

# Deep Learning and Practice Lab 1

311605011

黃品振

## 1. Introduction:

這次的實作是自己設計一個兩層的 neural network。需要先利用 forward pass 將輸入資料向前傳送得到一個預測答案，再透過 Loss function 得到正確答案與預測答案之間的差距，接著透過 backpropagation 得到 loss 對 weight 的偏微結果，再乘以 learning rate 去更新每層網路的權重，就可以使預測結果與真實結果更接近。除此之外我們也可以透過調整每層神經元數量、learning rate、資料數量、以及不同的激活函數來觀察結果。

## 2. Experiment setups:

### A. Sigmoid functions:

```
B. def sigmoid(x):  
C.     return 1 / (1+np.exp(-x))  
D.  
E. def d_sigmoid(x):  
F.     return np.multiply(x, 1.0 - x)
```

上圖為 sigmoid function 的實作，sigmoid function 在這的作用是讓輸入資料有非線性的變化，才得以分類出 XOR 類型的資料。而 sigmoid 用在 forward pass 中，而 d\_sigmoid 會再 backpropagation 的微分部分用到。

### B. Neural network:

```
A.     def forward(self, x, function_type = "sigmoid"):  
B.         self.x = x  
C.         self.z1 = self.activation((np.dot(x, self.w0)), function_type)  
D.         self.z2 = self.activation((np.dot(self.z1, self.w1)), function_type)  
E.         self.y = self.activation((np.dot(self.z2, self.w2)), function_type)
```

```

F.         self.o0 = np.dot(self.x, self.w0)
G.         self.o1 = np.dot(self.z1, self.w1)
H.         self.o2 = np.dot(self.z2, self.w2)
I.
J.         return self.y
K.
L.     def backward(self, groundtruth, function_type):
M.         dLdy = (groundtruth - self.y)
N.         dLdydw2 = self.d_activation(self.y, function_type) * dLdy
O.         self.dLdw2 = np.dot(self.z2.T, dLdydw2)
P.
Q.         dLdydz2w2 = (dLdydw2.dot(self.w2.T)) * self.d_activation(self.z2, function_type)
R.         self.dLdw1 = np.dot(self.z1.T, dLdydz2w2)
S.
T.         dLdydz1w1 = (dLdydz2w2.dot(self.w1.T)) * self.d_activation(self.z1, function_type)
U.         self.dLdw0 = np.dot(self.x.T, dLdydz1w1)
V.
W.     def optimizer(self, lr = 0.01):
X.         self.w2 += lr * self.dLdw2
Y.         self.w1 += lr * self.dLdw1
Z.         self.w0 += lr * self.dLdw0

```

上圖是我的 neural network 主架構，主要是先將輸入透過 forward 往前送，再利用 backpropagation 取得正確結果與預測結果之差異，然後再利用 optimizer 去更新權重，最後使預測結果趨近於正確結果。

## C. Backpropagation:

```

def backward(self, groundtruth, function_type):
    dLdy = (groundtruth - self.y)

    dLdydw2 = self.d_activation(self.y, function_type) * dLdy
    self.dLdw2 = np.dot(self.z2.T, dLdydw2)

    dLdydz2w2 = (dLdydw2.dot(self.w2.T)) * self.d_activation(self.z2, function_type)
    self.dLdw1 = np.dot(self.z1.T, dLdydz2w2)

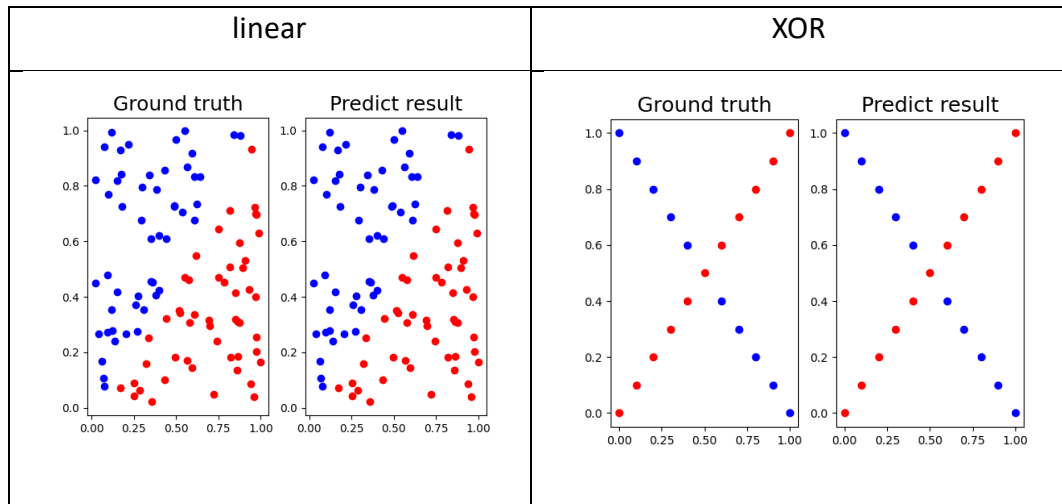
```

```
dLdydz1w1 = (dLdydz2w2.dot(self.w1.T)) * self.d_activation(self.z1, function_type)
self.dLdw0 = np.dot(self.x.T, dLdydz1w1)
```

上圖是我的 backpropagation 的實作，我的算法是從後面往前做偏微，取得各層對權重的偏微分。

### 3. Results of my testing:

#### A. Screenshot and comparison figure:



我的正確結果-預測結果比對圖如上，可以發現線性以及 XOR 的預測結果皆與正確結果吻合。

B. Show the accuracy of my prediction:

Linear:

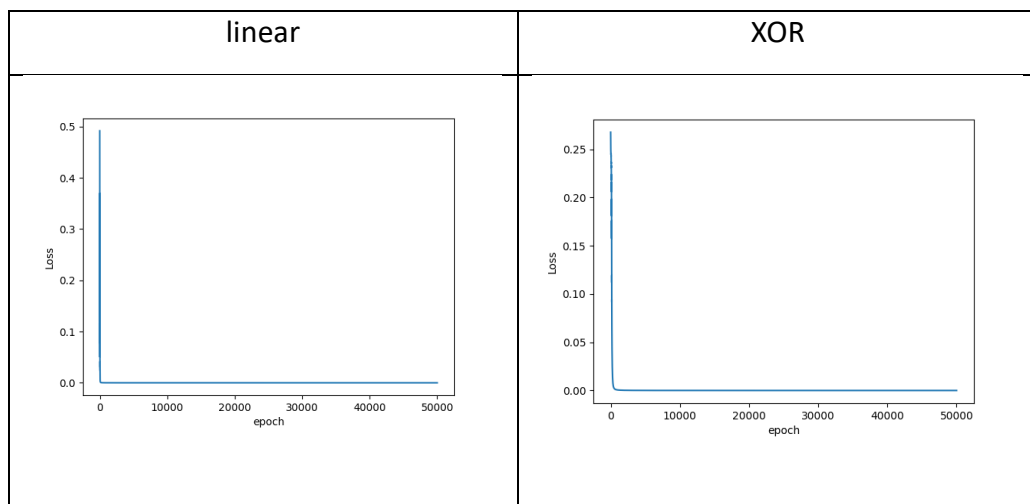
```
prediction: [2.67819003e-05] [9.99999857e-01]
[[9.99999790e-01] [9.99999954e-01] [9.99999354e-01]
[9.99993720e-01] [1.64814848e-07] [9.99999929e-01]
[1.78295384e-07] [1.67813194e-07] [9.99924715e-01]
[4.55858098e-05] [1.90256930e-07] [3.97299716e-07]
[1.88597101e-07] [4.31917223e-07] [9.99999060e-01]
[9.99992379e-01] [2.17834586e-07] [9.95406073e-01]
[9.99999960e-01] [4.17070017e-06] [1.81485907e-07]
[9.99999914e-01] [1.25107700e-05] [5.05055375e-07]
[4.53523454e-07] [3.29132361e-04] [4.05471768e-04]
[1.84099839e-07] [9.99999858e-01] [9.99999967e-01]
[3.33544784e-07] [9.99999537e-01] [9.99999588e-01]
[9.97827151e-01] [9.9999952e-01] [9.99979867e-01]
[1.30312493e-05] [2.35794073e-07] [1.91381290e-05]
[9.99999961e-01] [9.99999857e-01] [2.34342688e-07]
[3.18066933e-07] [9.99614303e-01] [1.87477041e-07]
[2.44499706e-07] [1.67126942e-07] [1.94747687e-07]
[2.23748479e-06] [2.93833206e-07] [3.70143675e-03]
[9.99985853e-01] [9.99999966e-01] [9.99999963e-01]
[2.10003879e-07] [9.99999956e-01] [1.76052356e-07]
[3.95617370e-07] [1.70173674e-07] [9.99587081e-01]
[5.36367657e-07] [3.39451888e-07] [accuracy = 100.0%]
[9.99997208e-01] [9.99999969e-01]
[9.98805023e-01] [9.99998710e-01]
[9.99999920e-01] [2.39141491e-06]
[9.99999906e-01] [2.82865474e-07]
[9.99999969e-01] [9.9999959e-01]
[2.90961489e-03] [3.10113241e-07]
[2.38320638e-03] [9.99999198e-01]
[9.99999966e-01] [9.99999967e-01]
[9.99768774e-01] [9.61581375e-06]
[9.99999969e-01] [9.9999952e-01]
[9.99999951e-01] [9.9999951e-01]
[9.99999951e-01] [9.99999852e-01]
[9.99999967e-01] [2.31312370e-07]
[2.72722529e-07] [1.64910008e-07]
[3.41175686e-06] [9.99990324e-01]
[4.19986375e-07] [2.53992733e-07]
[9.9999786e-01] [9.9999916e-01]
```

XOR:

```
predction:
[[0.00161964]
[0.99942026]
[0.00202364]
[0.99935592]
[0.0024434 ]
[0.99919279]
[0.00277317]
[0.99869342]
[0.00293332]
[0.99389613]
[0.00291258]
[0.00275517]
[0.99486333]
[0.00252177]
[0.99908991]
[0.00226218]
[0.99903809]
[0.00200762]
[0.99889009]
[0.00177388]
[0.99877571]]
accuracy = 100.0%
```

由上圖可以發現準確率皆為 100%，原因應該是因為資料量較少經過多次訓練後都能得到理想的預測。

### C. Learning curve(loss, epoch curve):

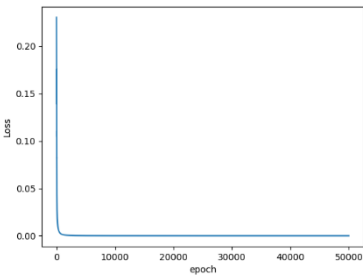
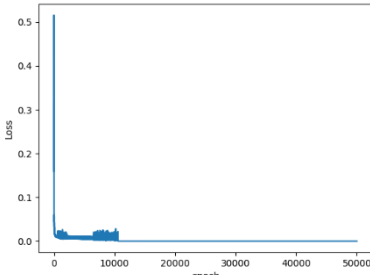
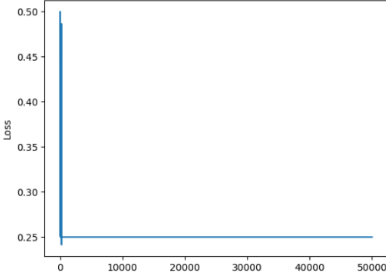
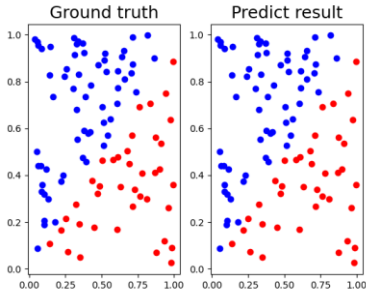
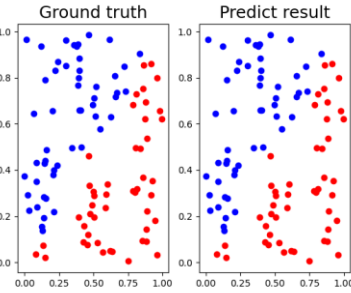
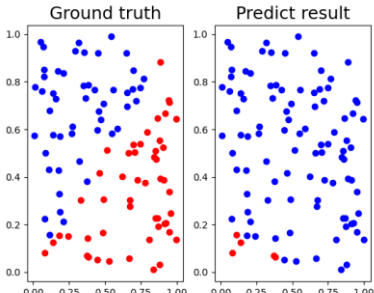


由上圖可以發現 loss 都收斂的相當低。

## 4. Discussion:

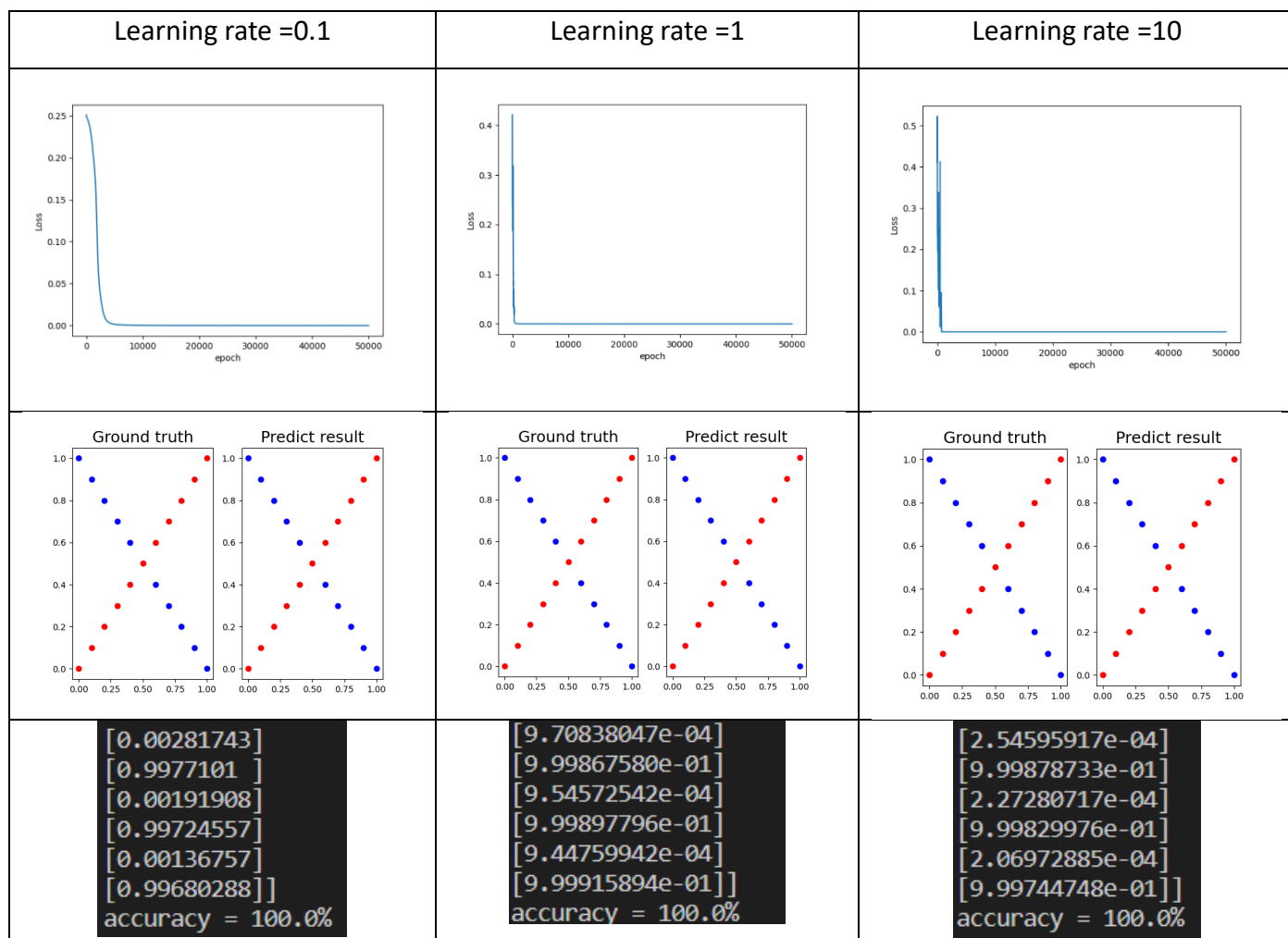
### A. Try different learning rate:

Linear:

Learning rate =0.1	Learning rate =1	Learning rate =10
		
		
<pre>[4.76329960e-08] [9.99999916e-01] [4.74968553e-08] [4.60418479e-08] [9.99999990e-01] [2.85501396e-03]] accuracy = 100.0%</pre>	<pre>[8.74141331e-05] [9.99999937e-01] [9.99999958e-01] [9.99997548e-01] [4.08262500e-05] [5.69204731e-05]] accuracy = 100.0%</pre>	<pre>[0.5] [0.5] [0.5] [0.5] [0.5] [0.5]] accuracy = 53.0%</pre>

由上圖以及個人在實驗時發現，loss curve 在 learning rate = 0.1 時最為平滑，  
且 learning rate 太大時，會導致結果相當不穩定。

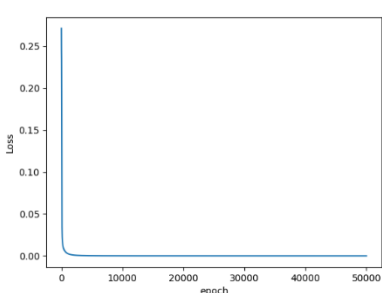
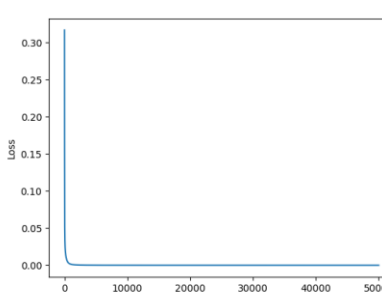
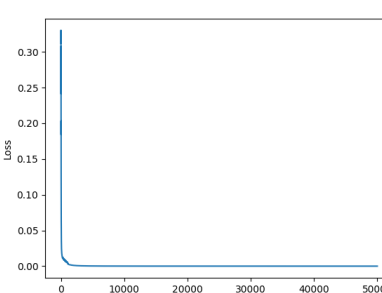
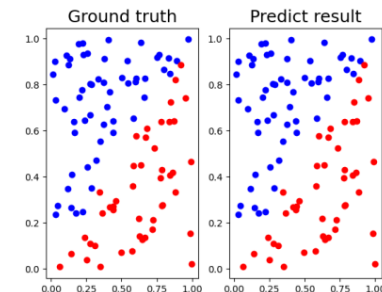
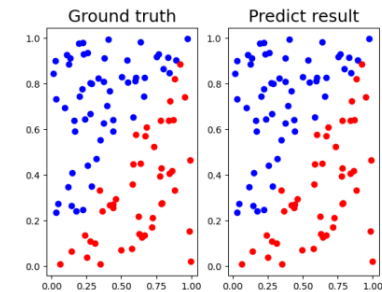
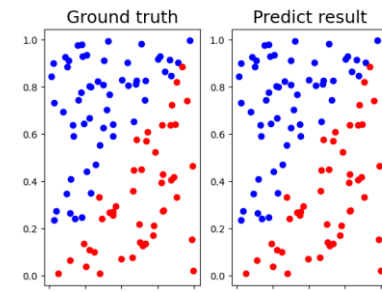
XOR:



上圖可以發現，loss curve 在 learning rate = 0.1 時最為平滑，在 learning rate 逐漸變大時曲線有時候會震盪，在 learning rate = 10 時，實驗結果相當不穩定，accuracy 的結果有 50%-100%都有。

## B. Try different numbers of hidden unit:

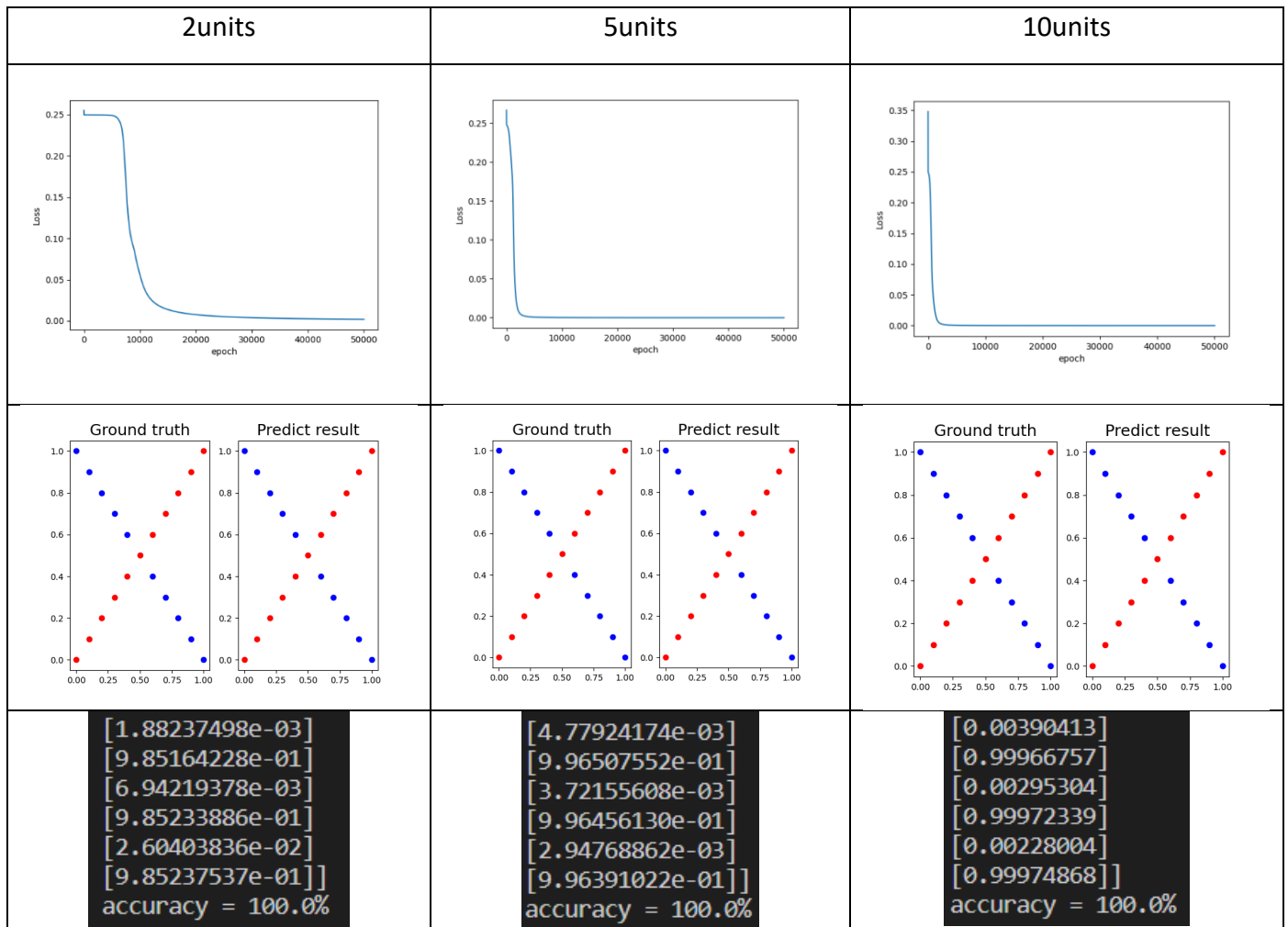
Linear:

2units	5units	10units
		
		
<pre>[1.10473493e-05] [1.12466416e-05] [9.99999522e-01] [1.10084375e-05] [5.59701171e-03] [9.97333263e-01] accuracy = 100.0%</pre>	<pre>[9.99999925e-01] [9.90476103e-01] [9.99996924e-01] [1.34753332e-07] [9.99999925e-01] [9.99999701e-01] accuracy = 100.0%</pre>	<pre>[2.85541594e-08] [9.99999985e-01] [9.99999998e-01] [8.13904180e-11] [6.98321232e-09] [9.19785355e-11] accuracy = 100.0%</pre>

由上圖實驗結果可以發現，在 hidden layer 每層神經元數量在 2、5、10 時表現都很好，猜測是因為 linear 是簡單的問題，所以不用太多神經元就可以表現得很好。



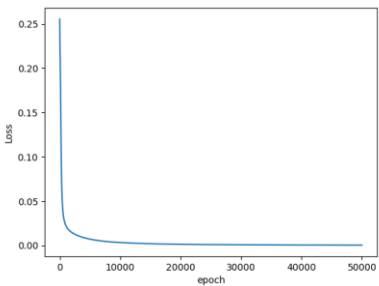
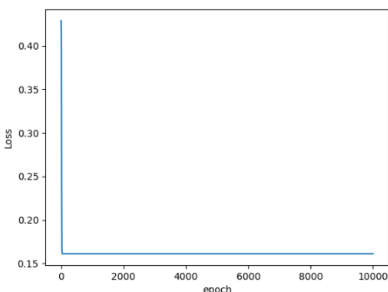
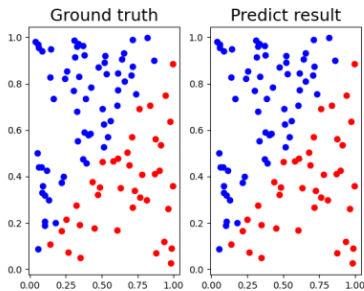
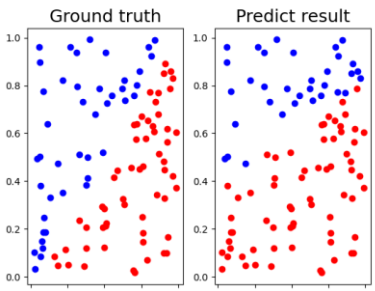
XOR:



由上圖實驗結果可以發現，當 hidden later unit 變多時，收斂速度變比較快，尤其是從 2 個 unit 變成 5 個 unit 時的差別最為明顯。顯現出 unit 較多效果越佳的結果。

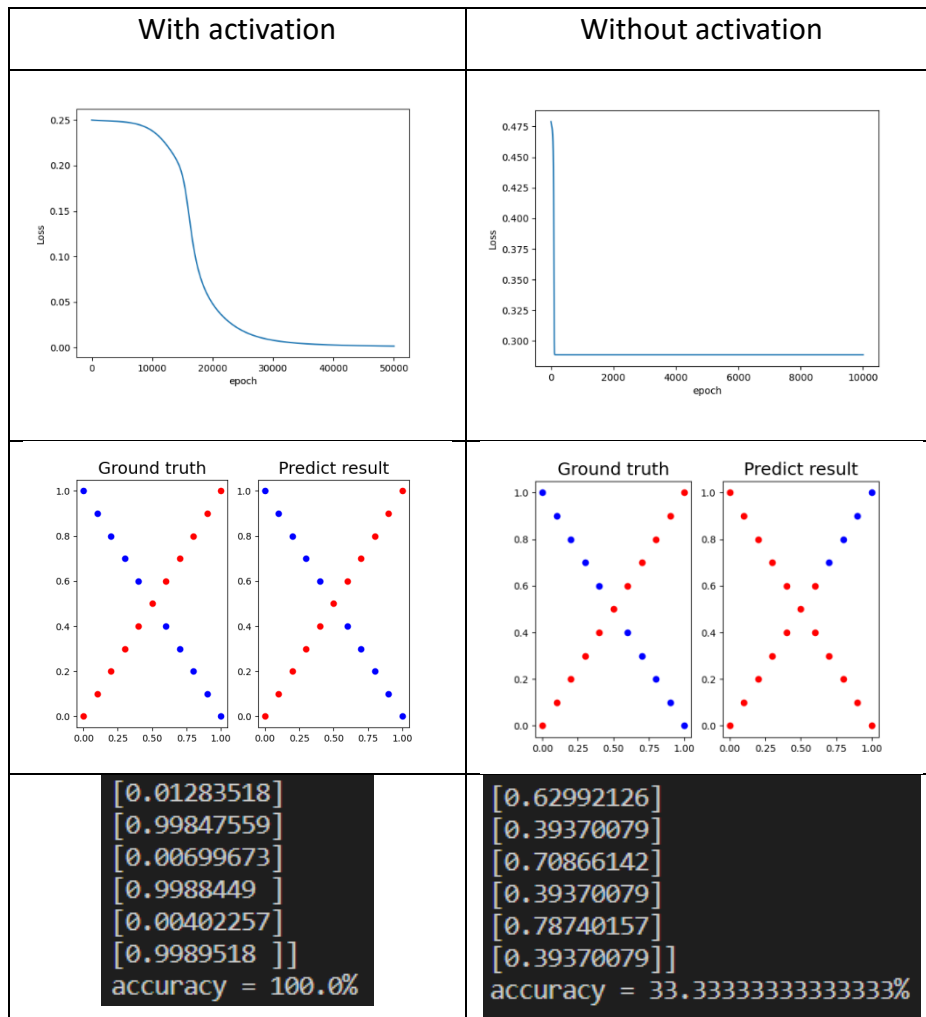
## C. Try without activation function:

### Linear:

With activation	Without activation
	
	
<pre>[4.76329960e-08] [9.99999916e-01] [4.74968553e-08] [4.60418479e-08] [9.99999990e-01] [2.85501396e-03]] accuracy = 100.0%</pre>	<pre>[ 0.571072 ] [ 0.42152194] [ 0.02441732] [ 0.59758779] [ 0.41434135] [ 0.58727519]] accuracy = 78.0%</pre>

由實驗可以發現，在同樣的 unit 數量(5, 5)、以及同樣的 learning rate(0.01)的狀況下，無 activation function 的 loss 收斂得比較快一點點，但是準確率也較差。

XOR:

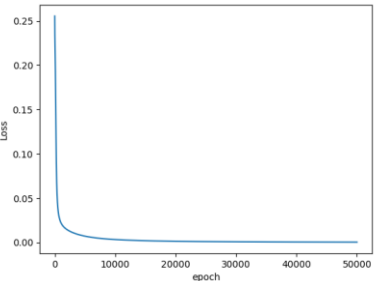
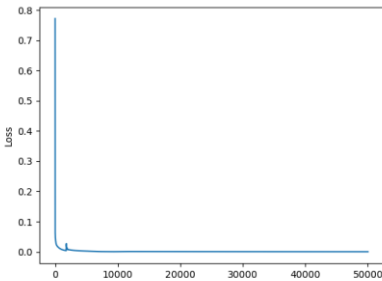
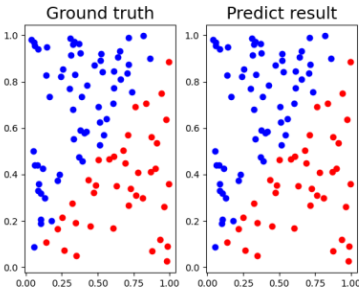
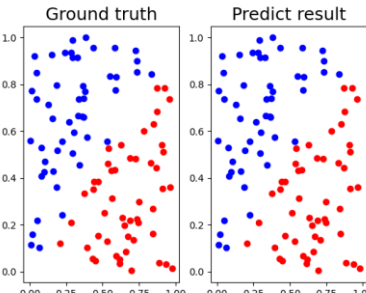


由上圖可以很明確地發現，without activation 無法解決非線性分類的問題。

## 5. Extra:

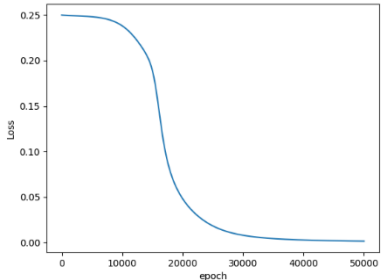
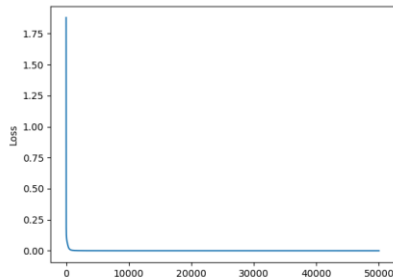
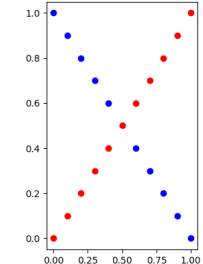
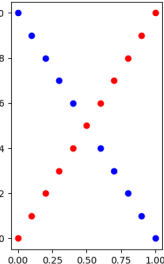
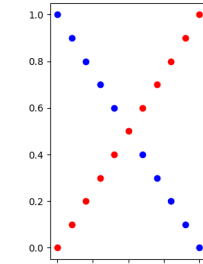
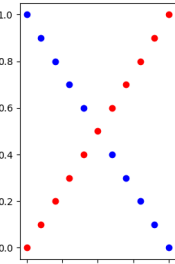
### A. Using $\tanh(x)$ :

Linear:

sigmoid	tanh
	
	
<pre>[4.76329960e-08] [9.99999916e-01] [4.74968553e-08] [4.60418479e-08] [9.99999990e-01] [2.85501396e-03] accuracy = 100.0%</pre>	<pre>[ 0.99994868] [ 0.99995018] [-0.02131515] [-0.02139269] [ 0.99995116] [ 0.99995441] accuracy = 100.0%</pre>

由上圖實驗發現  $\tanh$  在同樣的 unit 數量(5, 5)、以及同樣的 learning rate(0.01)的狀況下,  $\tanh$  會較快收斂且預測結果一樣是 100%。

XOR:

sigmoid	tanh
	
<div> <div>Ground truth</div>  </div> <div> <div>Predict result</div>  </div>	<div> <div>Ground truth</div>  </div> <div> <div>Predict result</div>  </div>
<pre>[0.01283518] [0.99847559] [0.00699673] [0.9988449 ] [0.00402257] [0.9989518 ]] accuracy = 100.0%</pre>	<pre>[0.00729954] [0.9999312 ] [0.00638832] [0.99989906] [0.00572596] [0.99979548]] accuracy = 100.0%</pre>

Tanh 收斂較快，預測結果也很優良，一樣是 100%。