden units = [10,10]

optimizer = SGD 、layer = Affine 、activation layer = Sigmoid

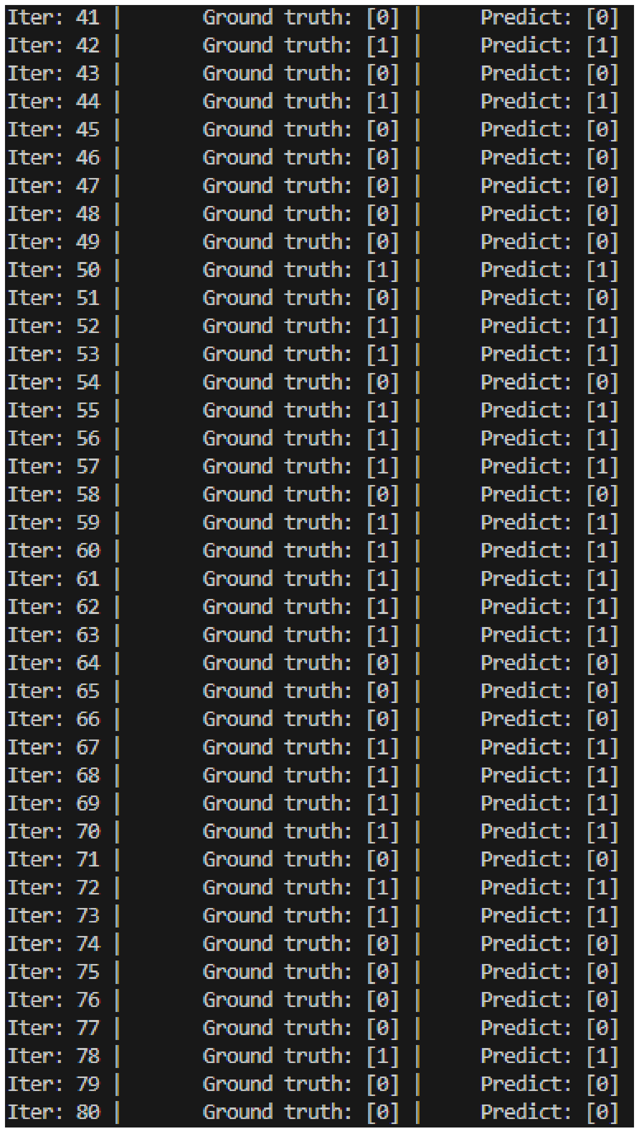
Linear



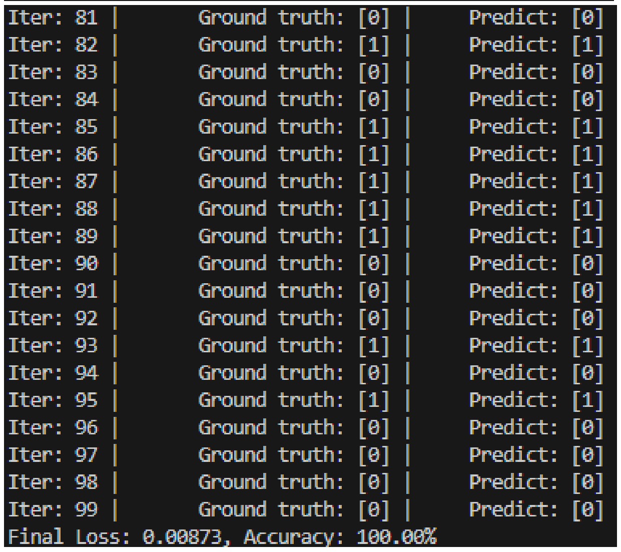
```
Training on Linear Dataset
Epoch 500: Loss = 0.13136
Epoch 1000: Loss = 0.07850
Epoch 1500: Loss = 0.05675
Epoch 2000: Loss = 0.04521
Epoch 2500: Loss = 0.03803
Epoch 3000: Loss = 0.03314
Epoch 3500: Loss = 0.02961
Epoch 4000: Loss = 0.02694
Epoch 4500: Loss = 0.02485
Epoch 5000: Loss = 0.02317
Epoch 5500: Loss = 0.02177
Epoch 6000: Loss = 0.02060
Epoch 6500: Loss = 0.01958
Epoch 7000: Loss = 0.01870
Epoch 7500: Loss = 0.01791
Epoch 8000: Loss = 0.01720
Epoch 8500: Loss = 0.01656
Epoch 9000: Loss = 0.01597
Epoch 9500: Loss = 0.01543
Epoch 10000: Loss = 0.01492
Epoch 10500: Loss = 0.01445
Epoch 11000: Loss = 0.01401
Epoch 11500: Loss = 0.01359
Epoch 12000: Loss = 0.01320
Epoch 12500: Loss = 0.01283
Epoch 13000: Loss = 0.01247
Epoch 13500: Loss = 0.01213
Epoch 14000: Loss = 0.01180
Epoch 14500: Loss = 0.01149
Epoch 15000: Loss = 0.01119
Epoch 15500: Loss = 0.01090
Epoch 16000: Loss = 0.01063
Epoch 16500: Loss = 0.01036
Epoch 17000: Loss = 0.01010
Epoch 17500: Loss = 0.00985
Epoch 18000: Loss = 0.00961
Epoch 18500: Loss = 0.00938
Epoch 19000: Loss = 0.00916
Epoch 19500: Loss = 0.00894
Epoch 20000: Loss = 0.00873
```



```
Iter: 0 |        Ground truth: [1] |      Predict: [1]
Iter: 1 |        Ground truth: [0] |      Predict: [0]
Iter: 2 |        Ground truth: [0] |      Predict: [0]
Iter: 3 |        Ground truth: [0] |      Predict: [0]
Iter: 4 |        Ground truth: [1] |      Predict: [1]
Iter: 5 |        Ground truth: [0] |      Predict: [0]
Iter: 6 |        Ground truth: [0] |      Predict: [0]
Iter: 7 |        Ground truth: [1] |      Predict: [1]
Iter: 8 |        Ground truth: [0] |      Predict: [0]
Iter: 9 |        Ground truth: [1] |      Predict: [1]
Iter: 10 |       Ground truth: [1] |      Predict: [1]
Iter: 11 |       Ground truth: [0] |      Predict: [0]
Iter: 12 |       Ground truth: [0] |      Predict: [0]
Iter: 13 |       Ground truth: [0] |      Predict: [0]
Iter: 14 |       Ground truth: [1] |      Predict: [1]
Iter: 15 |       Ground truth: [1] |      Predict: [1]
Iter: 16 |       Ground truth: [1] |      Predict: [1]
Iter: 17 |       Ground truth: [1] |      Predict: [1]
Iter: 18 |       Ground truth: [1] |      Predict: [1]
Iter: 19 |       Ground truth: [0] |      Predict: [0]
Iter: 20 |       Ground truth: [1] |      Predict: [1]
Iter: 21 |       Ground truth: [1] |      Predict: [1]
Iter: 22 |       Ground truth: [0] |      Predict: [0]
Iter: 23 |       Ground truth: [1] |      Predict: [1]
Iter: 24 |       Ground truth: [0] |      Predict: [0]
Iter: 25 |       Ground truth: [0] |      Predict: [0]
Iter: 26 |       Ground truth: [0] |      Predict: [0]
Iter: 27 |       Ground truth: [0] |      Predict: [0]
Iter: 28 |       Ground truth: [1] |      Predict: [1]
Iter: 29 |       Ground truth: [0] |      Predict: [0]
Iter: 30 |       Ground truth: [1] |      Predict: [1]
Iter: 31 |       Ground truth: [1] |      Predict: [1]
Iter: 32 |       Ground truth: [0] |      Predict: [0]
Iter: 33 |       Ground truth: [1] |      Predict: [1]
Iter: 34 |       Ground truth: [1] |      Predict: [1]
Iter: 35 |       Ground truth: [1] |      Predict: [1]
Iter: 36 |       Ground truth: [0] |      Predict: [0]
Iter: 37 |       Ground truth: [0] |      Predict: [0]
Iter: 38 |       Ground truth: [1] |      Predict: [1]
Iter: 39 |       Ground truth: [1] |      Predict: [1]
Iter: 40 |       Ground truth: [1] |      Predict: [1]
```

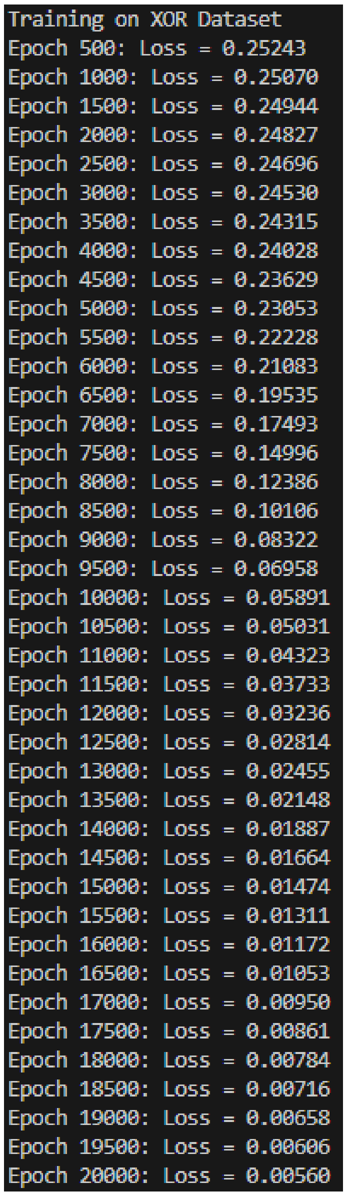


```
Iter: 41 |       Ground truth: [0] |      Predict: [0]
Iter: 42 |       Ground truth: [1] |      Predict: [1]
Iter: 43 |       Ground truth: [0] |      Predict: [0]
Iter: 44 |       Ground truth: [1] |      Predict: [1]
Iter: 45 |       Ground truth: [0] |      Predict: [0]
Iter: 46 |       Ground truth: [0] |      Predict: [0]
Iter: 47 |       Ground truth: [0] |      Predict: [0]
Iter: 48 |       Ground truth: [0] |      Predict: [0]
Iter: 49 |       Ground truth: [0] |      Predict: [0]
Iter: 50 |       Ground truth: [1] |      Predict: [1]
Iter: 51 |       Ground truth: [0] |      Predict: [0]
Iter: 52 |       Ground truth: [1] |      Predict: [1]
Iter: 53 |       Ground truth: [1] |      Predict: [1]
Iter: 54 |       Ground truth: [0] |      Predict: [0]
Iter: 55 |       Ground truth: [1] |      Predict: [1]
Iter: 56 |       Ground truth: [1] |      Predict: [1]
Iter: 57 |       Ground truth: [1] |      Predict: [1]
Iter: 58 |       Ground truth: [0] |      Predict: [0]
Iter: 59 |       Ground truth: [1] |      Predict: [1]
Iter: 60 |       Ground truth: [1] |      Predict: [1]
Iter: 61 |       Ground truth: [1] |      Predict: [1]
Iter: 62 |       Ground truth: [1] |      Predict: [1]
Iter: 63 |       Ground truth: [1] |      Predict: [1]
Iter: 64 |       Ground truth: [0] |      Predict: [0]
Iter: 65 |       Ground truth: [0] |      Predict: [0]
Iter: 66 |       Ground truth: [0] |      Predict: [0]
Iter: 67 |       Ground truth: [1] |      Predict: [1]
Iter: 68 |       Ground truth: [1] |      Predict: [1]
Iter: 69 |       Ground truth: [1] |      Predict: [1]
Iter: 70 |       Ground truth: [1] |      Predict: [1]
Iter: 71 |       Ground truth: [0] |      Predict: [0]
Iter: 72 |       Ground truth: [1] |      Predict: [1]
Iter: 73 |       Ground truth: [1] |      Predict: [1]
Iter: 74 |       Ground truth: [0] |      Predict: [0]
Iter: 75 |       Ground truth: [0] |      Predict: [0]
Iter: 76 |       Ground truth: [0] |      Predict: [0]
Iter: 77 |       Ground truth: [0] |      Predict: [0]
Iter: 78 |       Ground truth: [1] |      Predict: [1]
Iter: 79 |       Ground truth: [0] |      Predict: [0]
Iter: 80 |       Ground truth: [0] |      Predict: [0]
```

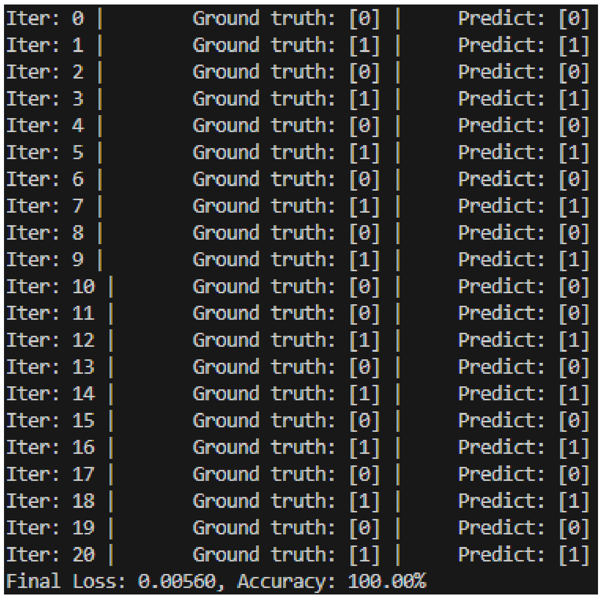


```
Iter: 81 |       Ground truth: [0] |      Predict: [0]
Iter: 82 |       Ground truth: [1] |      Predict: [1]
Iter: 83 |       Ground truth: [0] |      Predict: [0]
Iter: 84 |       Ground truth: [0] |      Predict: [0]
Iter: 85 |       Ground truth: [1] |      Predict: [1]
Iter: 86 |       Ground truth: [1] |      Predict: [1]
Iter: 87 |       Ground truth: [1] |      Predict: [1]
Iter: 88 |       Ground truth: [1] |      Predict: [1]
Iter: 89 |       Ground truth: [1] |      Predict: [1]
Iter: 90 |       Ground truth: [0] |      Predict: [0]
Iter: 91 |       Ground truth: [0] |      Predict: [0]
Iter: 92 |       Ground truth: [0] |      Predict: [0]
Iter: 93 |       Ground truth: [1] |      Predict: [1]
Iter: 94 |       Ground truth: [0] |      Predict: [0]
Iter: 95 |       Ground truth: [1] |      Predict: [1]
Iter: 96 |       Ground truth: [0] |      Predict: [0]
Iter: 97 |       Ground truth: [0] |      Predict: [0]
Iter: 98 |       Ground truth: [0] |      Predict: [0]
Iter: 99 |       Ground truth: [0] |      Predict: [0]
Final Loss: 0.00873, Accuracy: 100.00%
```

XOR



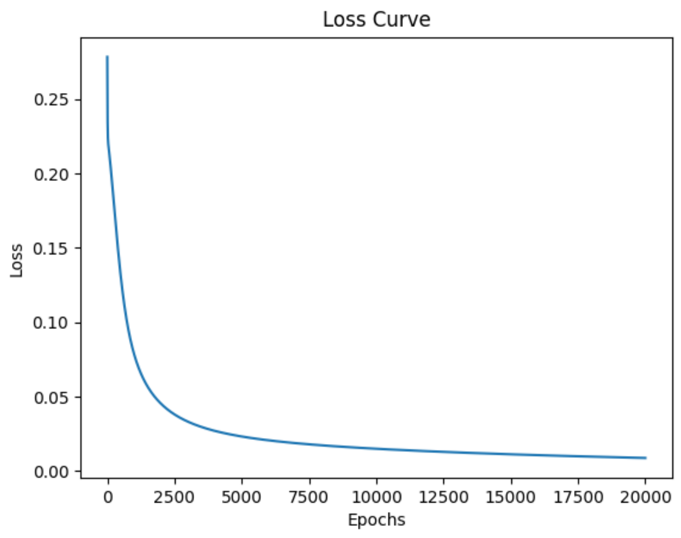
```
Training on XOR Dataset
Epoch 500: Loss = 0.25243
Epoch 1000: Loss = 0.25070
Epoch 1500: Loss = 0.24944
Epoch 2000: Loss = 0.24827
Epoch 2500: Loss = 0.24696
Epoch 3000: Loss = 0.24530
Epoch 3500: Loss = 0.24315
Epoch 4000: Loss = 0.24028
Epoch 4500: Loss = 0.23629
Epoch 5000: Loss = 0.23053
Epoch 5500: Loss = 0.22228
Epoch 6000: Loss = 0.21083
Epoch 6500: Loss = 0.19535
Epoch 7000: Loss = 0.17493
Epoch 7500: Loss = 0.14996
Epoch 8000: Loss = 0.12386
Epoch 8500: Loss = 0.10106
Epoch 9000: Loss = 0.08322
Epoch 9500: Loss = 0.06958
Epoch 10000: Loss = 0.05891
Epoch 10500: Loss = 0.05031
Epoch 11000: Loss = 0.04323
Epoch 11500: Loss = 0.03733
Epoch 12000: Loss = 0.03236
Epoch 12500: Loss = 0.02814
Epoch 13000: Loss = 0.02455
Epoch 13500: Loss = 0.02148
Epoch 14000: Loss = 0.01887
Epoch 14500: Loss = 0.01664
Epoch 15000: Loss = 0.01474
Epoch 15500: Loss = 0.01311
Epoch 16000: Loss = 0.01172
Epoch 16500: Loss = 0.01053
Epoch 17000: Loss = 0.00950
Epoch 17500: Loss = 0.00861
Epoch 18000: Loss = 0.00784
Epoch 18500: Loss = 0.00716
Epoch 19000: Loss = 0.00658
Epoch 19500: Loss = 0.00606
Epoch 20000: Loss = 0.00560
```



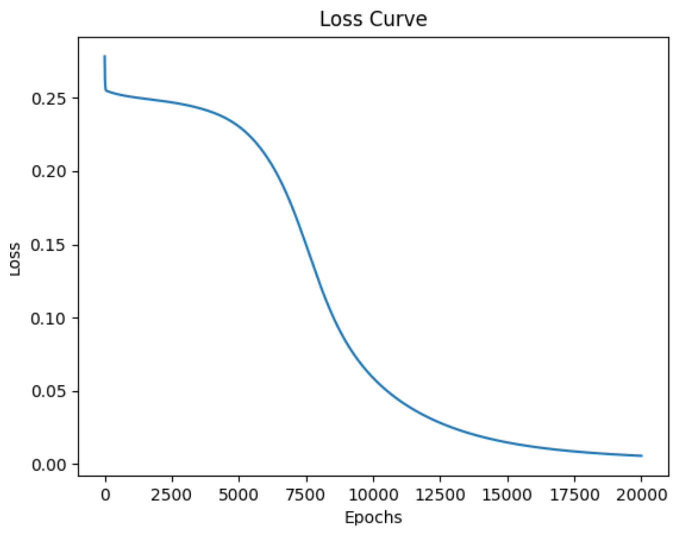
```
Iter: 0 |       Ground truth: [0] |     Predict: [0]
Iter: 1 |       Ground truth: [1] |     Predict: [1]
Iter: 2 |       Ground truth: [0] |     Predict: [0]
Iter: 3 |       Ground truth: [1] |     Predict: [1]
Iter: 4 |       Ground truth: [0] |     Predict: [0]
Iter: 5 |       Ground truth: [1] |     Predict: [1]
Iter: 6 |       Ground truth: [0] |     Predict: [0]
Iter: 7 |       Ground truth: [1] |     Predict: [1]
Iter: 8 |       Ground truth: [0] |     Predict: [0]
Iter: 9 |       Ground truth: [1] |     Predict: [1]
Iter: 10 |      Ground truth: [0] |     Predict: [0]
Iter: 11 |      Ground truth: [0] |     Predict: [0]
Iter: 12 |      Ground truth: [1] |     Predict: [1]
Iter: 13 |      Ground truth: [0] |     Predict: [0]
Iter: 14 |      Ground truth: [1] |     Predict: [1]
Iter: 15 |      Ground truth: [0] |     Predict: [0]
Iter: 16 |      Ground truth: [1] |     Predict: [1]
Iter: 17 |      Ground truth: [0] |     Predict: [0]
Iter: 18 |      Ground truth: [1] |     Predict: [1]
Iter: 19 |      Ground truth: [0] |     Predict: [0]
Iter: 20 |      Ground truth: [1] |     Predict: [1]
Final Loss: 0.00560, Accuracy: 100.00%
```

C. Learning curve (loss-epoch curve)

- Linear



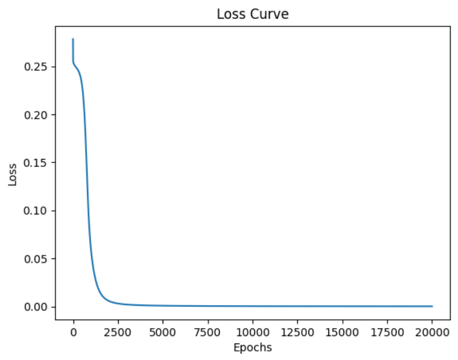
- XOR



D. Anything you want to share

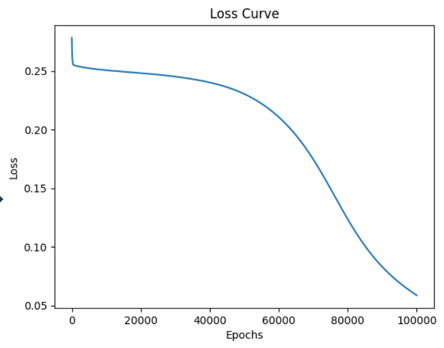
導致這個 loss 曲線可能是因為 XOR 問題是非線性，Sigmoid 在初期輸出接近 0.5，導致梯度較小，學習進度變慢，權重更新較小，模型仍在尋找適合的權重組合，因此初期 loss 下降速度較慢，中後期權重開始收斂到較好的值。
解決方法：將 learning rate 提高至 1 收斂速度較快。

反之，當我將 learning rate 從 0.1 -> 0.01，XOR 的正確率只有 52.38%，若將 epochs 拉高至 5 倍 100000 後，正確率達到 95.24% 但還不是 100%，可知 learning rate 設定（尤其對非線性）很重要。



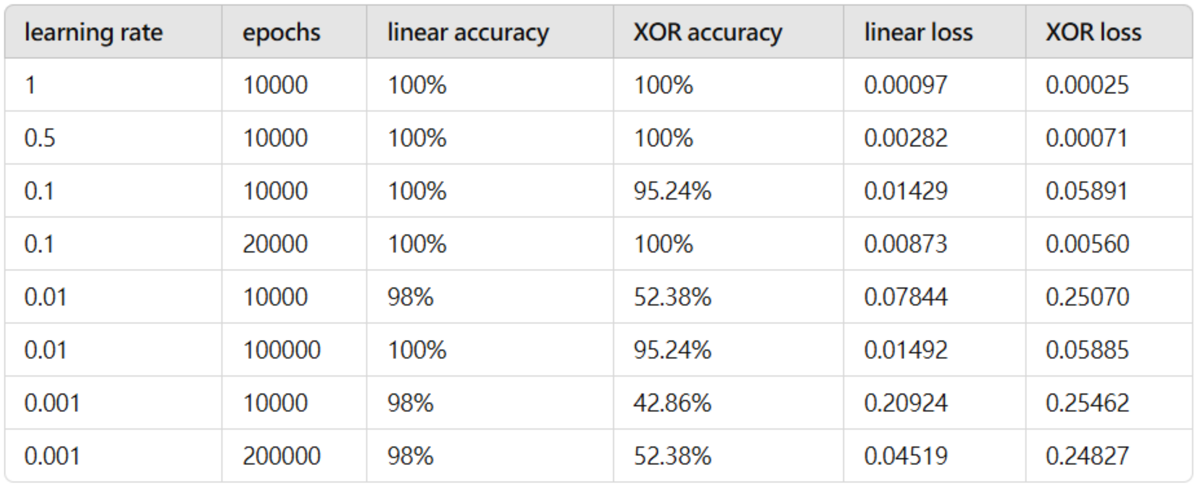
Learning rate = 1
Epochs = 20000

Learning rate = 0.01
Epochs = 100000



4. Discussion

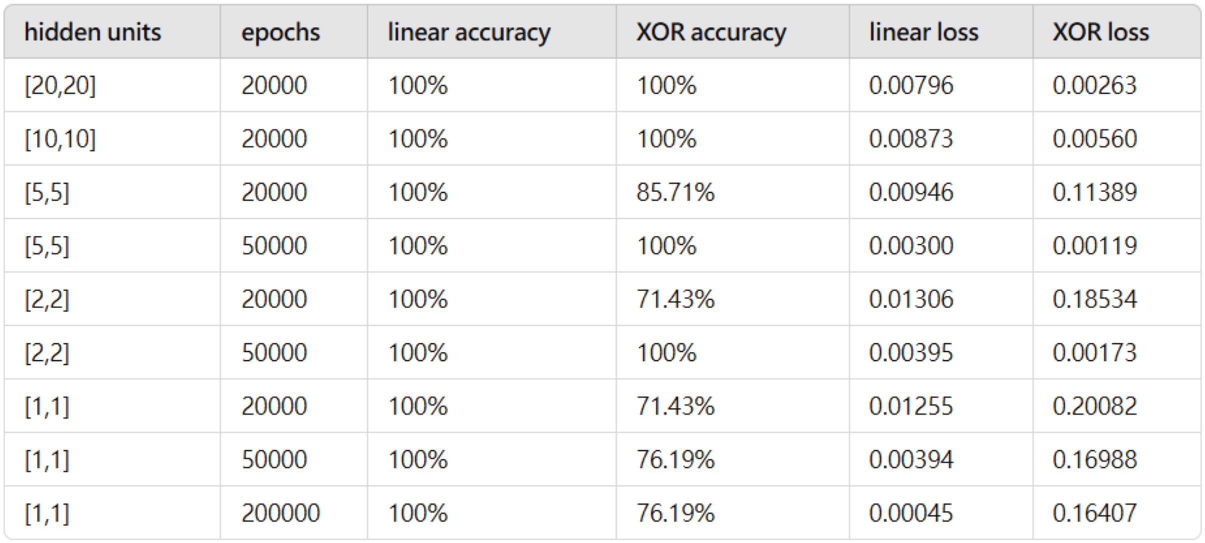
A. Try different learning rates



learning rate	epochs	linear accuracy	XOR accuracy	linear loss	XOR loss
1	10000	100%	100%	0.00097	0.00025
0.5	10000	100%	100%	0.00282	0.00071
0.1	10000	100%	95.24%	0.01429	0.05891
0.1	20000	100%	100%	0.00873	0.00560
0.01	10000	98%	52.38%	0.07844	0.25070
0.01	100000	100%	95.24%	0.01492	0.05885
0.001	10000	98%	42.86%	0.20924	0.25462
0.001	200000	98%	52.38%	0.04519	0.24827

線性問題無論 learning rate 如何調整，正確率幾乎都能達到 100%，loss 也非常小，得知線性問題對於 MLP 是輕鬆的；非線性問題比較複雜，learning rate 較高時，訓練更快收斂正確率也較高，但較低的 learning rate 需要更多（甚至數十倍以上）epochs 來達到理想結果。

B. Try different numbers of hidden units



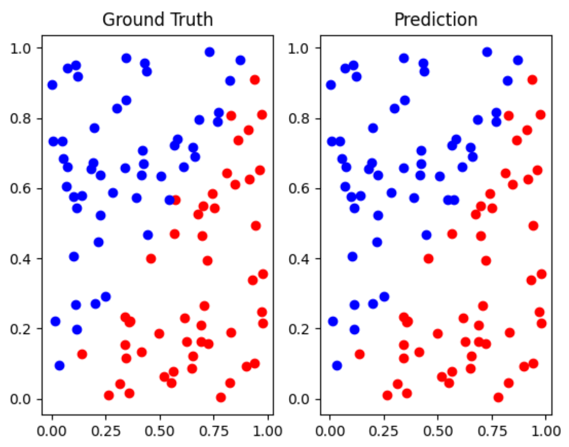
hidden units	epochs	linear accuracy	XOR accuracy	linear loss	XOR loss
[20,20]	20000	100%	100%	0.00796	0.00263
[10,10]	20000	100%	100%	0.00873	0.00560
[5,5]	20000	100%	85.71%	0.00946	0.11389
[5,5]	50000	100%	100%	0.00300	0.00119
[2,2]	20000	100%	71.43%	0.01306	0.18534
[2,2]	50000	100%	100%	0.00395	0.00173
[1,1]	20000	100%	71.43%	0.01255	0.20082
[1,1]	50000	100%	76.19%	0.00394	0.16988
[1,1]	200000	100%	76.19%	0.00045	0.16407

對於非線性問題，Hidden units 較多有助於學習複雜問題，較少（[5,5]或以下）hidden units 學習表現較差，即使增加訓練次數，仍無法達到 100% 準確度（例如[1,1]即使 epochs 增加至 10 倍正確率依然較差）；線性問題都表現很好。

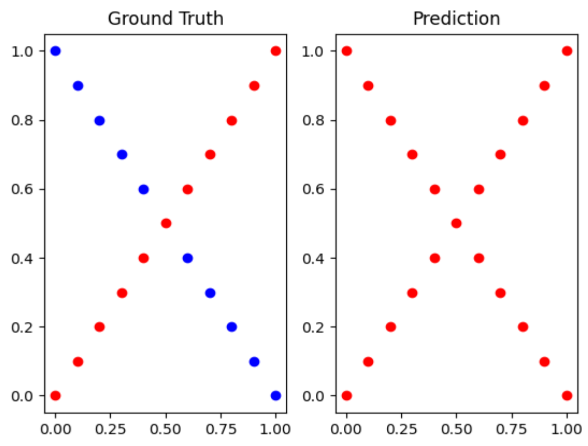
C. Try without activation functions

Parameters:
epochs = 20000、learning rate = 0.1
hidden layers = 2、hidden units = [10,10]
optimizer = SGD、layer = Affine、activation layer = Sigmoid

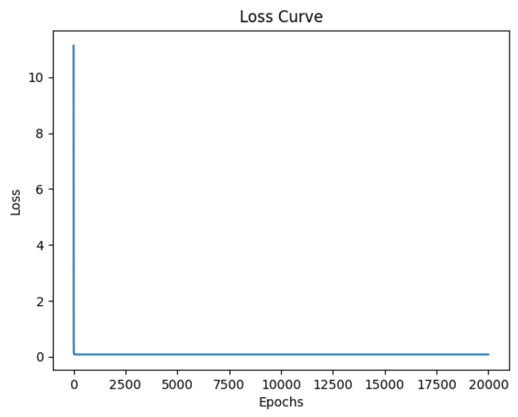
Linear



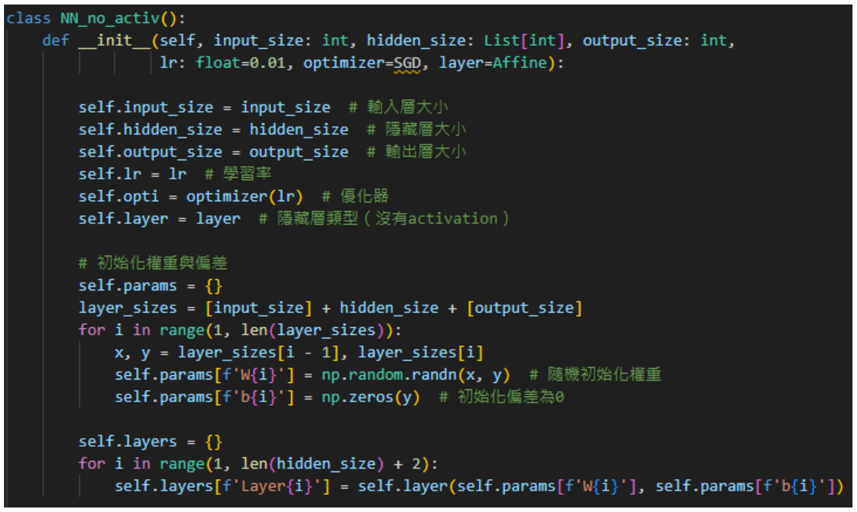
XOR



Loss



部分 without activation function：NN_no_activ 程式碼



```
class NN_no_activ():
    def __init__(self, input_size: int, hidden_size: List[int], output_size: int,
                 lr: float=0.01, optimizer=SGD, layer=Affine):

        self.input_size = input_size  # 輸入層大小
        self.hidden_size = hidden_size  # 隱藏層大小
        self.output_size = output_size  # 輸出層大小
        self.lr = lr  # 學習率
        self.opti = optimizer(lr)  # 優化器
        self.layer = layer  # 隱藏層類型（沒有activation）

        # 初始化權重與偏差
        self.params = {}
        layer_sizes = [input_size] + hidden_size + [output_size]
        for i in range(1, len(layer_sizes)):
            x, y = layer_sizes[i - 1], layer_sizes[i]
            self.params[f'W{i}'] = np.random.randn(x, y)  # 隨機初始化權重
            self.params[f'b{i}'] = np.zeros(y)  # 初始化偏差為0

        self.layers = {}
        for i in range(1, len(hidden_size) + 2):
            self.layers[f'Layer{i}'] = self.layer(self.params[f'W{i}'], self.params[f'b{i}'])
```

NN_no_activ()初始化中只包含了線性層（Affine），沒有 activation layer，讓模型變得簡單但無法處理複雜的非線性關係，因此可以看到 XOR 測試結果完全沒有分類成功。

5. Questions

A. What is the purpose of activation functions?

Activation functions 讓模型能夠學習複雜的 non-linear patterns，且在 backpropagation 階段，activation functions 會影響 gradients 在 network 中的傳遞，並讓訓練能夠逼近複雜的函數。

B. What might happen if the learning rate is too large or too small?

當 learning rate 太大時，更新的步伐可能會太大，導致每次權重更新偏離正確方向，有可能造成 loss function 在訓練過程中來回震盪無法收斂；若 learning rate 太小時，每次更新權重的步伐會非常小，導致訓練過程非常緩慢，可能會在 loss function 的 local minimum 卡住，無法找到 global minimum。

C. What is the purpose of weights and biases in a neural network?

weights 決定輸入數據中每個特徵對神經元輸出的影響程度，每一個輸入特徵都會和對應的 weights 相乘，然後進行加總，這樣可以調整每個特徵對最終輸出的貢獻大小；bias 是一個額外參數添加在結果中，用來調整每一層的輸出，這樣可以讓模型在所有輸入為零時，有一個不為零的輸出，提升模型的表達能力。

可以看做 weight 是縮放，bias 是平移。

Score:

If there are any format errors, you will be punished (-5%)

Reference:

1. https://github.com/hank891008/Deep-Learning/tree/main
2. https://github.com/KJLdefeated/NYCU_DLP_2024/tree/main
3. https://github.com/steven112163/Deep-Learning-and-Practice
4. https://github.com/romanycc/NYCU_DLP_2023
5. https://github.com/cxyfer/NYCU_DLP
6. https://github.com/gyes00205/NYCU_DLP_2022
7. https://github.com/ray0727/Deep-Learning-and-Practice