Lab1 : Back-Propagation

311581005 智能碩一 吳佳豪

1 INTRODUCTION

Backpropagation 主要概念為將預測值與實際值計算出 gradients 後，利用 chain rule 方法由後往前傳播，來更新整個神經網路每層的權重以達到損失函數最小化。

此作業主要使用 NumPy 與基本 Library 來進行實作 Backpropagation。首先會先建構一個基本的神經網路，使用 linear 和 XOR 資料來訓練，使預測達到最佳準確度，並從中探討有關 hyperparameters 的設定差別。

2 EXPERIMENT SETUPS

2.1 SIGMOID

Sigmoid 是將任何數值映射到[0,1]之間的函數，常用於人工神經網路中的激活函數(Activation Function)。訓練模型進行二元分類的情境使用此函數非常適合，因輸出數值介於[0,1]之間，所以可以設定大於或小於閥值 0.5 進行二元分類預測。但缺點為此函數的一階微分會使值最大只有 0.25，如果層數過於多時會出

現「梯度消失」的問題，導致權重無法有效更新而停止訓練，所以通常 hidden layer 會使用 ReLU 函數來解決梯度問題。

Sigmoid Function:

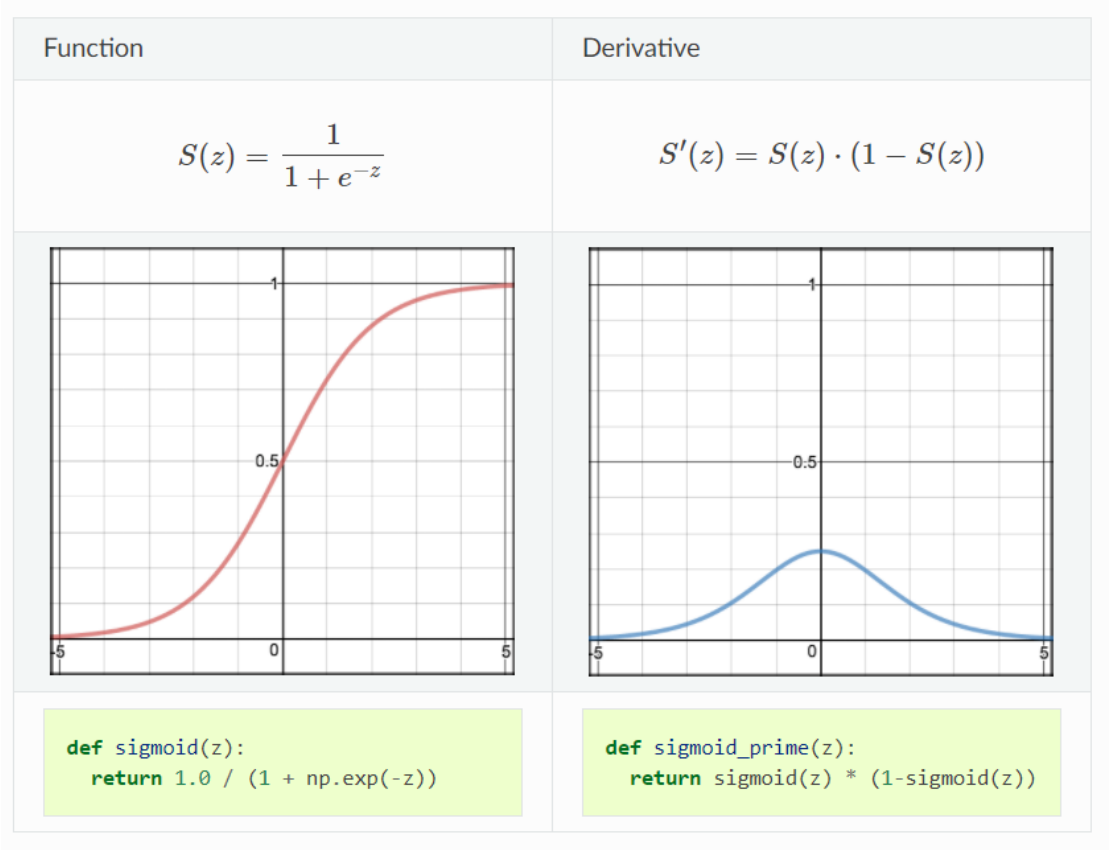

Figure 1. Sigmoid Function and Derivative Sigmoid Function

2.2 NEURAL NETWORK

以下為此次神經網路架構與參數設定。

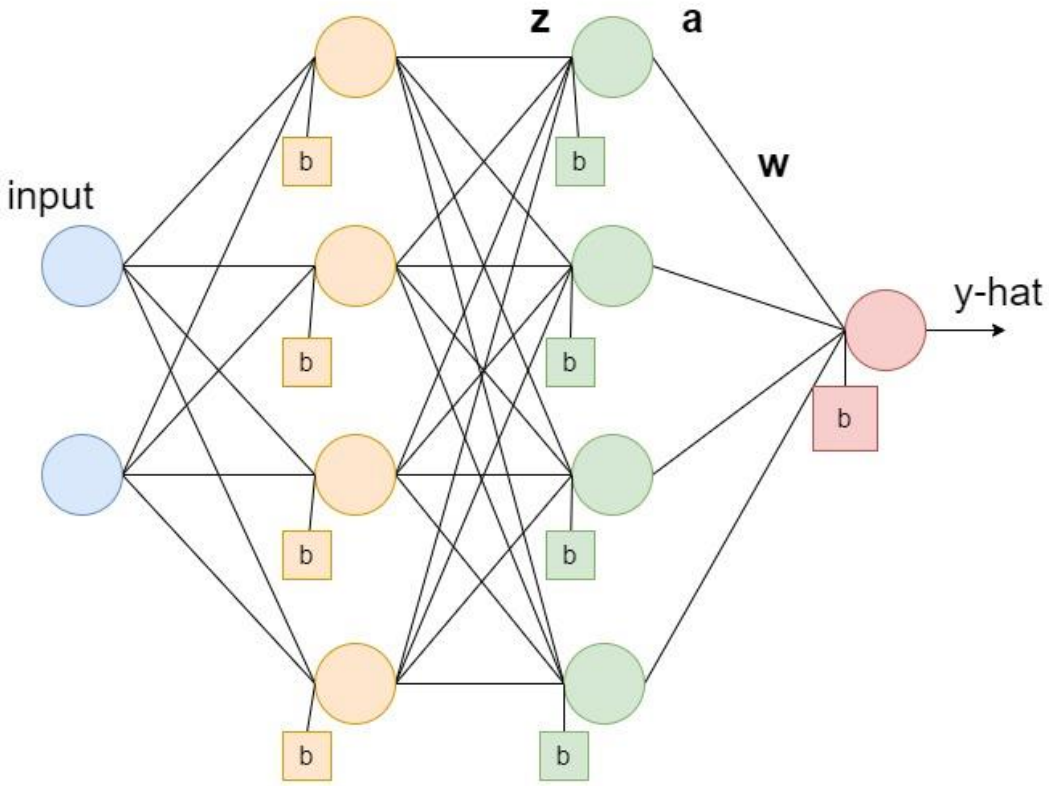

Figure 2. Neural Network Architecture

Hyperparameters	Linear	XOR
Initial weight	Normal Distribution	Normal Distribution
Initial bias	Normal Distribution	Normal Distribution
Learning rate	0.1	0.05
Loss	MSE	MSE
Optimizer	SGD without Momentum	SGD without Momentum
Epochs	10000	10000
Batch size	1	1
Hidden Layer Neural	4	4

Table 1. Parameter setting

2.3 BACKPROPAGATION

主要為先計算出每層網路的 gradient 後再去更新權重。

如果有 L 層的 Layer， 則每層參數 wights、bias、輸出、經過 activation Function 的輸出，分別用 w^L 、b^L 、z^L 、a^L 表示，訓練資料輸入為 x。

2.3.1 Backward

for loop 從最後面 layer 開始計算每層的 gradient:

1. 先計算 dz^L：

$if\ L == last\ layer$

$$dz^L = MSE'(y_{hat}, y) * activation\ function'(z^L)$$

$else$

$$dz^L = (w^{L+1})^T.\ dz^{L+1} * activation\ function'(z^L)$$

2. 之後回推 dw^L：

$if\ L\ != first\ layer$

$$dw^L = dz^L.(a^{L-1})^T * activation\ function'(z^L)$$

$else$

$$dw^L = dz^L.x^T * activation\ function'(z^L)$$

3. 最後計算 db^L：

$$db^L = dz^L$$

根據上方算法即更新每層的 gradient，以利後面 weight update

```
def backward(self, x, y_hat, Y):
    for i in range(self.layers_size - 1, -1, -1):
        derivative_activate_func = self.choose_derivative_func(
            self.layers[i]["activate_function"])
        if i == self.layers_size - 1:
            self.gards["dz"][i] = self.derivative_MSE(
                y_hat, Y) * derivative_activate_func(self.params["z"][i])
        else:
            self.gards["dz"][i] = np.dot(
                self.params["weight"][i + 1].T, self.gards["dz"][i + 1]) * derivative_activate_func(self.params["z"][i])

        self.gards["dw"][i] = np.dot(self.gards["dz"][i], np.transpose(
            self.params["a"][i - 1] if i != 0 else x)) / Y.shape[1]
        self.gards["db"][i] = np.sum(
            self.gards["dz"][i]) / Y.shape[1]
```

Figure 3. Backward Implement

2.3.2 Weight update

將 gradient 乘上 learning rate 後更新並加到權重上，如果有 momentum 則再加上一個比例乘上上次更新 gradient 的值。

```
class SGD():
    def __init__(self, lr=0.01, momentum=0.0):
        self.lr = lr
        self.momentum = momentum
        self.v = None

    def update(self,params, gards):
        if self.v is None:
            self.v = [np.zeros_like(gard) for gard in gards]
        for layer in range(len(gards)):
            self.v[layer] = -self.lr * gards[layer] + self.momentum * self.v[layer]
            params[layer] += self.v[layer]
        return params
```

Figure 2. Implement of updating weights with SGD

3 RESULTS OF YOURS TESTING

3.1 SCREENSHOT AND COMPARISON FIGURE

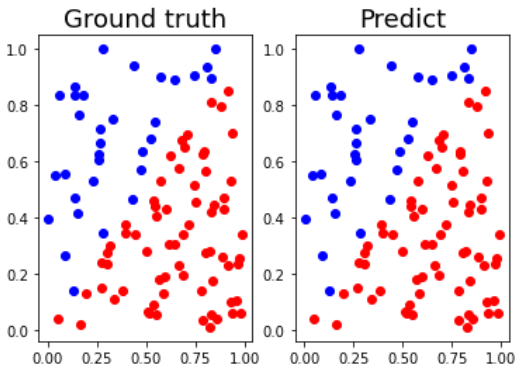

Figure 4. Linear Predict

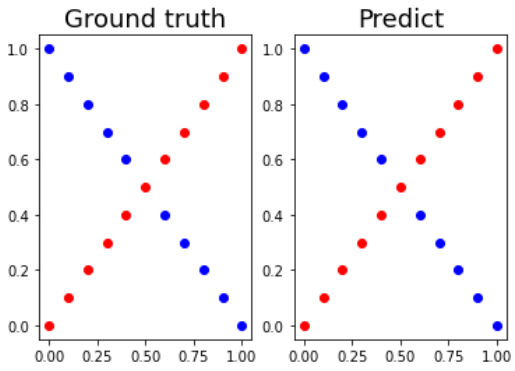

Figure 5.XOR Predict

3.2 SHOW THE ACCURACY OF PREDICTION

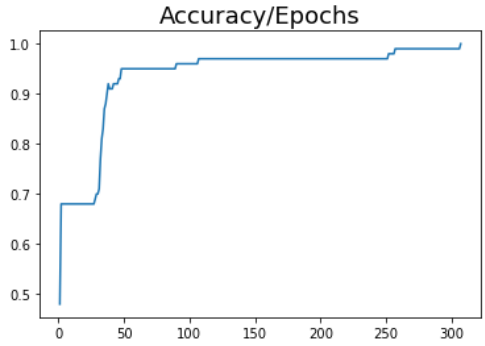

Figure 6. Linear Accuracy

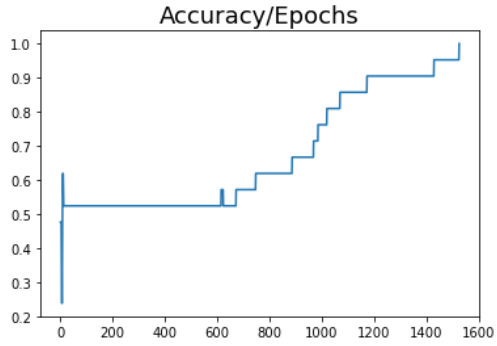

Figure 7.XOR Accuracy

3.3 LEARNING CURVE

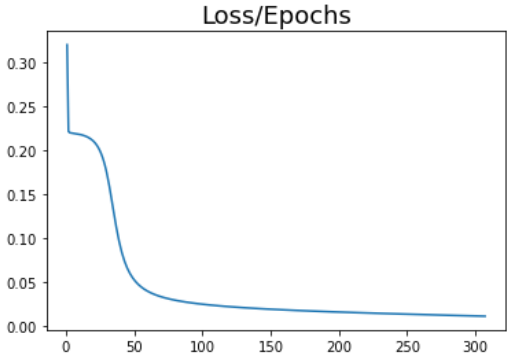

Figure 8. Linear Loss

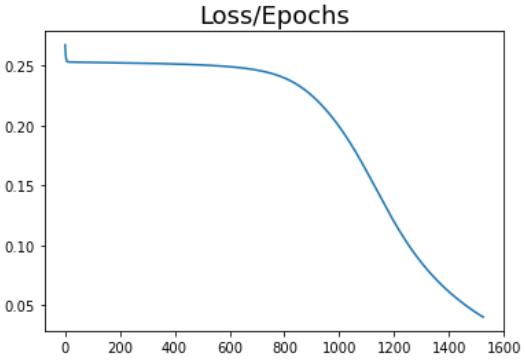

Figure 9.XOR Loss

3.4 PARAMETER INITIALIZATION

在實驗中發現一開始權重的隨機參數初始化方法對後面的訓練收斂速度有很大的影響。一開始是使用 uniform 的方法，使參數都均勻分布在[0,1]之間，但發現經過 Sigmoid 後容易使值超過 0.5 的閥值，因此訓練不易。所以後來變更為 Normal Distribution 方式產生值後大大的改善，收斂較為容易許多。

實驗設定為同樣 lr = 0.05、dataset 為 XOR dataset。可以發現使用 Normal Distribution 產生參數可以快速收斂，而 uniform 即使跑到 10000 個 epochs 還是沒辦法 100% 的準確度。

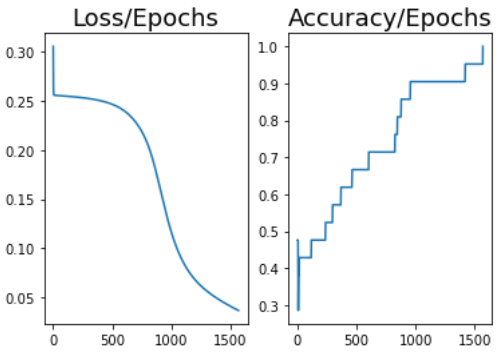

Figure 10.Use Normal Distribution

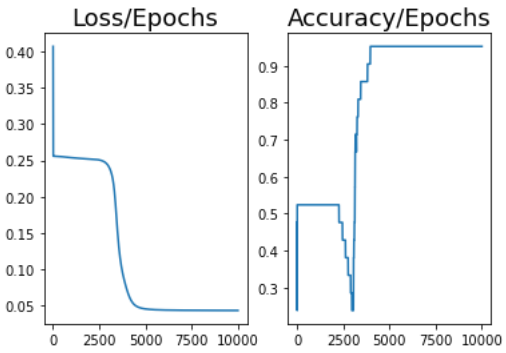

Figure 11.Use Uniform Distribution

4 DISCUSSION

4.1 TRY DIFFERENT LEARNING RATES

實驗設定為同樣 epochs=500，發現 learning rate 越大收斂速度越快，但可以發現過大的 learning rate 會使學習曲線有抖動不平滑的現象發生。

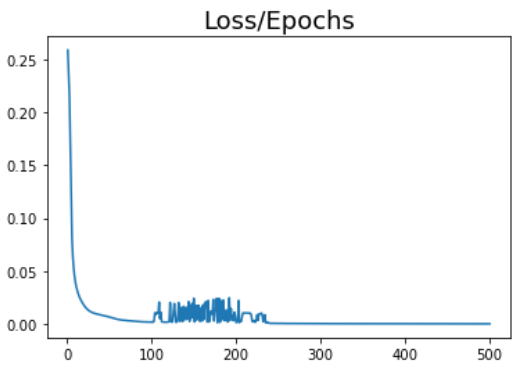

Figure 12. lr= 0.5

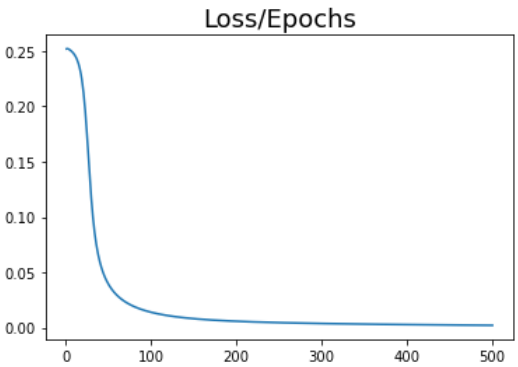

Figure 13. lr = 0.1

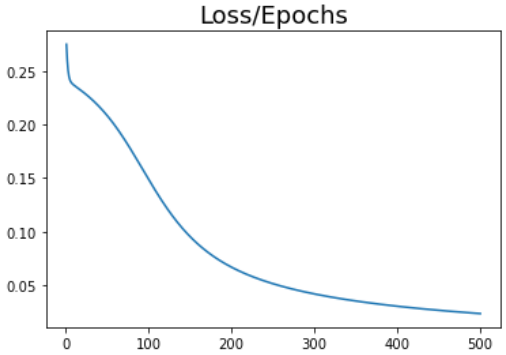

Figure 14. lr = 0.01

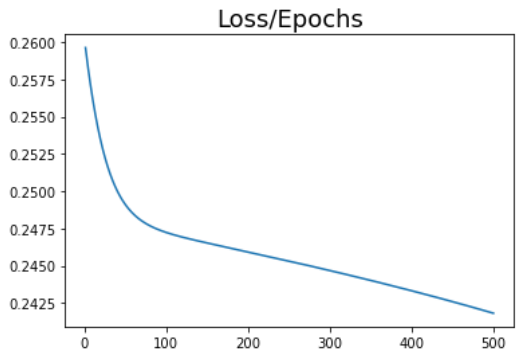

Figure 15. lr = 0.001

4.2 TRY DIFFERENT NUMBERS OF HIDDEN UNITS

實驗設定為同樣 epochs=1000、lr=0.1，可以發現越多的神經元在相同的 dataset 時，可以更快的收斂並準確預測。只是增加過多的 units 可能會使神經網路過度擬合與消耗更多的資源時間，所以不能一直增加 units。

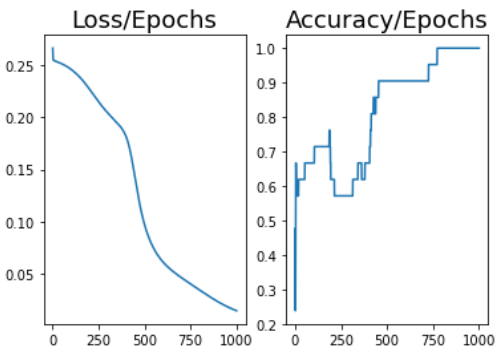

Figure 16. Units = 4

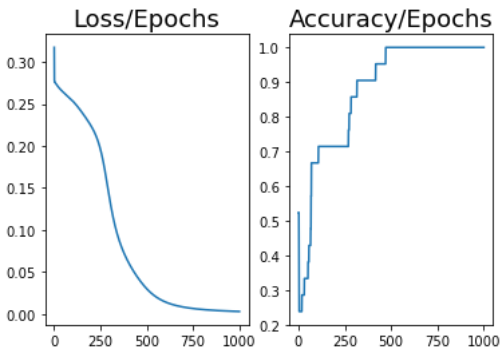

Figure 17. Units = 10

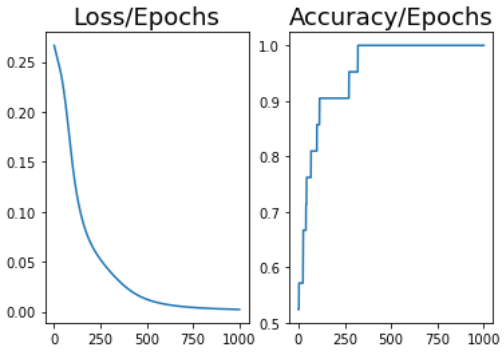

Figure 18. Units = 20

4.3 TRY WITHOUT ACTIVATION FUNCTIONS

因為沒有 activation function 控制數值的範圍，所以當 learning rate 設定相同時會發生梯度爆炸而影響訓練，因此 learning rate 數值都需要較低以避免發生上述狀況。

4.3.1 Linear dataset

實驗設定為在測試相同的 linear dataset 中。沒有 activation function 的設定為 lr=0.01 則 Accuracy=98%，而有 activation function 的設定為 lr=0.1 則達到 Accuracy=100%。

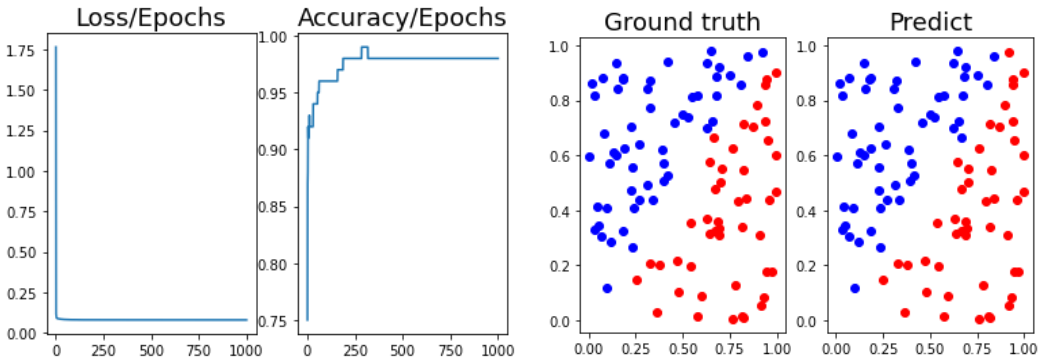

Figure 19. Without activation function, lr=0.01

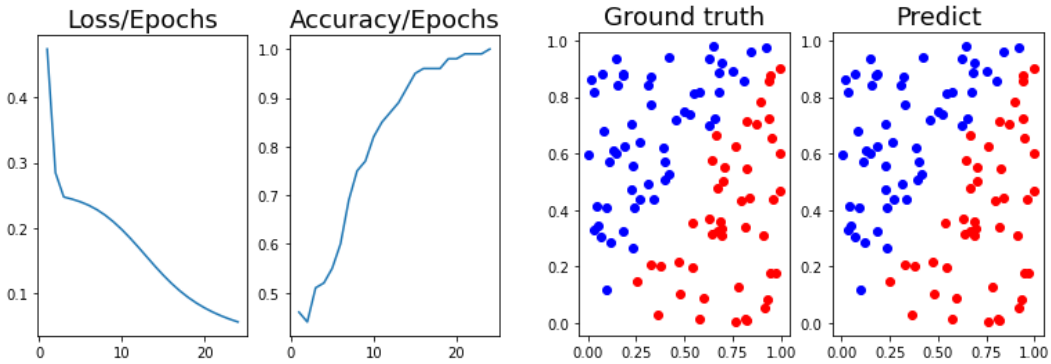

Figure 20. Activation function with sigmoid, lr=0.1

4.3.2 XOR dataset

實驗設定為在測試相同的 XOR dataset 中。沒有 activation function 的設定為 lr=0.005 則 Accuracy=52.38%，而有 activation function 的設定為 lr=0.05 則達到 Accuracy=100%。可以發現在此 dataset 如果沒有 activation function 會卡住而無法繼續收斂。

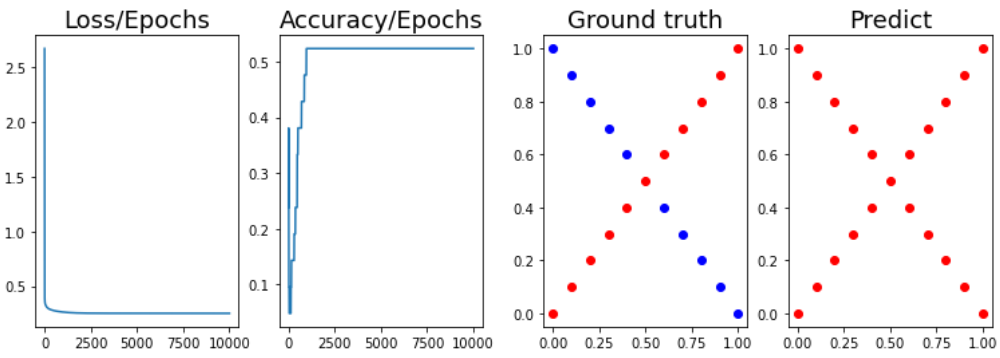

Figure 21. without activation function, lr=0.005

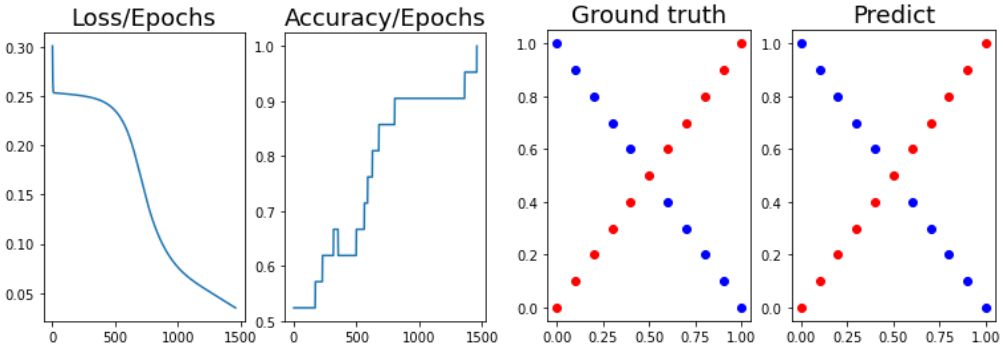

Figure 22. activation function with sigmoid , lr=0.05

4.4 TRY DIFFERENT NUMBERS OF HIDDEN LAYERS

實驗設定為同樣 epochs=1000、lr=0.05、資料集為 XOR dataset、hidden layer 的 activation function 都變更為 relu 而最後輸出層為 sigmoid、每層 units 都為 4。由下面圖發現，在相同的 learning rate 下，層數越多學習曲線會越來越抖動，到最後無法收斂，除非將 learning rate 降低才會繼續收斂。

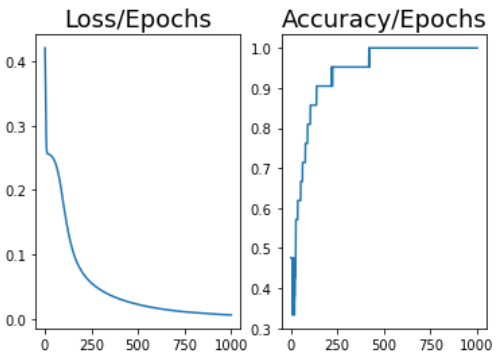

Figure 23. hidden layer = 1, lr = 0.05

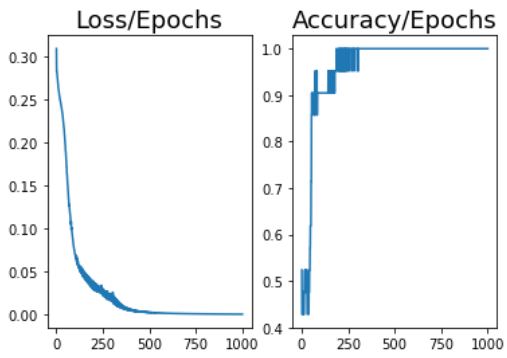

Figure 24. hidden layer = 2, lr = 0.05

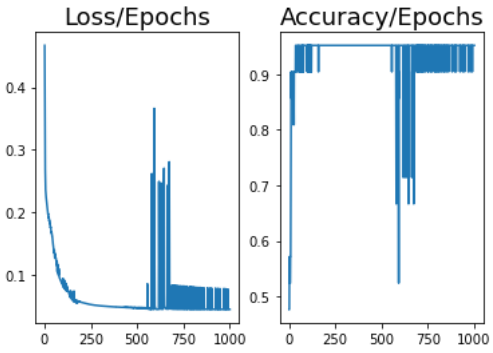

Figure 25. hidden layer = 3, lr = 0.05

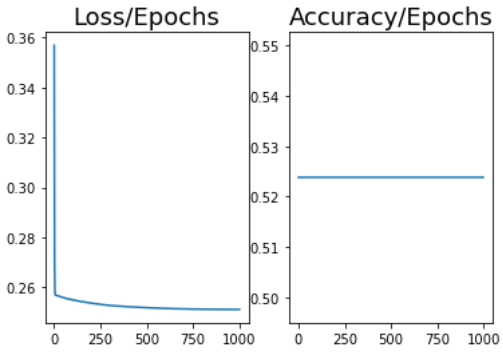

Figure 26. hidden layer = 4, lr = 0.05

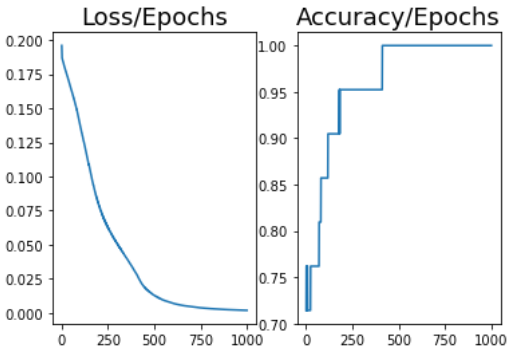

Figure 27. hidden layer = 4, lr = 0.005

5 EXTRA

5.1 IMPLEMENT DIFFERENT OPTIMIZERS

Optimizer 實作 SGD + Momentum。Momentum 可以在前期增加梯度下降的速度，當中後期時梯度下降變為趨緩時因加上一定比例加上前期動量，可使卡在 local minimum 的情況降低，達到最佳化。

```python
class SGD():
    def __init__(self, lr=0.01, momentum=0.0):
        self.lr = lr
        self.momentum = momentum
        self.v = None

    def update(self,params, gards):
        if self.v is None:
            self.v = [np.zeros_like(gard) for gard in gards]
        for layer in range(len(gards)):
            self.v[layer] = -self.lr * gards[layer] + self.momentum * self.v[layer]
            params[layer] += self.v[layer]
        return params
```

Figure 28. SGD + Momentum Implement

以下實驗設定為同樣 epochs=1000、lr=0.1、dataset 為 linear dataset，momentum=0.9 與 momentum=0.0 的差別。可以發現有加上 Momentum 的神經網路快速收斂。

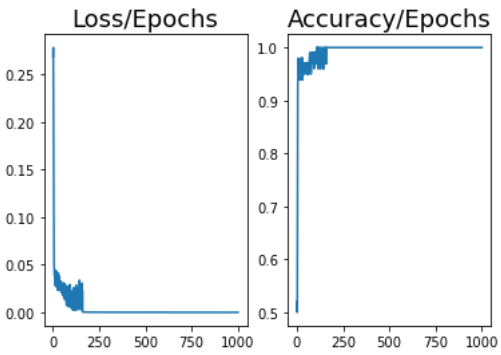

Figure29 . momentum=0.9

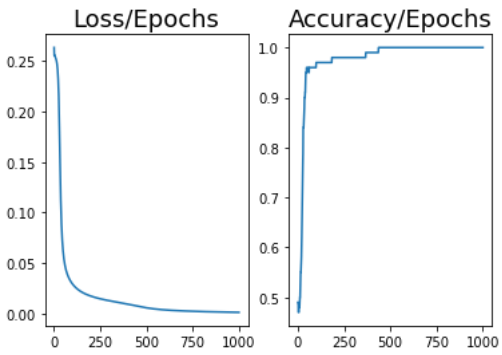

Figure 30. momentum=0.0

5.2 IMPLEMENT DIFFERENT ACTIVATION FUNCTIONS

實作 ReLU，ReLU 是將值小於 0 時為 0，大於 0 時則輸出相同值。此方法可以避免 sigmoid 會發生的梯度消失問題且計算效率較高。但缺點是反向傳播時當輸入為負數時梯度將完全為零，則該神經元的梯度永遠都會是 0，發生所謂的 Dead ReLU 問題。

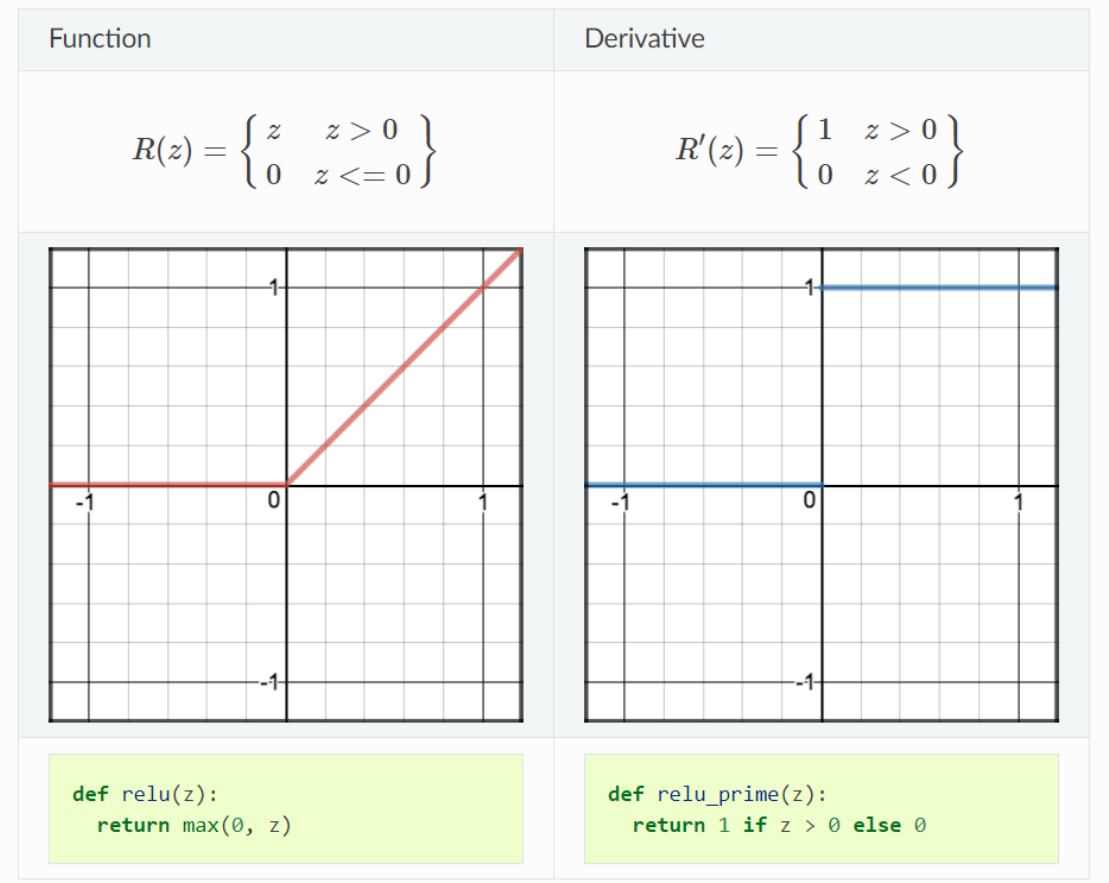

Figure 31. ReLU Function

以下實驗設定為 epochs=1000、lr=0.05、momentum=0.0、dataset 為 linear dataset，輸出層 activation function 為 sigmoid，其餘層階用 ReLU。

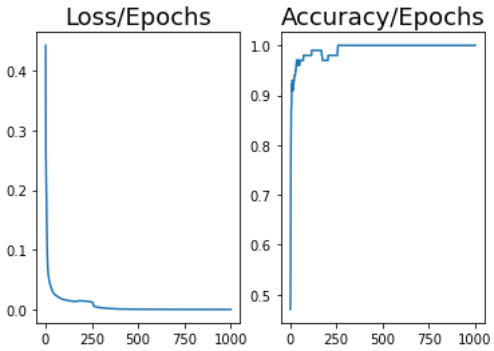

Figure 32. Use ReLU Function

```python
def relu(x):
    return np.maximum(x, 0)


def derviative_relu(x):
    x[x <= 0] = 0
    x[x > 0] = 1
    return x
```

Figure 33. ReLU Function Implement