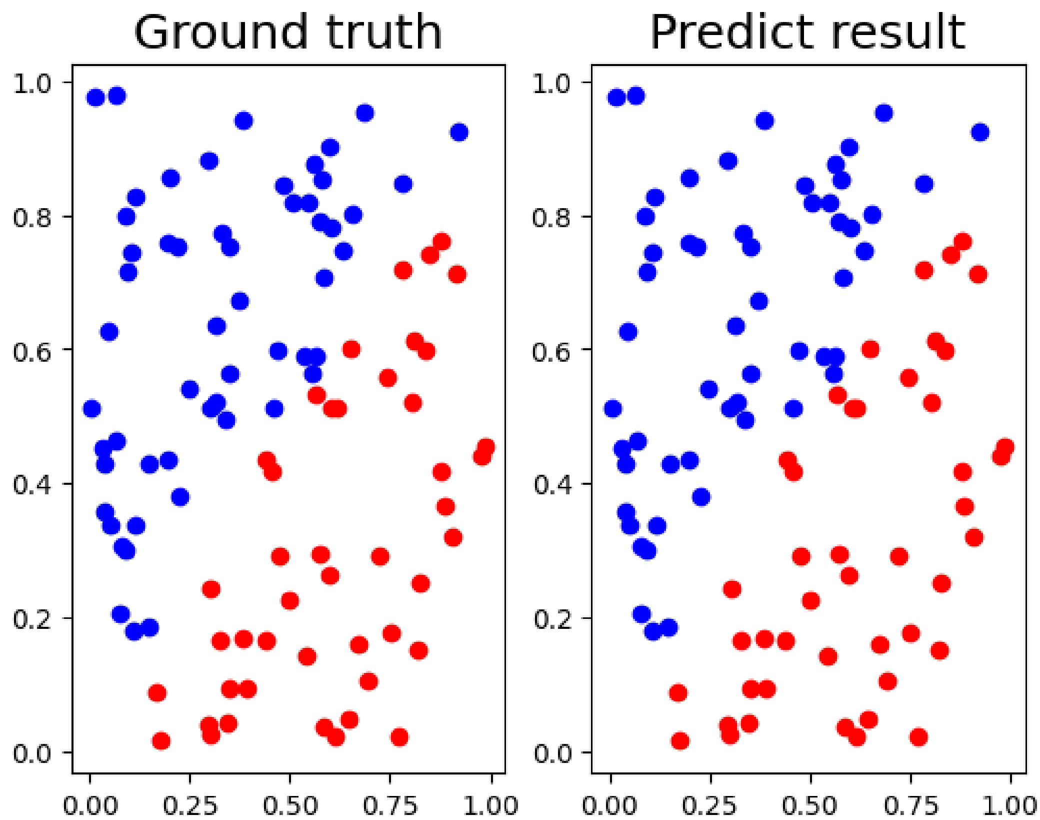
Lab 1: Backpropagation

311605010 周孫甫

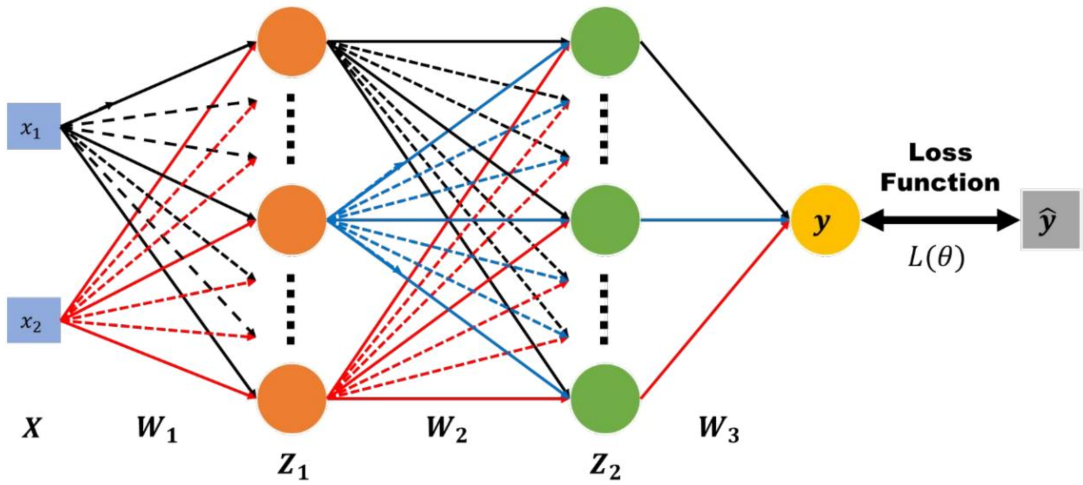
1. Introduction

本次 Lab1 使用指定的 fully-connected neural network 結構對資料進行分類預測， neural network 的 forward pass、backpropagation 皆需實作。

資料格式為一組二維資料，包含輸入 x_1, x_2、輸出 y，如圖所示。其中輸出 y 以顏色表示，紅色表示數值為 0 的分類，藍色表示數值為 1 的分類。



Neural network 結構為 Fully-connected，包含 2 層 hidden layers，以及最後的輸出層，每層 neuron 不定，在我的程式中，兩層 hidden layer 具有相同的 number of unit。



計算流程：

(1) 產生 training data

(2) 決定 hyperparameters，初始化 parameters

(3) Forward pass 得出 prediction

(4) Compute loss

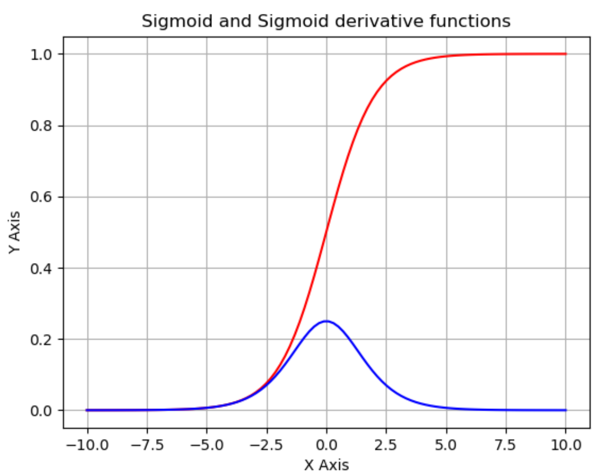
(5) Backpropagation and update parameters

(6) 重複(3)~(5)直至預設 epochs

2. Experiment setups

(A) Sigmoid function

Sigmoid function $S(x) = \frac{1}{1+e^{-x}}$，$S'(x) = \left(1 - f(x)\right) \times f(x)$



實作在 actfcn.py:15

```python
class Sigmoid(ActFcn):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        return 1.0 / (1.0 + np.exp(-x))

    def backward(self, x):
        return np.multiply(x, 1.0 - x)
```

(B) Neural network

(1) Layers

每層 Layer 首先初始化時會亂數初始化所有參數並記錄 activation function、optimizer，之後計算 forward pass，backpropagation and update parameters 都在這邊實作。

實作在 nn.py:7

```python
class Layer:
    def __init__(
        self,
        input_dim=2,
        output_dim=2,
        have_bias=True,
        act_fcn=actfcn.ActFcn,
        optimizer=optimizer.Optimizer,
        optimizer_parameter=optimizer.Optimizer
    ):
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.have_bias = have_bias
        self.act_fcn = act_fcn()
        self.optimizer = optimizer(*optimizer_parameter.get_param)

        self.w = np.random.randn(self.input_dim, self.output_dim)
        self.b = np.random.randn(1, self.output_dim)

        # self.w *= 0.1
        # self.b *= 0.1

    def forward(self, input):
        self.intputs = input
        if self.have_bias:
            self.outputs = self.act_fcn.forward(input.dot(self.w) + self.b)
        else:
            self.outputs = self.act_fcn.forward(input.dot(self.w))
        return self.outputs

    def backward(self, dy):
        dy_new = dy * self.act_fcn.backward(self.outputs)
        dw = self.intputs.T.dot(dy_new)
        db = np.sum(dy_new, axis=0)
        self.w, self.b = self.optimizer.optimize(self.w, dw, self.b, db)
        return dy_new.dot(self.w.T)
```

(2) Neural Network

Neural Network 為制定各層 hyperparameter，對各層依序計算 forward pass，反序計算 backpropagation

實作在 nn.py:45

```python
45 class NN:
46     def __init__(
47         self,
48         dims,
49         have_bias=True,
50         act_fcn=actfcn.ActFcn,
51         optimizer=optimizer.Optimizer,
52         optimizer_parameter=optimizer.Optimizer
53     ):
54         self.layers = (
55             Layer(2, dims[0], have_bias, act_fcn, optimizer, optimizer_parameter),
56             Layer(dims[0], dims[1], have_bias, act_fcn, optimizer, optimizer_parameter),
57             Layer(dims[1], 1, have_bias, act_fcn, optimizer, optimizer_parameter),
58         )
59 
60     def forward(self, x):
61         for layer in self.layers:
62             x = layer.forward(x)
63         return x
64 
65     def backward(self, dy):
66         for layer in reversed(self.layers):
67             dy = layer.backward(dy)
```

(C) Backpropagation

每層參數均被亂數初始化，之後根據 loss 按照 backpropagation 流程更新參數，實作包含

在 Layer 中

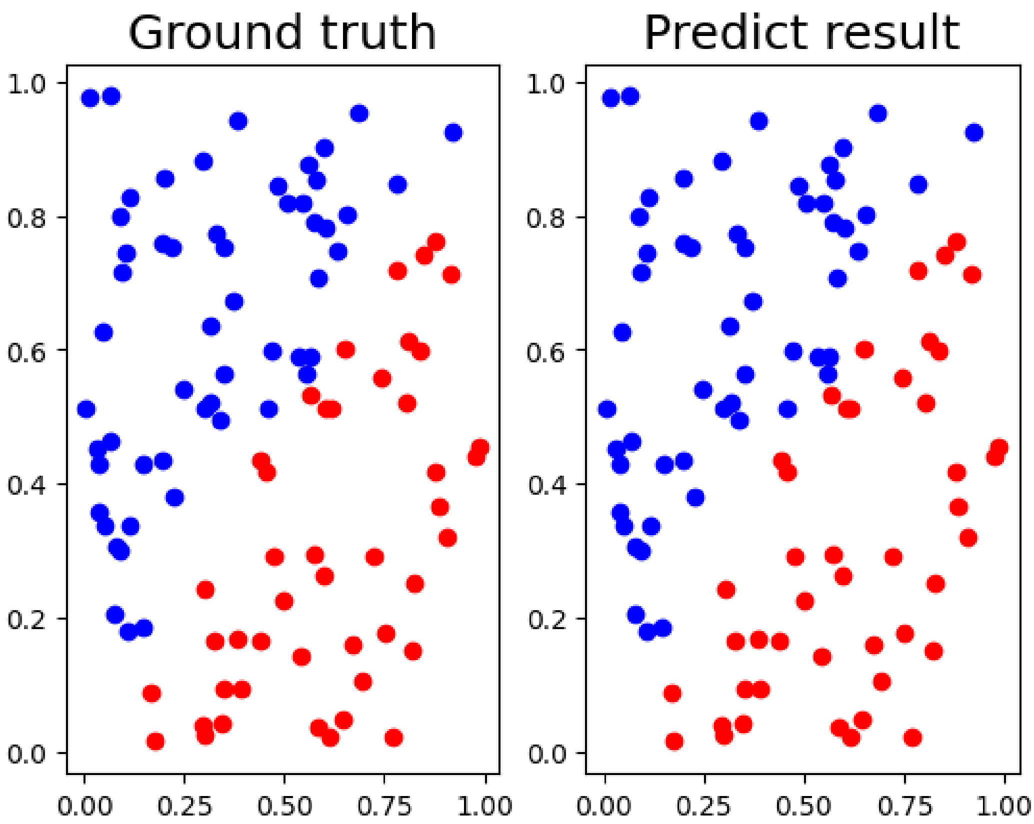
```python
    def backward(self, dy):
        dy_new = dy * self.act_fcn.backward(self.outputs)
        dw = self.intputs.T.dot(dy_new)
        db = np.sum(dy_new, axis=0)
        self.w, self.b = self.optimizer.optimize(self.w, dw, self.b, db)
        return dy_new.dot(self.w.T)
```

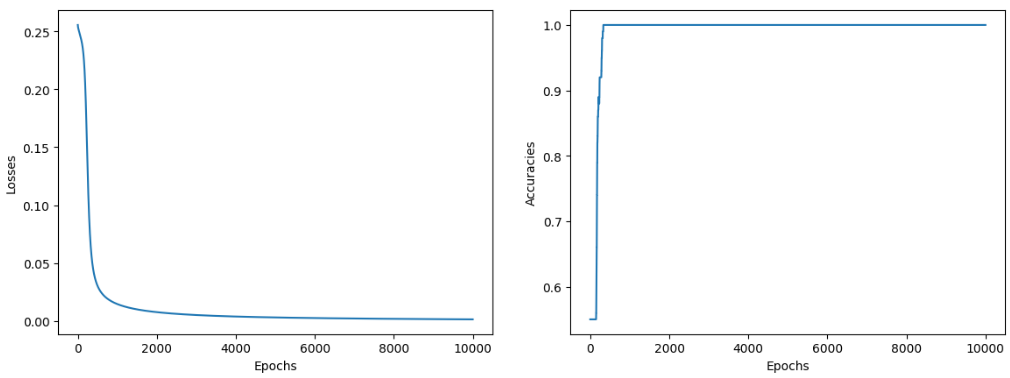
3. Results of your testing

Linear dataset

```
epoch: 0, loss: 0.25550160163820484, accuracy: 0.55
epoch: 1000, loss: 0.01470348415634327, accuracy: 1.0
epoch: 2000, loss: 0.007739255502554464, accuracy: 1.0
epoch: 3000, loss: 0.005247297474932305, accuracy: 1.0
epoch: 4000, loss: 0.003962679559763278, accuracy: 1.0
epoch: 5000, loss: 0.0031790441330778723, accuracy: 1.0
epoch: 6000, loss: 0.0026469361860291325, accuracy: 1.0
epoch: 7000, loss: 0.0022581562656887166, accuracy: 1.0
epoch: 8000, loss: 0.0019592046313445855, accuracy: 1.0
epoch: 9000, loss: 0.0017208519286392642, accuracy: 1.0
```

```
[[5.63065938e-06]
 [9.99824210e-01]
 [1.26418293e-05]
 [9.99990153e-01]
 [4.30840136e-04]
 [9.99981574e-01]
 [9.99989705e-01]
 [3.37829506e-05]
 [7.29683097e-06]
 [3.58126511e-05]
 [7.70946899e-06]
 [9.99983813e-01]
 [2.73424027e-01]
 [4.52418694e-04]
 [2.26573999e-02]
 [1.28905791e-05]
 [6.06169999e-06]
 [9.99960901e-01]
 [9.99990815e-01]
 [9.99990267e-01]
 [9.99981646e-01]
 [9.99990628e-01]
 [9.99988127e-01]
 [9.99990298e-01]
 [4.43241976e-05]
 [1.07532249e-05]
 [5.79567049e-06]
 [9.99990115e-01]
 [6.37351670e-06]
 [6.12413435e-06]
 [9.99984462e-01]
 [9.95398367e-01]
 [9.99984196e-01]
 [9.99968103e-01]
 [9.99988897e-01]
 [9.99987608e-01]
 [9.99985081e-01]
 [1.04885455e-05]
 [5.56150907e-04]
 [6.95755776e-06]
 [8.45882540e-04]
 [9.38157256e-06]
 [1.20663891e-05]
 [1.95109335e-03]
 [7.88325658e-01]
 [3.36810985e-05]
 [9.99991159e-01]
 [9.99913534e-01]
 [6.34033608e-06]
 [9.99962569e-01]
 [9.99982127e-01]
 [9.99987830e-01]
 [1.36849272e-02]
 [6.25392748e-06]
 [9.99968146e-01]
 [9.99967254e-01]
 [6.46127495e-06]
 [9.99981379e-01]
 [9.90185889e-01]
 [6.43312518e-06]
 [9.99989941e-01]
 [9.99990557e-01]
 [9.99849799e-01]
 [8.48474410e-01]
 [6.03471659e-06]
 [9.99982907e-01]
 [9.99897002e-01]
 [9.48558587e-01]
 [6.00480789e-02]
 [9.99991206e-01]
 [9.99972916e-01]
 [9.99985043e-01]
 [9.99858774e-01]
 [9.99981024e-01]
 [6.06170171e-06]
 [1.25093096e-05]
 [9.99984776e-01]
 [9.99990804e-01]
 [9.49185685e-01]
 [9.99973927e-01]
 [1.34437890e-05]
 [9.99988435e-01]
 [3.96942691e-05]
 [1.97815339e-05]
 [1.96129691e-05]
 [9.98972760e-01]
 [9.99956084e-01]
 [6.04346763e-06]
 [6.12498856e-06]
 [9.93132925e-01]
 [9.99986959e-01]
 [3.94031675e-05]
 [8.92078533e-04]
 [1.90407627e-02]
 [9.99967332e-01]
 [9.99990485e-01]
 [9.99666037e-01]
 [1.23554551e-05]
 [9.99927687e-01]
 [1.36464709e-05]]
```

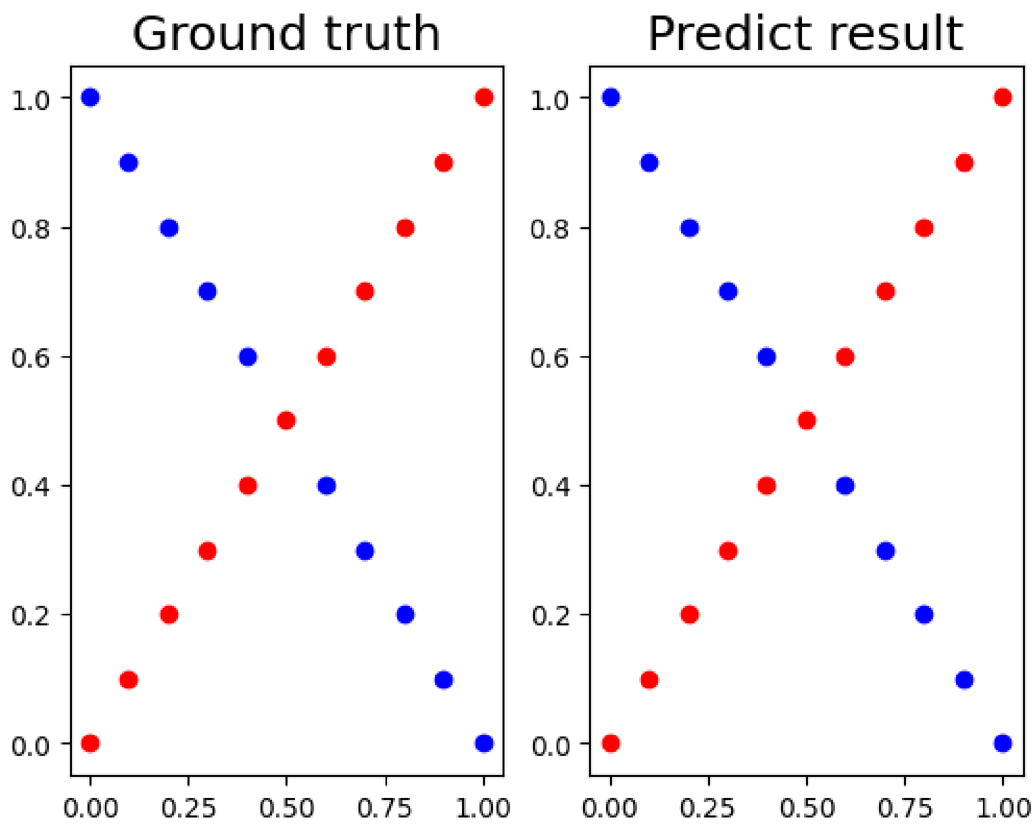


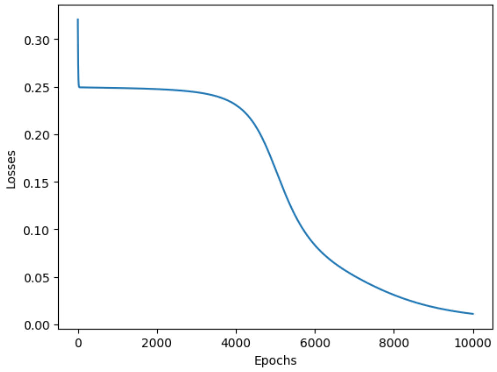


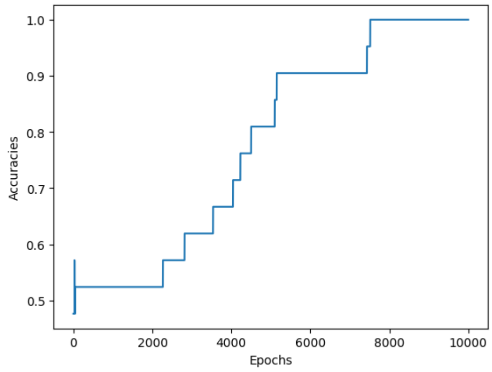
XOR dataset

```
epoch: 0, loss: 0.3207647588892037, accuracy: 0.47619047619047616
epoch: 1000, loss: 0.2487133667562864, accuracy: 0.5238095238095238
epoch: 2000, loss: 0.2475132921125923, accuracy: 0.5238095238095238
epoch: 3000, loss: 0.24426297972530572, accuracy: 0.6190476190476191
epoch: 4000, loss: 0.2308542521833423, accuracy: 0.6666666666666666
epoch: 5000, loss: 0.16375823104729859, accuracy: 0.8095238095238095
epoch: 6000, loss: 0.0834434833592583, accuracy: 0.9047619047619048
epoch: 7000, loss: 0.051018131674399045, accuracy: 0.9047619047619048
epoch: 8000, loss: 0.031055700737388593, accuracy: 1.0
epoch: 9000, loss: 0.018200919185683957, accuracy: 1.0
```

```
[[0.09594101]
 [0.98826482]
 [0.0957082 ]
 [0.98772497]
 [0.09550124]
 [0.98561807]
 [0.09533072]
 [0.97326501]
 [0.09520608]
 [0.74460074]
 [0.09513539]
 [0.09512534]
 [0.74240562]
 [0.09518123]
 [0.97631308]
 [0.09530702]
 [0.98845526]
 [0.09550548]
 [0.99044618]
 [0.09577832]
 [0.99093367]]
```





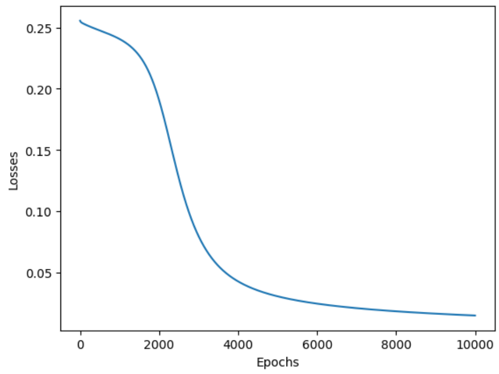
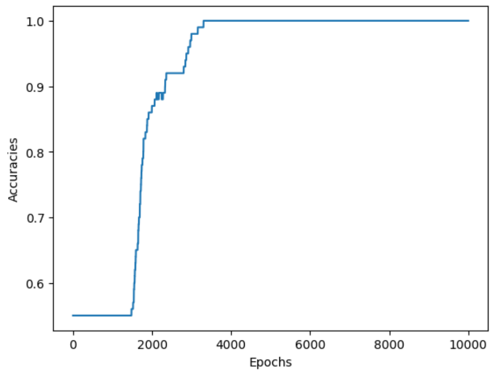
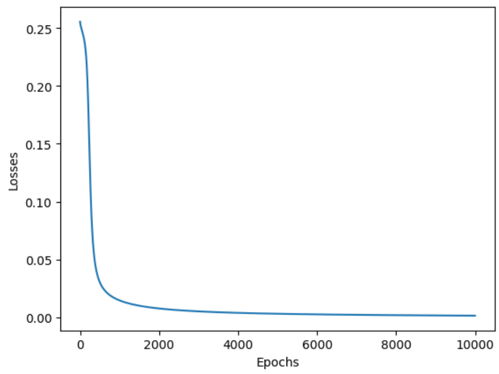
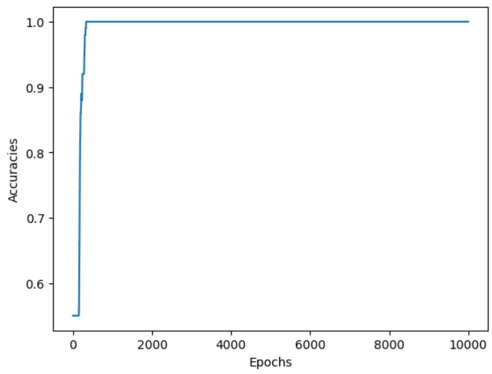


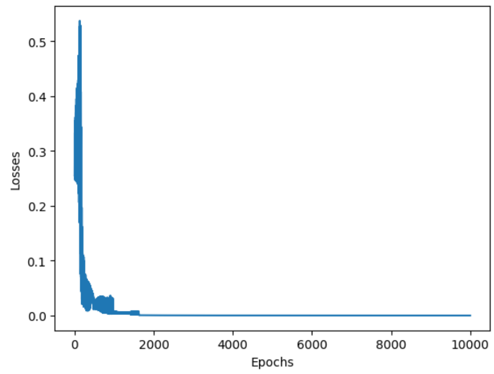
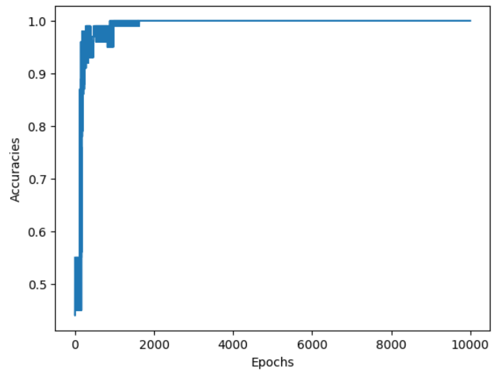
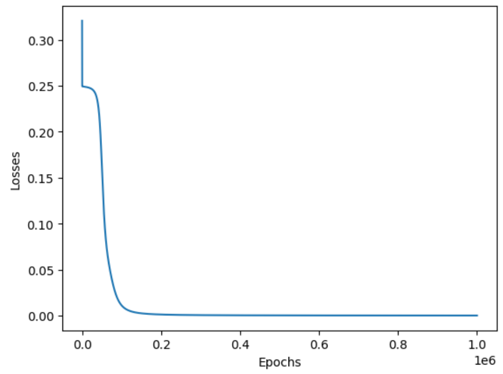
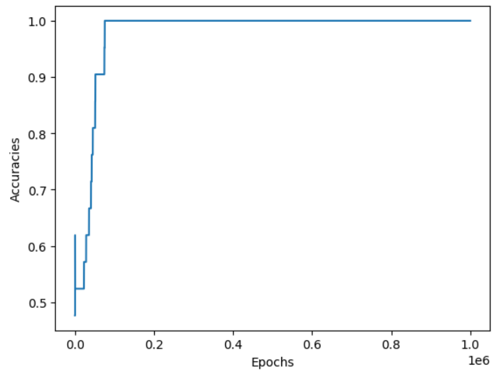
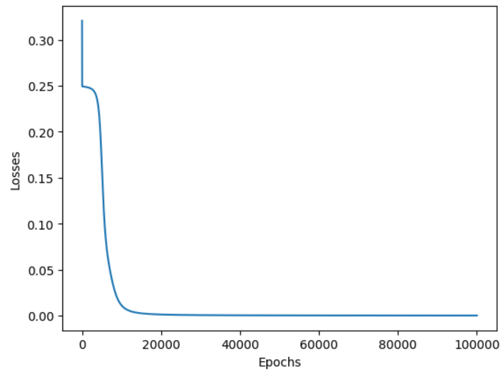
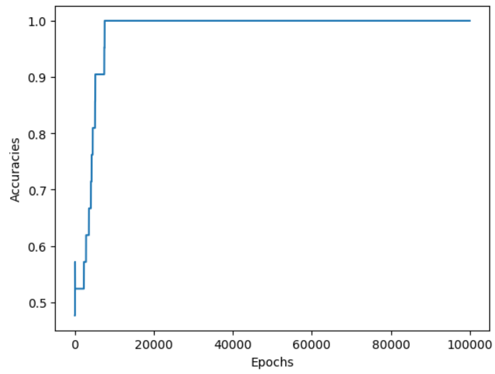
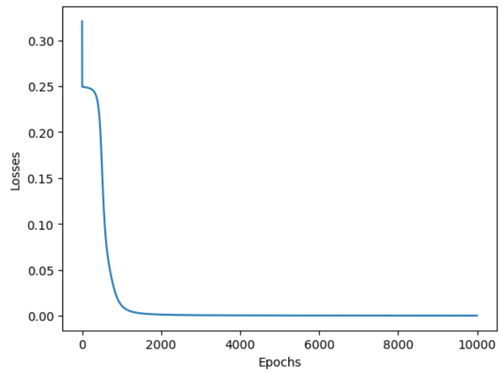
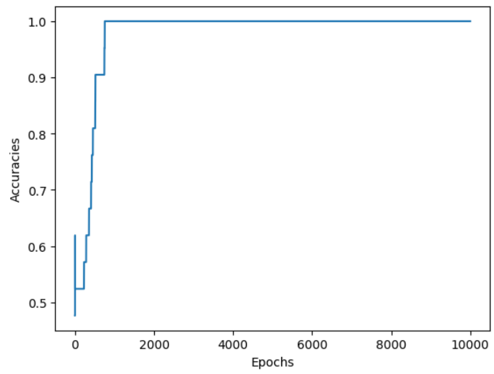
4. Discussion

(A) Learning rate

實驗設計

Dataset	Linear			XOR		
Learning rate	0.001	0.01	0.1	0.001	0.01	0.1
Number of hidden units	(4, 4)			(4, 4)		
Bias	Have bias			Have bias		
Activation Function	Sigmoid			Sigmoid		
Optimizer	SGD			SGD		
Epochs	1.E+04			1.E+06	1.E+05	1.E+04
代號	A	B	C	D	E	F

代號	Loss	Accuracy
A		
B		

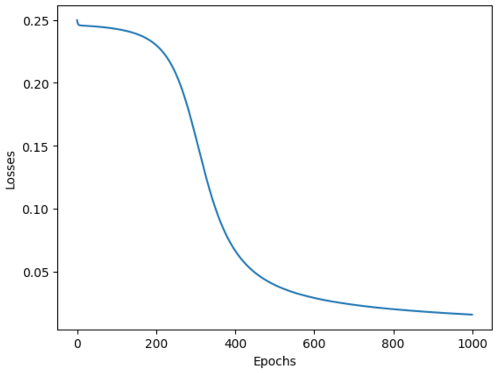
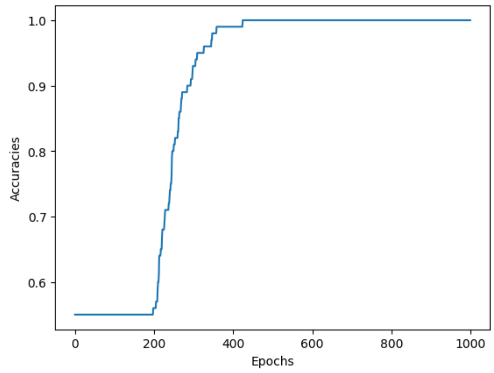
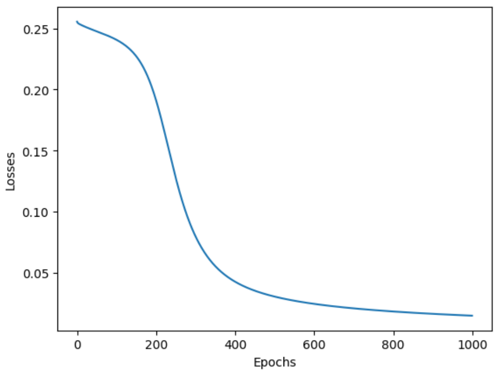
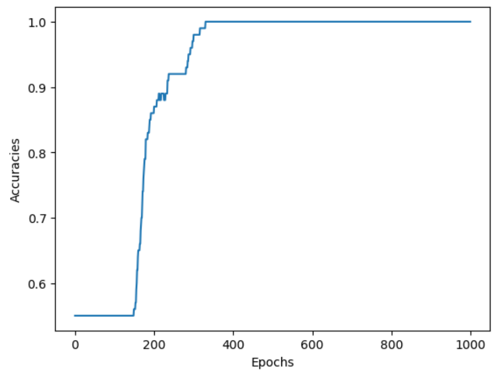
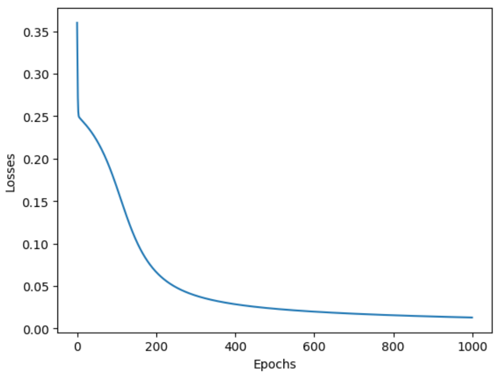
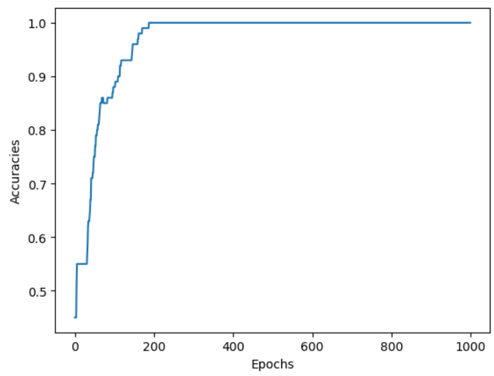
C		
D		
E		
F		

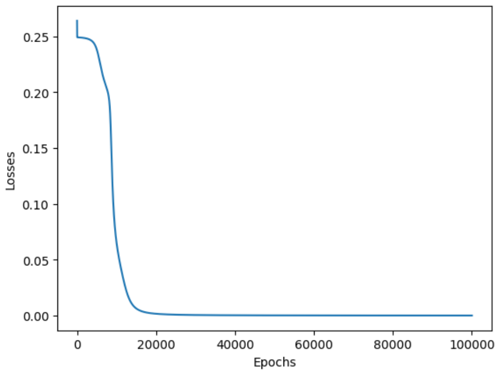
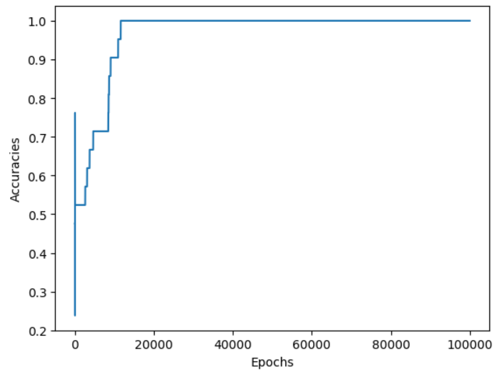
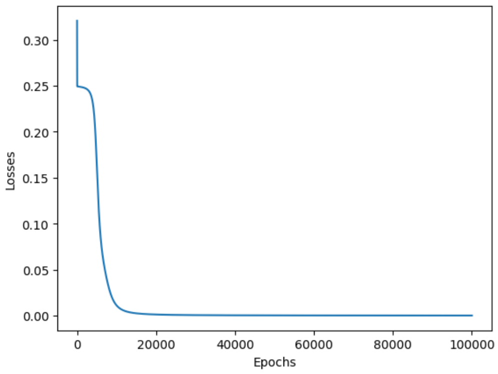
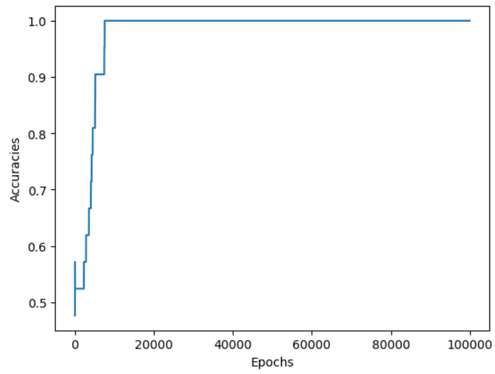
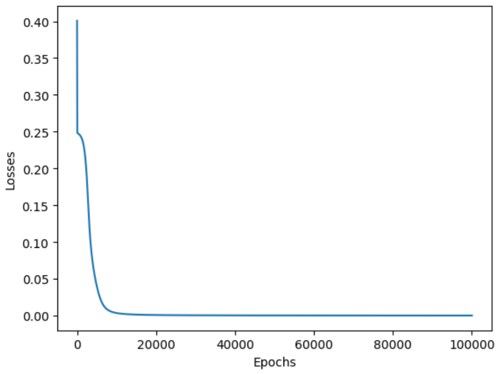
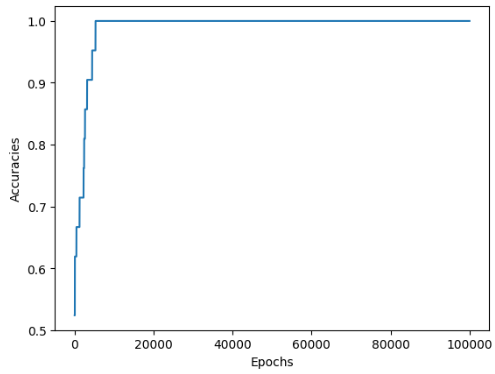
觀察到兩現象，較小的 learning rate 會導致 loss 下降較慢，如 A、D 圖，反之則較快如 C、F 圖。較大的 learning rate 亦可能導致 loss 訓練時不穩定發生震盪。

(B) Number of hidden units

實驗設計

Dataset	Linear			XOR		
Learning rate	0.01			0.01		
Number of hidden units	(2, 2)	(4, 4)	(8, 8)	(2, 2)	(4, 4)	(8, 8)
Bias	Have bias			Have bias		
Activation Function	Sigmoid			Sigmoid		
Optimizer	SGD			SGD		
Epochs	1.E+03			1.E+04		
代號	A	B	C	D	E	F

代號	Loss	Accuracy
A		
B		
C		

D		
E		
F		

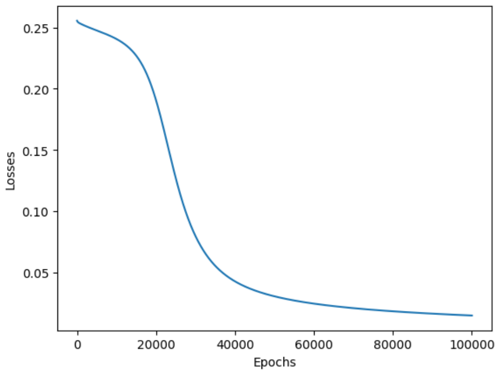
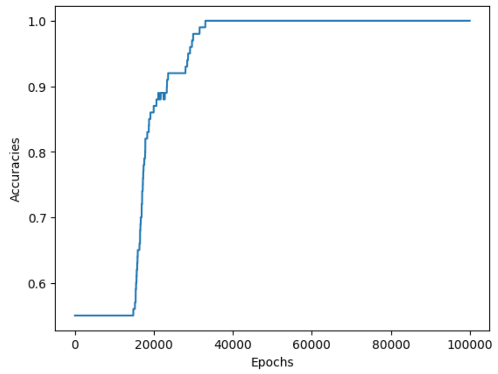
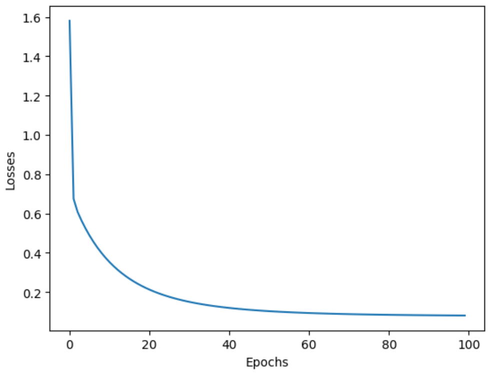
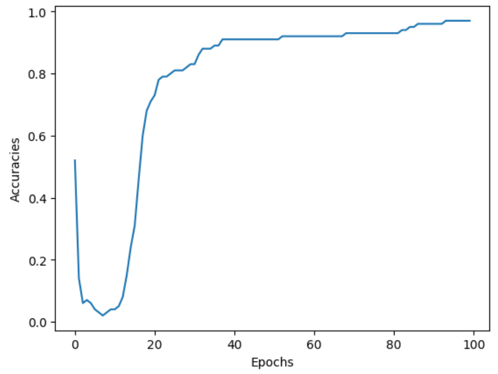
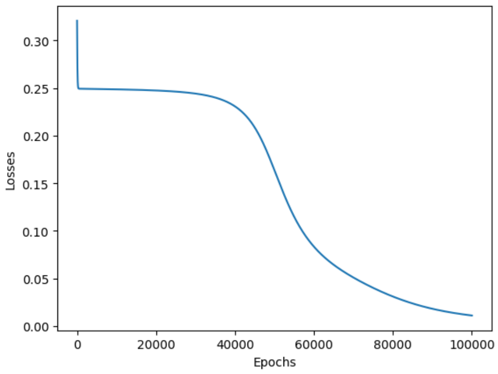
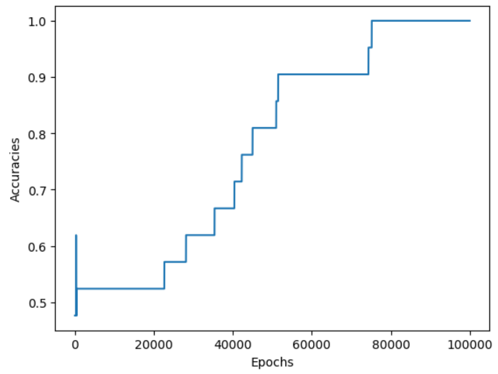
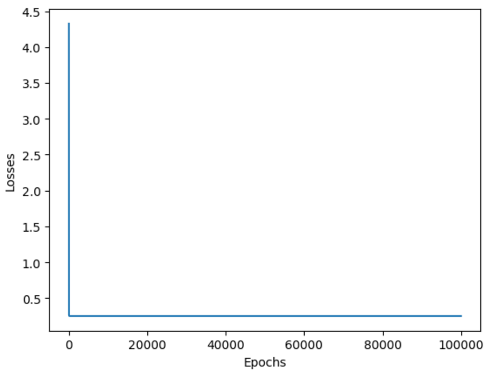
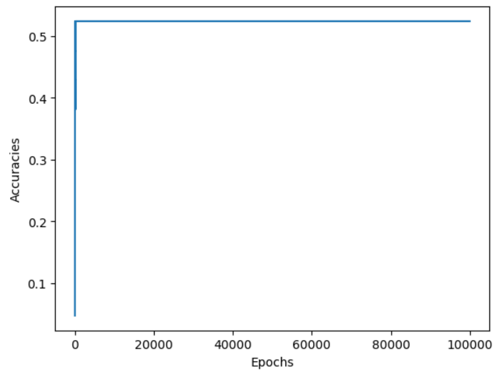
在這個部分可以看到，無論是線性資料或是 XOR，適度增加 neurons 數量皆會導致較快收斂。

(C) Without activation function

實驗設計

Dataset	Linear		XOR	
Learning rate	0.0001		0.001	
Number of hidden units	(4, 4)		(4, 4)	
Bias	Have bias		Have bias	
Activation Function	Sigmoid	None	Sigmoid	None
Optimizer	SGD		SGD	
Epochs	1.E+05	1.E+02	1.E+05	
代號	A	B	C	D

代號	Loss	Accuracy

A		
B		
C		
D		

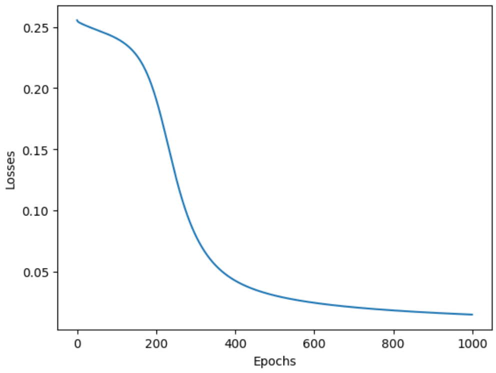
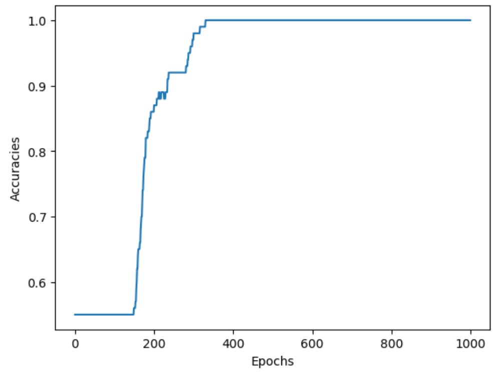
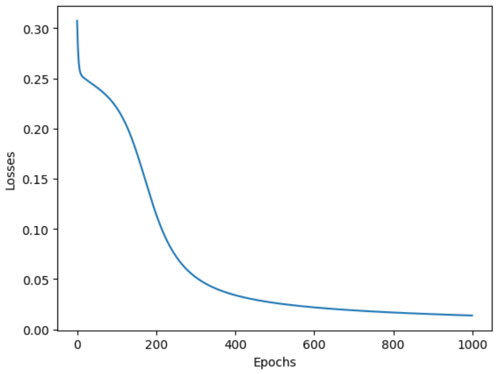
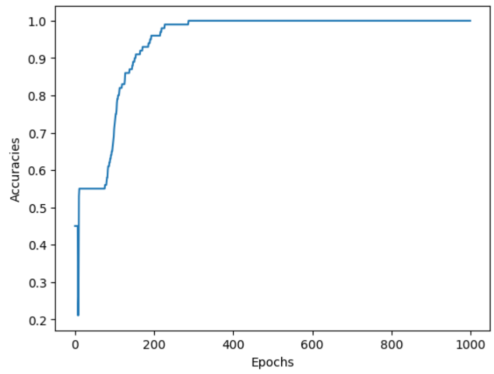
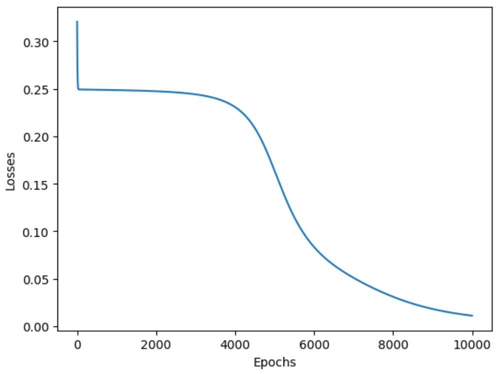
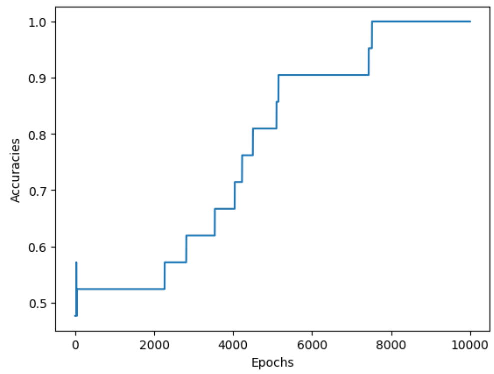
對於 linear dataset，不使用 activation function，會有較快的收斂速度，這是因為資料簡單且線性，而不使用 activation function 的斜率較大能較快更新參數。

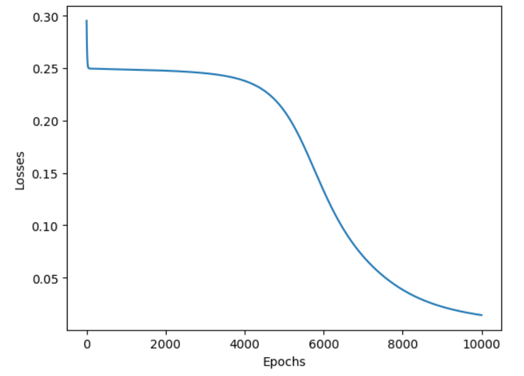
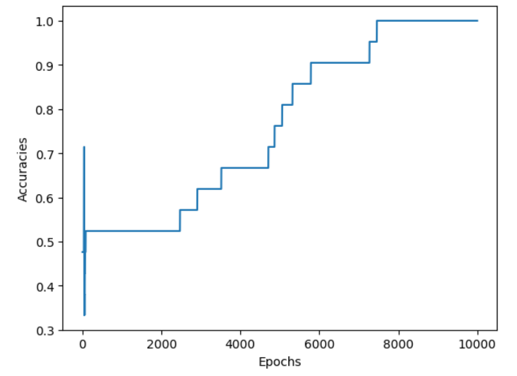
對於 XOR dataset，由於是非線性資料，不使用 Activation function 無法處理非線性資料，因此 D 的 loss 永遠無法收斂。

(D)Without bias

實驗設計

Dataset	Linear		XOR	
Learning rate	0.01		0.01	
Number of hidden units	(4, 4)		(4, 4)	
Bias	Have bias	No bias	Have bias	No bias
Activation Function	Sigmoid		Sigmoid	
Optimizer	SGD		SGD	
Epochs	1.E+03		1.E+04	
代號	A	B	C	D

代號	Loss	Accuracy
A		
B		
C		

D		

這部份想比較的是參數僅有 weight 與 weight+bias 的比較，如 A、B 圖可以看到捨去 bias 後收斂速度反而加快，這是由於原始資料本就是簡單的線性分布，可以只簡單的使用權重表示，因此只保留比較參數加快收斂速度，但是反之 XOR dataset 則不然，並沒有顯著改變。

5. Extra

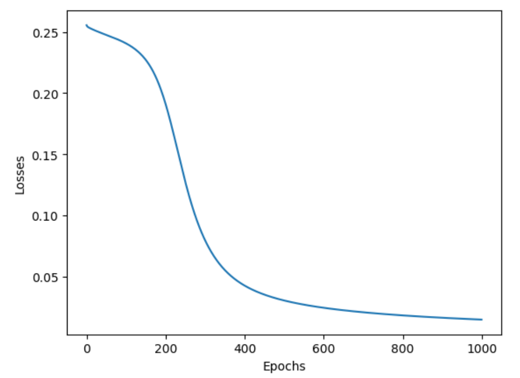
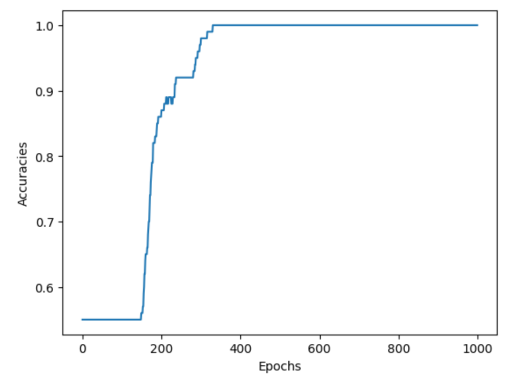
(A) Optimizers

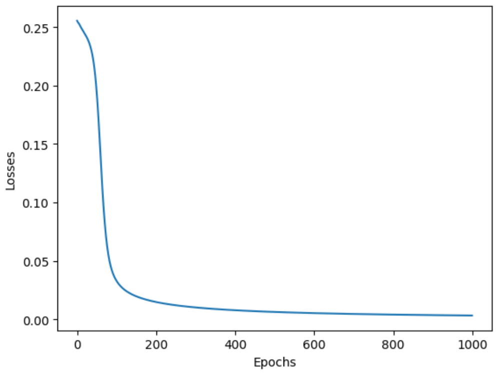
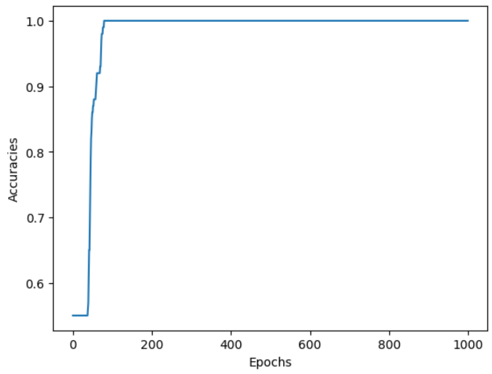
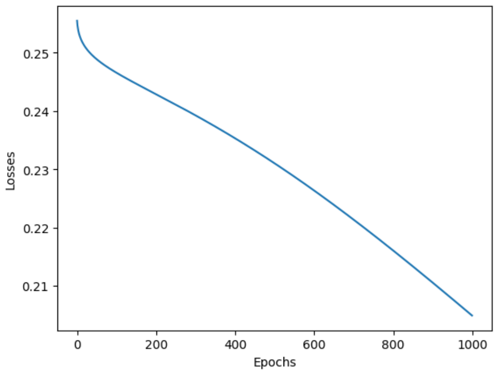
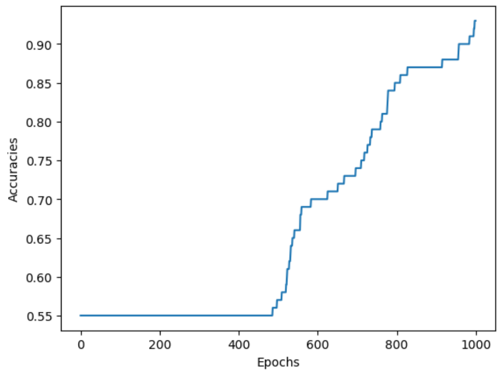
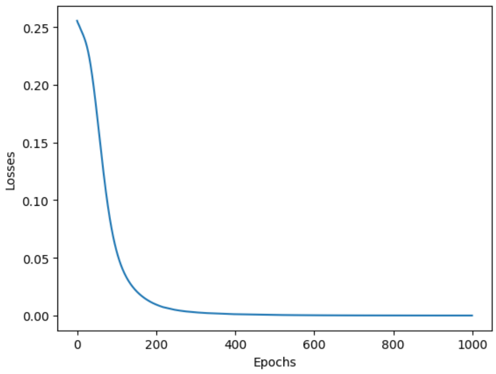
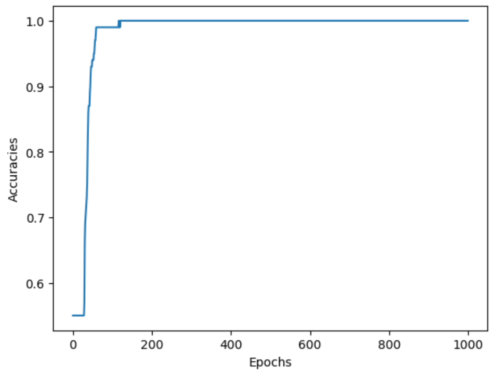
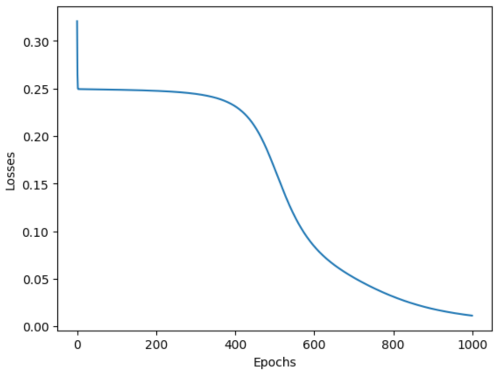
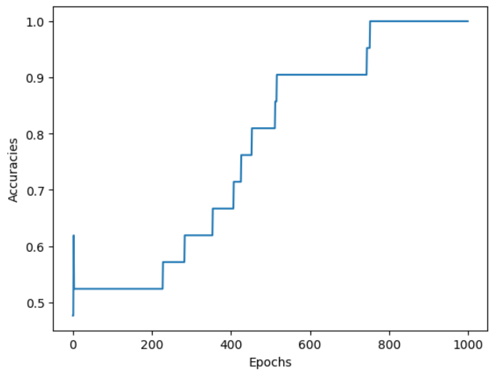
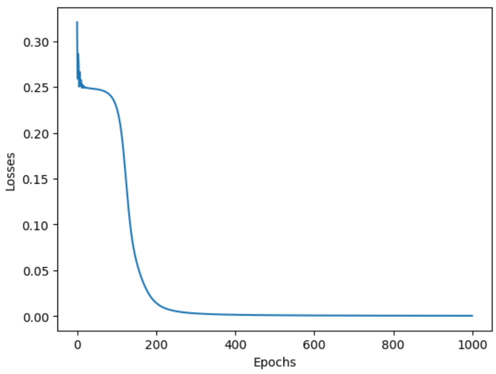
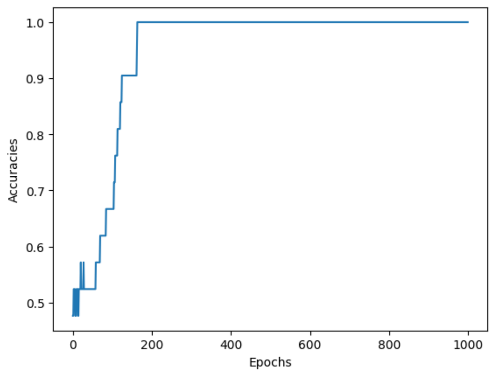
Dataset	Linear				XOR			
Learning rate	0.01				0.1			
Number of hidden units	(4, 4)				(4, 4)			
Bias	Have bias				Have bias			
Activation Function	Sigmoid				Sigmoid			
Optimizer	SGD	Momentum GD	Adagrad	Adam	SGD	Momentum GD	Adagrad	Adam
Epochs	1.E+03				1.E+03			
代號	A	B	C	D	E	F	G	H

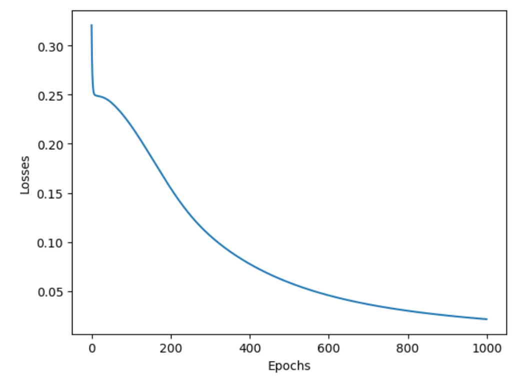
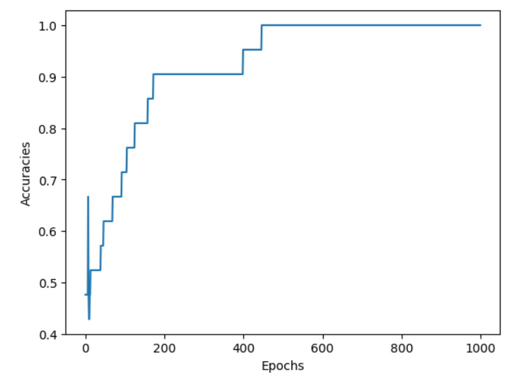
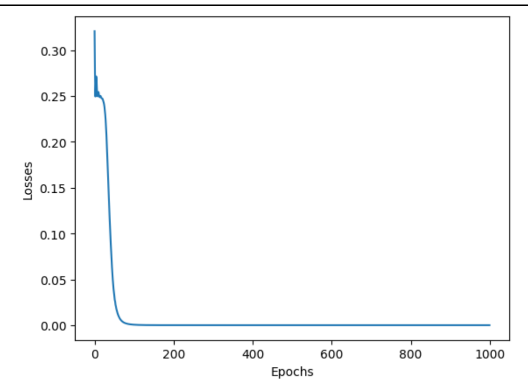
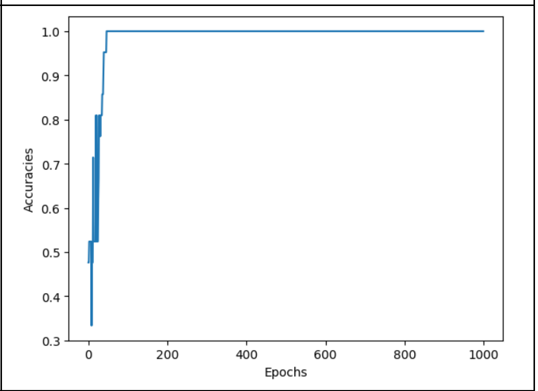
Momentum: $\beta = 0.8$

Adagrad: $\epsilon = 1e - 8$

Adam: $\beta_1 = 0.8, \beta_2 = 0.9, \epsilon = 1e - 8$

代號	Loss	Accuracy
A		

<table>
<tr><td>B</td><td></td><td></td></tr>
<tr><td>C</td><td></td><td></td></tr>
<tr><td>D</td><td></td><td></td></tr>
<tr><td>E</td><td></td><td></td></tr>
<tr><td>F</td><td></td><td></td></tr>
</table>

G		
		

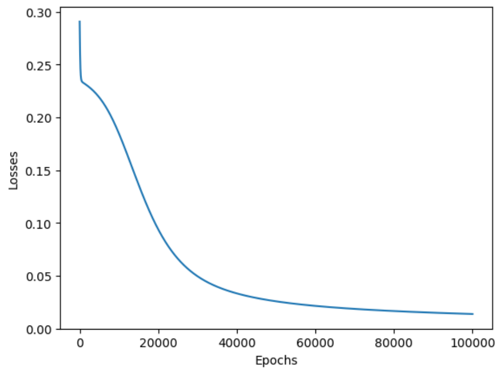
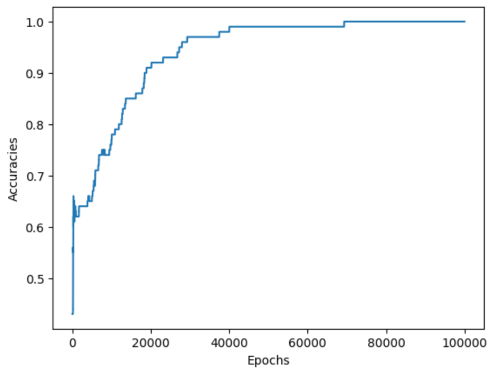
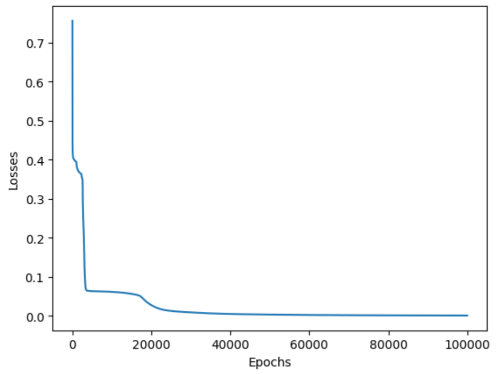
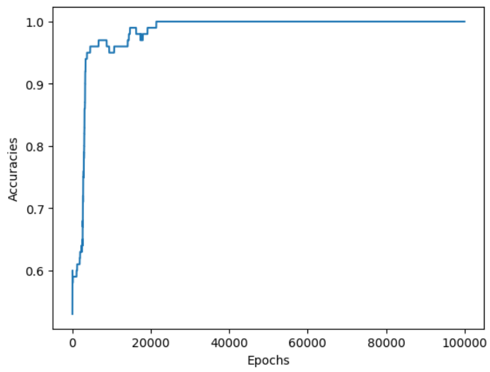
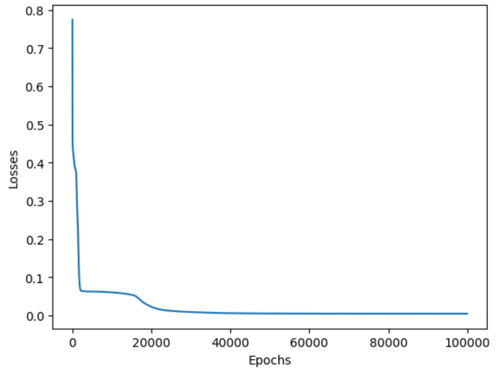
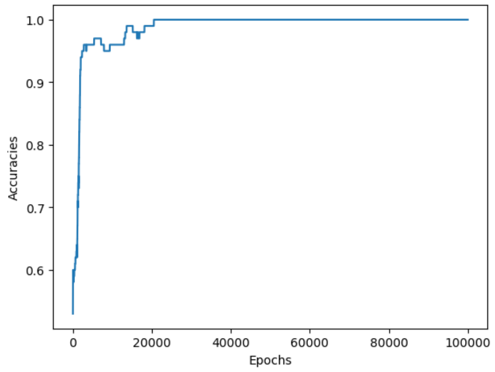
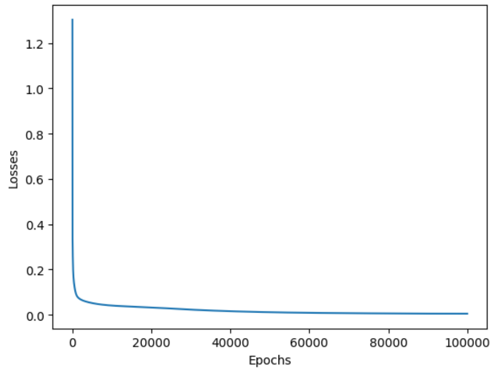
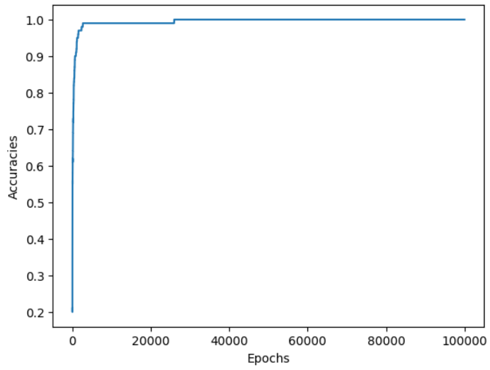
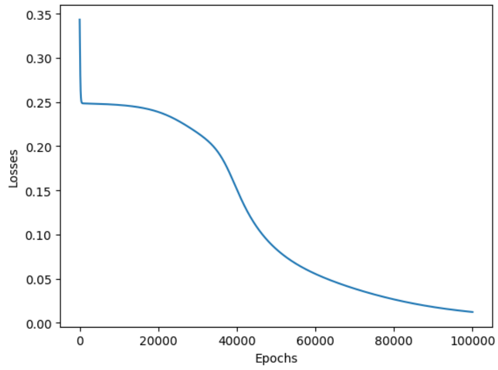
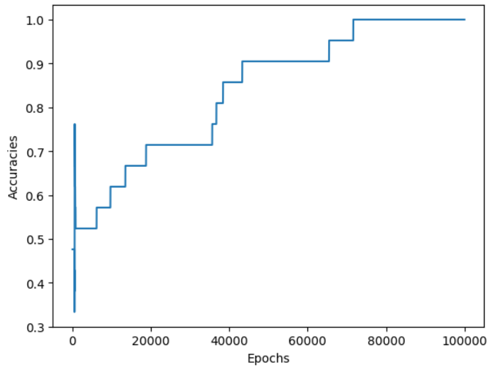
Momentum, Adam 相較 SGD 而言可以很快速地幫助收斂，而 adagrad 由於訓練後期 learning rate 逐漸降低，反而會導致在相同參數下的訓練速度降低，若初始 learning rate 較高也會有還可以的收斂表現。

(B) Activation functions

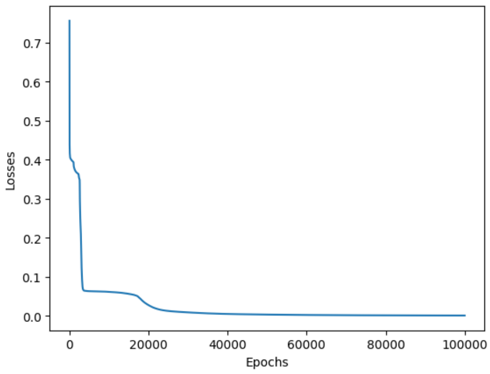
實驗設計

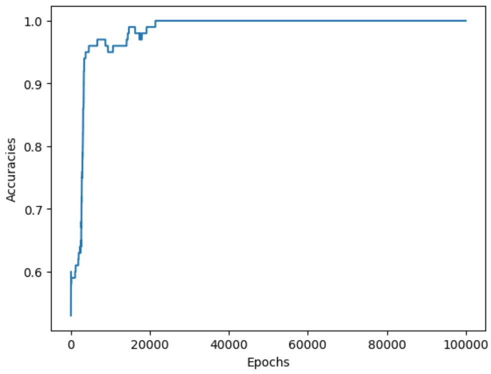
Dataset	**Linear**				**XOR**			
Learning rate	0.0001				0.001			
Number of hidden units	(4, 4)				(4, 4)			
Bias	Have bias				Have bias			
Activation Function	Sigmoid	ReLU	Leaky ReLU	tanh	Sigmoid	ReLU	Leaky ReLU	tanh
Optimizer	SGD				SGD			
Epochs	1.E+05				1.E+05			
代號	A	B	C	D	E	F	G	H

Leaky ReLU -x 方向斜率 0.01

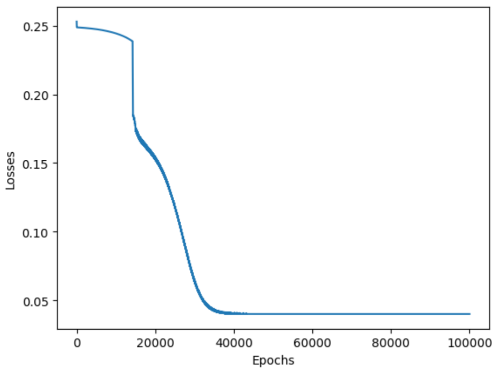
<table>
<tr><td>代號</td><td>Loss</td><td>Accuracy</td></tr>
<tr><td>A</td><td>
</td><td>
</td></tr>
<tr><td>B</td><td>
</td><td>
</td></tr>
<tr><td>C</td><td>
</td><td>
</td></tr>
<tr><td>D</td><td>
</td><td>
</td></tr>
<tr><td>E</td><td>
</td><td>
</td></tr>
</table>

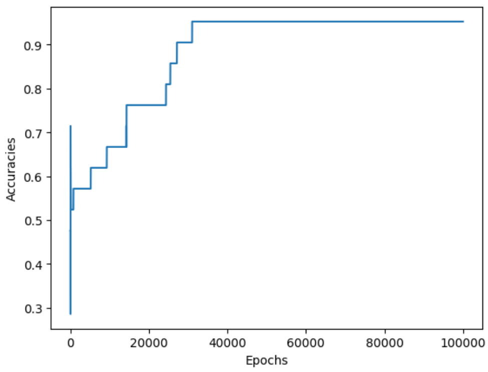
F



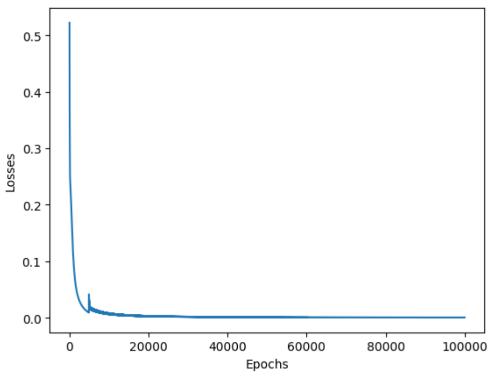


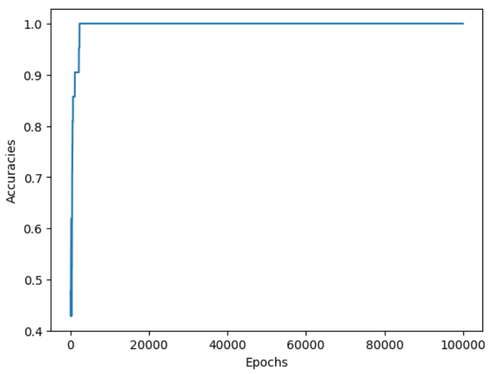
G





H





可以看到無論是 ReLU、Leaky ReLU、Tanh 相較 sigmoid 均有較快的收斂速度，在實驗中 tanh 的速度最快，但計算代價也最大，需要多次計算指數，ReLU 計算則最快。