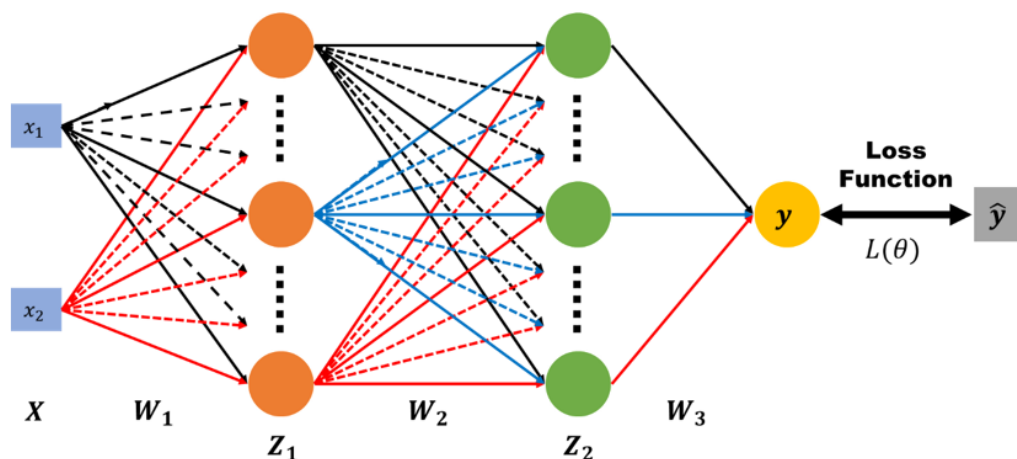


Lab1: Back-propagation

311511043 李承翰

1. Introduction

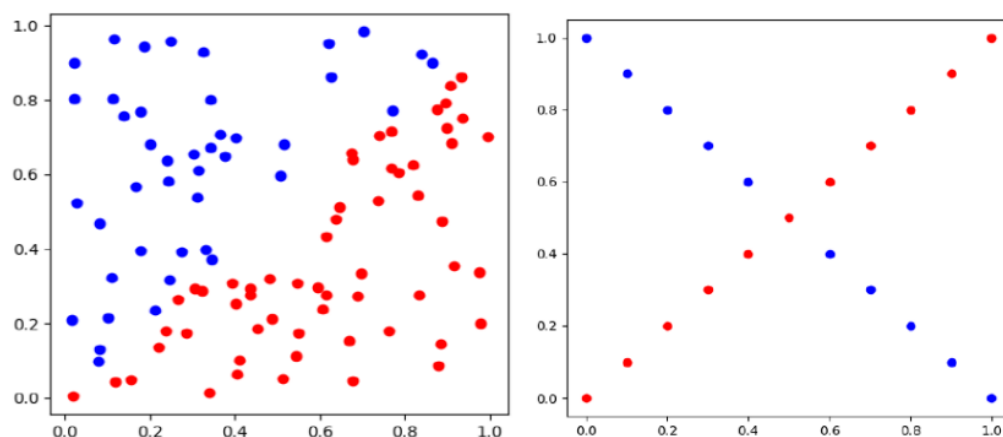


這次 lab 我們要實作出一個有兩層 hidden layers 的神經網路，並且分別對兩種二元分類問題進行訓練，將分類結果、訓練過程、預測結果等數據可視化，用來探討神經網路實作中的各項細節，包含 back-propagation 如何實作、如何利用 chain-rule 來計算 gradient 以及 使用不同 optimizer 和 activation functions 會帶來怎樣的結果

實作流程:

1. 初始化神經網路(初始化各層權重及設定 hyper-parameters)
2. 生成訓練用資料
3. Forward propagation 得出 prediction
4. 計算誤差
5. Back propagation
6. 更新權重
7. 重複直到設定的 epoch 或者 loss、accuracy 收斂

實驗中使用的 input data:



(1) Linear

(2) XOR

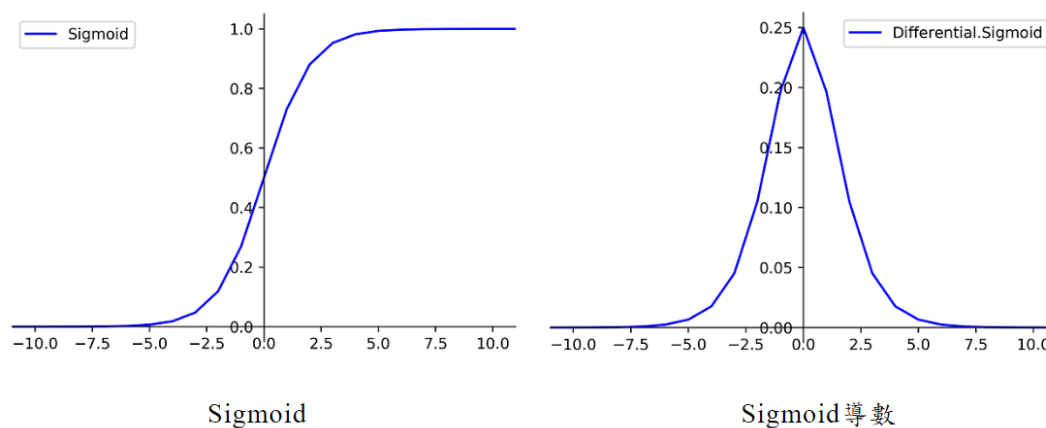
圖中點的顏色分別代表 $Y = 0$ 、 1

2. Experiment setups

A. Sigmoid functions

Sigmoid functions 是深度學習領域開始時最常見的一種 activation function，因為他是一個便於求導數的平滑函數

其方程式為: $f(x) = \frac{1}{1+e^{-x}}$ ，其微分為: $f'(x) = (1 - f(x)) * f(x)$



Sigmoid

Sigmoid 導數

從圖中我們可以發現因為此函數的導數最大值只有 0.25，所以在實作當中如果有多層隱藏層則可能會有梯度受到多層函數導數的影響而造成梯度消失的問題。不過在此次實驗當中因為我們只有兩層隱藏層，所以相對而言較不會受到這個問題影響。

此外，因為函數當中牽涉到指數函數的運算，因此也比較消耗計算資源。

B. Neural networks

神經網路的特色是利用神經元(節點)來做非線性的特徵轉換，依賴訓練資料來學習並隨著網路架構的深度提高其精確度。其運作方式為將輸入乘上 權重後，再經由激活函數輸出，一層一層反覆傳遞得到最後的結果。而訓練 的目標就是要出節點和節點之間的關係，也就是隱藏層的權重參數。這次 lab 由一個輸入層、兩個隱藏層以及一個輸出層來組成。

將輸入的 input 乘上權重 weight 之後加總並加上一個 bias 得到一個 score，使用這個 score 作為 sigmoid function 的輸入來得到一個 output。從 input 層開始往下經歷了兩層隱藏層的運算後到達 output 層，即為我們的神經網路對於 input 資料的預測結果，使用預測結果。使用此預測結果與誤差作為一個損失函數的 Input，則可得到這一次預測的 cost 是多少。而在這次實驗當中我們使用 MSE 來作為我們的損失函數，接著我們就會透過梯度下降的方法來修正我們的權重，如此一來就可以逐步降低 loss，使我們的預測結果愈發準確。

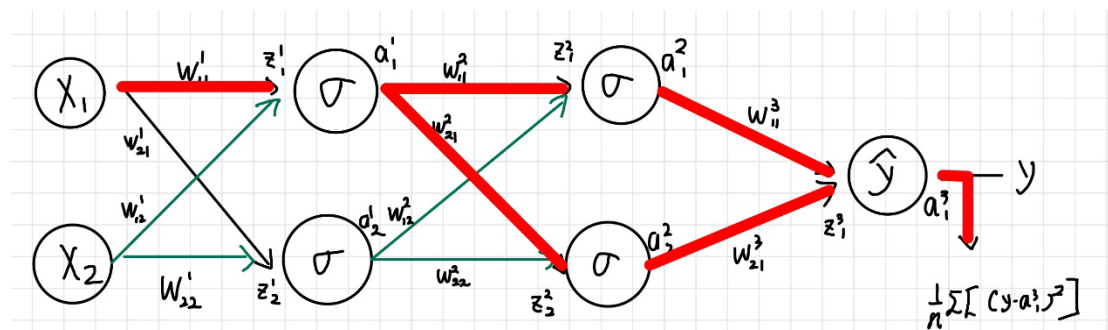
C. Backpropagation

在前面我們提到了梯度下降的方法來修正權重，就是引用了 back propagation 的概念，將誤差值作為回饋的資訊並且透過 chain rule 我們可以計算出 loss 對於每一個節點的梯度，接著我們可以算出節點中每一個權重的梯度利用這些梯度來更新我們的節點則可以使得誤差縮小。

以此次的模型為例：

若要更新 w_{11}^1 (第一層的第一個 neuron 到第二層的第一個 neuron 的 weight)

先列出此權重會影響的路徑(紅色路線)：



之後列出在路徑中的方程式關係，即可得到 $\frac{\partial J}{\partial w_{11}^1}$, $J = \frac{1}{n} \sum [(a_1^3 - \text{ground truth})^2]$

透過 chain rule 展開

$$\frac{\partial J}{\partial w_{11}^1} = \frac{\partial J}{\partial a_1^3} \cdot \left[\frac{\partial z_1^3}{\partial a_1^2} \frac{\partial a_1^2}{\partial z_1^2} \frac{\partial z_1^2}{\partial a_1^1} + \frac{\partial z_1^3}{\partial a_2^2} \frac{\partial a_2^2}{\partial z_2^2} \frac{\partial z_2^2}{\partial a_1^1} \right] \frac{\partial a_1^1}{\partial z_1^1} \frac{\partial z_1^1}{\partial w_{11}^1}$$

$$\frac{\partial J}{\partial w_{11}^1} = \frac{\partial J}{\partial a_1^3} \frac{\partial a_1^3}{\partial z_1^3} \cdot \left[\frac{\partial z_1^3}{\partial a_1^1} \frac{\partial a_1^2}{\partial z_1^2} \frac{\partial z_1^2}{\partial a_1^1} + \frac{\partial z_1^3}{\partial a_2^2} \frac{\partial a_2^2}{\partial z_2^2} \frac{\partial z_2^2}{\partial a_1^1} \right] \frac{\partial a_1^1}{\partial z_1^1} \frac{\partial z_1^1}{\partial w_{11}^1}$$

$$\frac{1}{n} (a_1^3 - \text{ground truth}) \cdot \sigma(z_1^3) [1 - \sigma(z_1^3)] \left\{ \sum_{\lambda=1}^2 w_{\lambda 1}^3 w_{\lambda 1}^2 \sigma(z_{\lambda}^2) [\sigma(1 - z_{\lambda}^2)] \right\} \sigma(z_1^1) [1 - \sigma(z_1^1)] x_1^1$$

得到了 $\frac{\partial J}{\partial w_{11}^1}$ 即可修正 w_{11}^1 的數值:

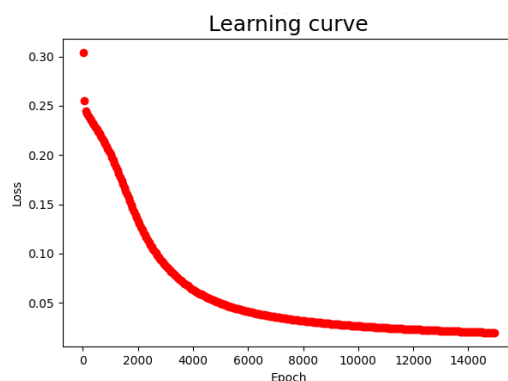
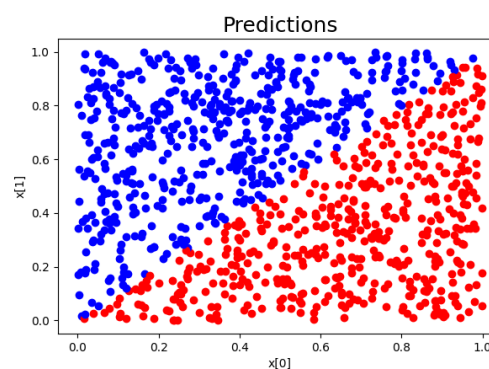
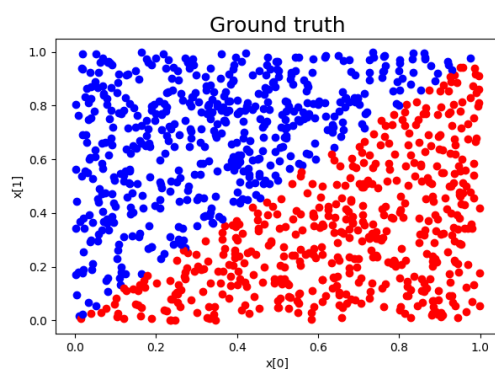
$$w_{11}^1 \leftarrow w_{11}^1 - \text{learning rate} \times \frac{\partial J}{\partial w_{11}^1}$$

以此類推即可修正所有的權重。

3. Result of testing

1. Linear

epoch	Learning rate	Data size	在隱藏層的 neuron 數量
15000	0.05	1000	4



```
Now training...
epoch 1000, loss: 0.3041, accuracy: 0.4890
epoch 2000, loss: 0.2016, accuracy: 0.8650
epoch 3000, loss: 0.1486, accuracy: 0.8890
epoch 4000, loss: 0.1004, accuracy: 0.9260
epoch 5000, loss: 0.0661, accuracy: 0.9640
epoch 6000, loss: 0.0493, accuracy: 0.9750
epoch 7000, loss: 0.0407, accuracy: 0.9860
epoch 8000, loss: 0.0356, accuracy: 0.9930
epoch 9000, loss: 0.0322, accuracy: 0.9940
epoch 10000, loss: 0.0297, accuracy: 0.9970
epoch 11000, loss: 0.0277, accuracy: 0.9970
epoch 12000, loss: 0.0261, accuracy: 0.9970
epoch 13000, loss: 0.0248, accuracy: 0.9980
epoch 14000, loss: 0.0236, accuracy: 0.9990
epoch 15000, loss: 0.0226, accuracy: 0.9990
```

```

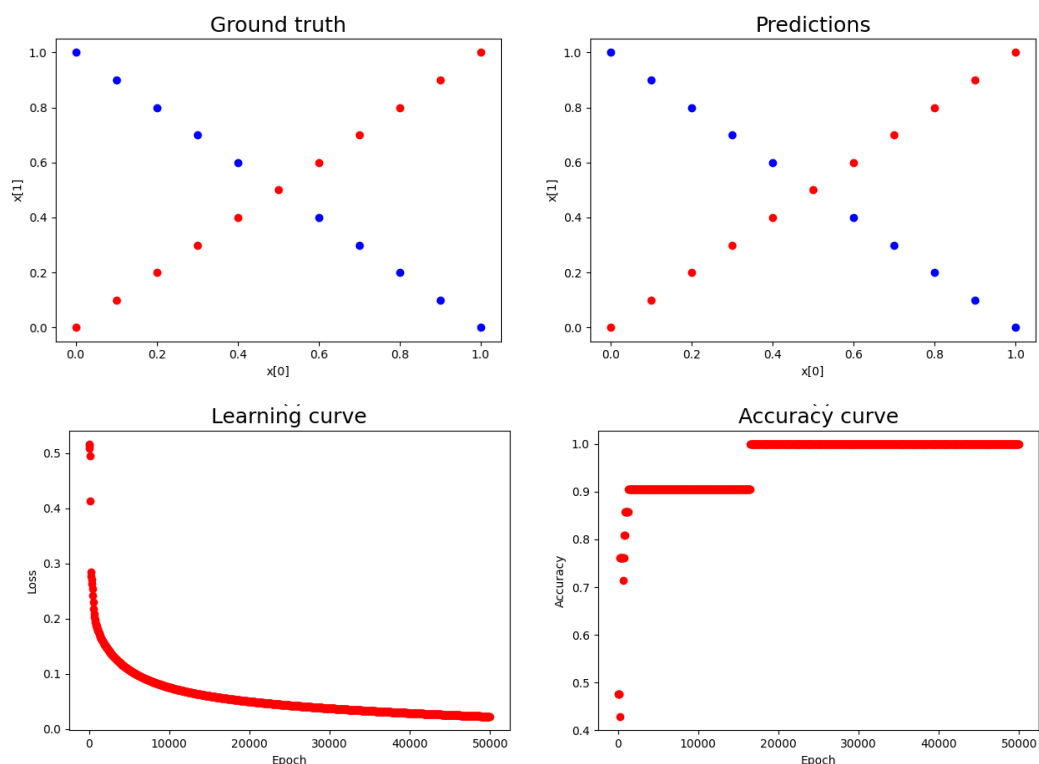
prediction: 0.9301, label: 1, actual: 1
prediction: 0.7630, label: 1, actual: 1
prediction: 0.9817, label: 1, actual: 1
prediction: 0.9841, label: 1, actual: 1
prediction: 0.7779, label: 1, actual: 1
prediction: 0.3077, label: 0, actual: 0
prediction: 0.3044, label: 0, actual: 0
prediction: 0.0066, label: 0, actual: 0
prediction: 0.8470, label: 1, actual: 1
prediction: 0.0041, label: 0, actual: 0
prediction: 0.0719, label: 0, actual: 0
prediction: 0.9803, label: 1, actual: 1
prediction: 0.0098, label: 0, actual: 0
prediction: 0.9027, label: 1, actual: 1
prediction: 0.9836, label: 1, actual: 1
prediction: 0.0116, label: 0, actual: 0
prediction: 0.8250, label: 1, actual: 1
prediction: 0.0144, label: 0, actual: 0
prediction: 0.8938, label: 1, actual: 1
prediction: 0.9808, label: 1, actual: 1
prediction: 0.7679, label: 1, actual: 1
prediction: 0.7648, label: 1, actual: 1
prediction: 0.9278, label: 1, actual: 1
Validate accuracies: 0.997
prediction: 0.0675, label: 0, actual: 0
prediction: 0.5095, label: 1, actual: 0
prediction: 0.0032, label: 0, actual: 0
prediction: 0.9054, label: 1, actual: 1
prediction: 0.9446, label: 1, actual: 1
prediction: 0.9660, label: 1, actual: 1
prediction: 0.9729, label: 1, actual: 1
prediction: 0.1002, label: 0, actual: 0
prediction: 0.0078, label: 0, actual: 0
prediction: 0.0896, label: 0, actual: 0
prediction: 0.0456, label: 0, actual: 0
prediction: 0.9799, label: 1, actual: 1
prediction: 0.0207, label: 0, actual: 0
prediction: 0.0037, label: 0, actual: 0
prediction: 0.0039, label: 0, actual: 0
prediction: 0.9025, label: 1, actual: 1
prediction: 0.8802, label: 1, actual: 1
prediction: 0.1155, label: 0, actual: 0
prediction: 0.9716, label: 1, actual: 1
prediction: 0.0815, label: 0, actual: 0
prediction: 0.9444, label: 1, actual: 1
prediction: 0.8092, label: 1, actual: 1
prediction: 0.0084, label: 0, actual: 0

```

因為在我們的 prediction 當中我們得到的是不會是單純的 0 或 1，所以我在判斷他的 label 的時候自行設定了一個 threshold value，當 prediction 出來的結果 ≥ 0.5 時，則我將他的 label 設定為 1，反之則將其設定為 0。

2. XOR

epoch	Learning rate	在隱藏層的 neuron 數量
30000	0.01	100



```

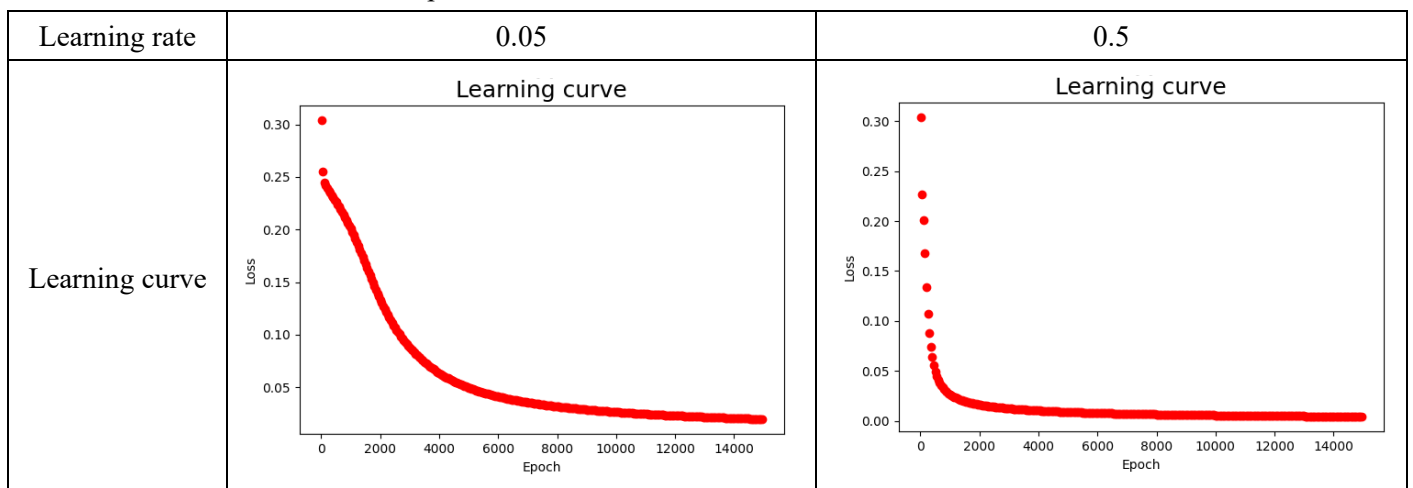
prediction: 0.0959, label: 0, actual: 0
prediction: 0.8861, label: 1, actual: 1
prediction: 0.1634, label: 0, actual: 0
prediction: 0.7882, label: 1, actual: 1
prediction: 0.2630, label: 0, actual: 0
prediction: 0.6690, label: 1, actual: 1
prediction: 0.3762, label: 0, actual: 0
prediction: 0.5641, label: 1, actual: 1
prediction: 0.4630, label: 0, actual: 0
prediction: 0.5023, label: 1, actual: 1
prediction: 0.4925, label: 0, actual: 0
prediction: 0.4597, label: 0, actual: 0
prediction: 0.5308, label: 1, actual: 1
prediction: 0.3771, label: 0, actual: 0
prediction: 0.6042, label: 1, actual: 1
prediction: 0.2707, label: 0, actual: 0
prediction: 0.6903, label: 1, actual: 1
prediction: 0.1721, label: 0, actual: 0
prediction: 0.7667, label: 1, actual: 1
prediction: 0.1006, label: 0, actual: 0
prediction: 0.8227, label: 1, actual: 1
Validate accuracies: 1.0

```

4. Discussion

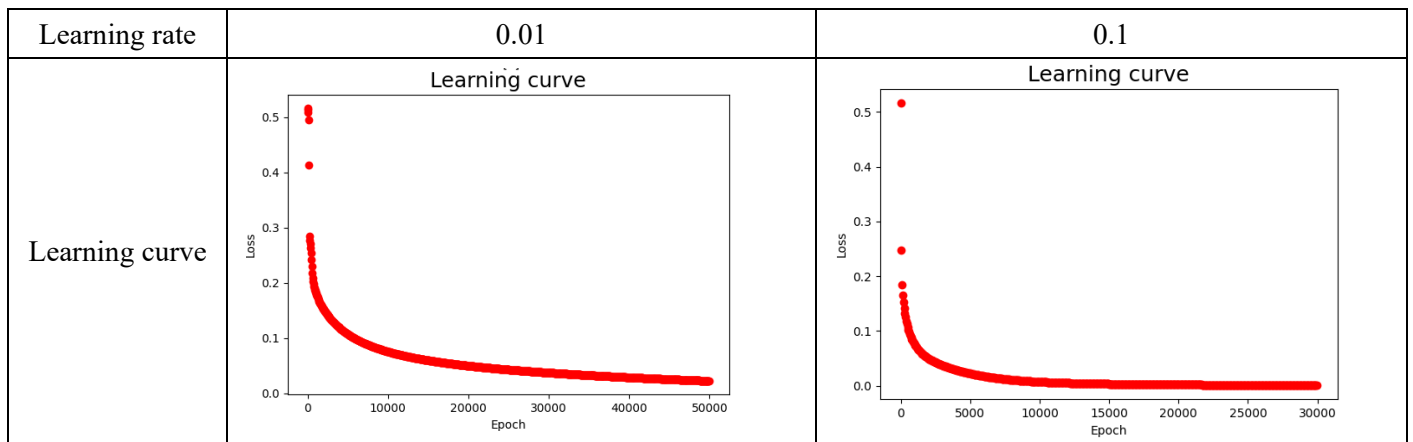
1. 試試看不同的 learning rate 對 Loss curve 的影響

Linear model: epoch 為 15000、Data amount 為 1000、隱藏層 neuron 數量為 4



在線性的模型當中我們可以發現無論 learning rate 大小，整個模型都有機會收斂到一個相對小的 loss，不過兩者在收斂的速度上就有了不小的差距。

XOR model: epoch 為 30000、隱藏層 neuron 數量為 100

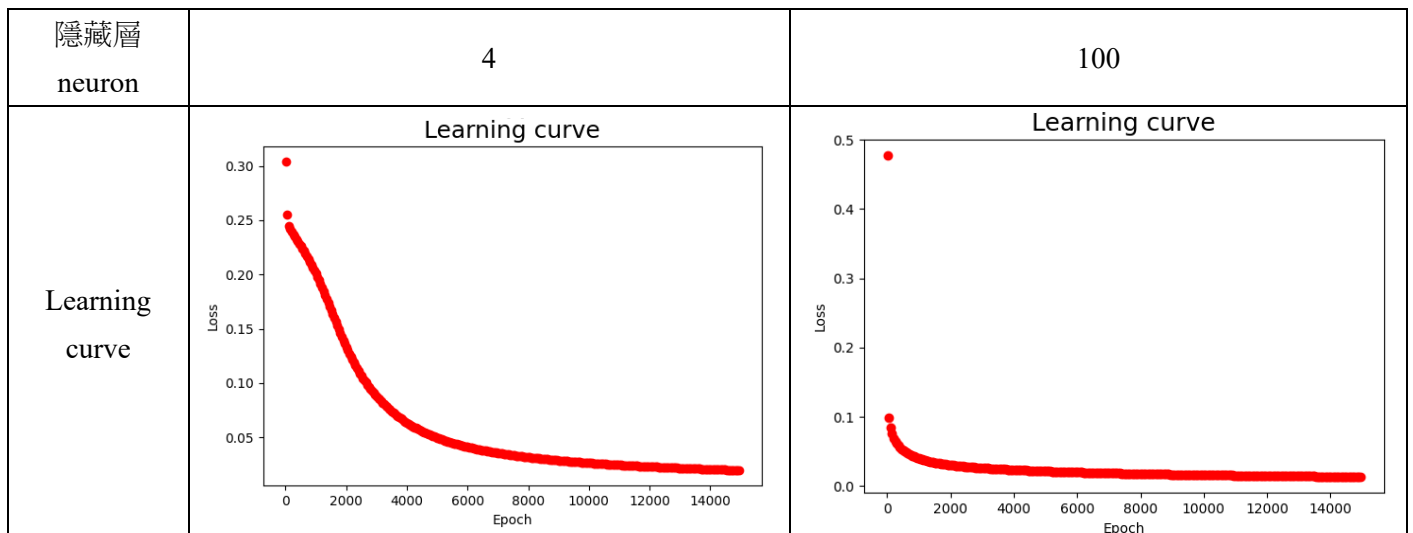


在 XOR 這個較複雜的非線性模型當中，我們可以看見在 lr 較小的時候，同樣出現了需要比較長的時間才收斂的結果，當我們將 lr 調大之後，可以看見收斂的速度明顯快了很多

Learning rate	0.01	0.1
Accuracy	1.0000	1.0000

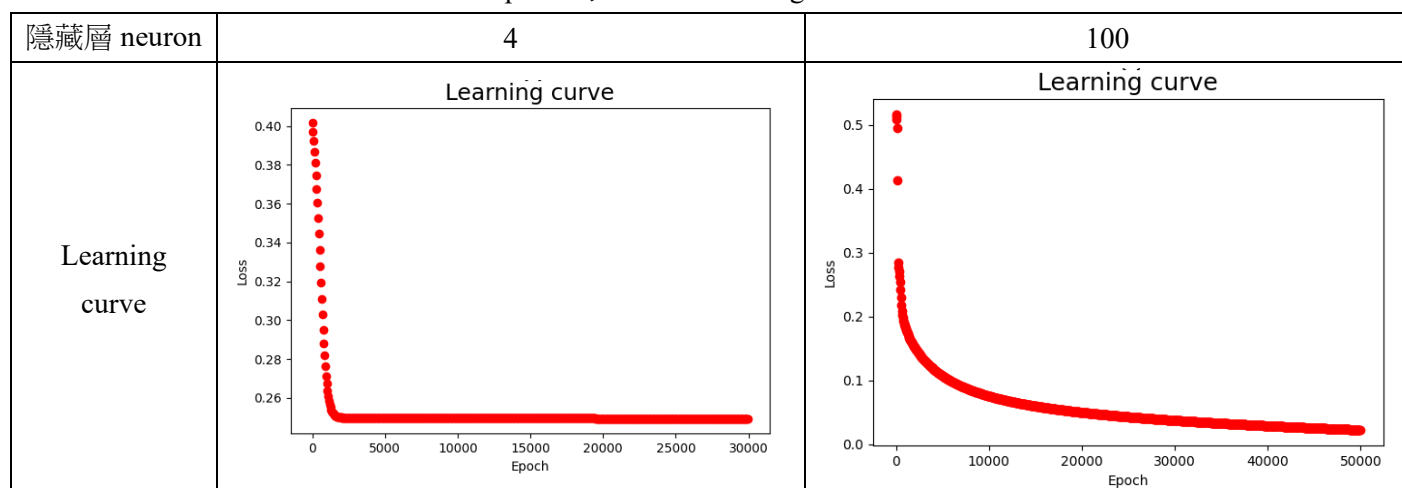
2. 在 hidden layers 當中使用不同的 neuron amount

Linear model: epoch 為 15000、Data amount 為 1000、learning rate = 0.05



從圖中可以看見當隱藏層 neuron 數量上升，我們的模型用更快的速度收斂了，不過這是指 epoch 上的速度，實際上在訓練過程當中所需要的時間更長了，而在準確度方面，兩個都很好的到達了 $accuracy = 1.000$ 的情況

XOR model: model: epoch 為 30000、learning rate = 0.01

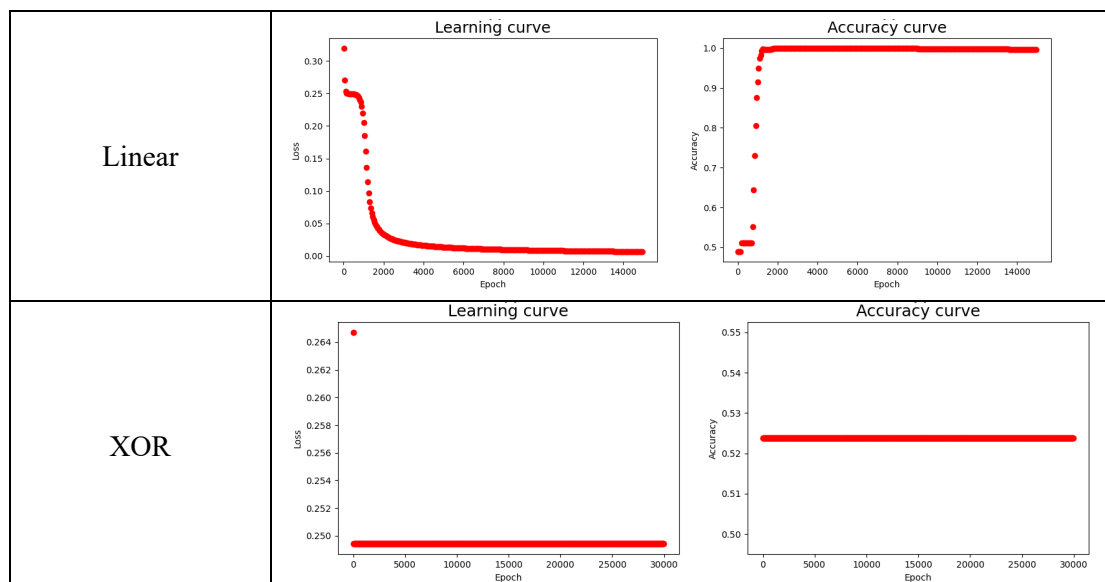


在 XOR 這個非線性的模型當中，隱藏層中的神經元數量就有很顯著的影響了，當我們使用了數量較大的神經元做隱藏層後，可以看見 loss 有效的降到了較低的數值，推測是因為在非線性模型當中，需要較高維度的模型才能有效的訓練，而從最後的 accuracy 當中也可以看見數量為 100 的模型表現優於數量為 4 的模型。

隱藏層 neuron	4	100
Accuracy	0.5238	1.000

3. 試著不使用激勵函數

為了要單純找出激勵函數對我們模型的影響，所以這次我採用前面提到的參數，這樣就可以單純的觀察激勵函數究竟會帶來什麼樣的影響



從上面的圖當中我們可以看見，linear data 的部分即使少了激勵函數還是可以正確地被判斷出來，而到了 XOR data 的部分，因為少了 non-linear 的激勵函數之後，模型就少了 non-linear classifier 的能力，所以我們的 accuracy 就變得慘澹了起來。

5. Extra

1. 試著使用不同的 activation functions

使用 XOR 模型來做實驗

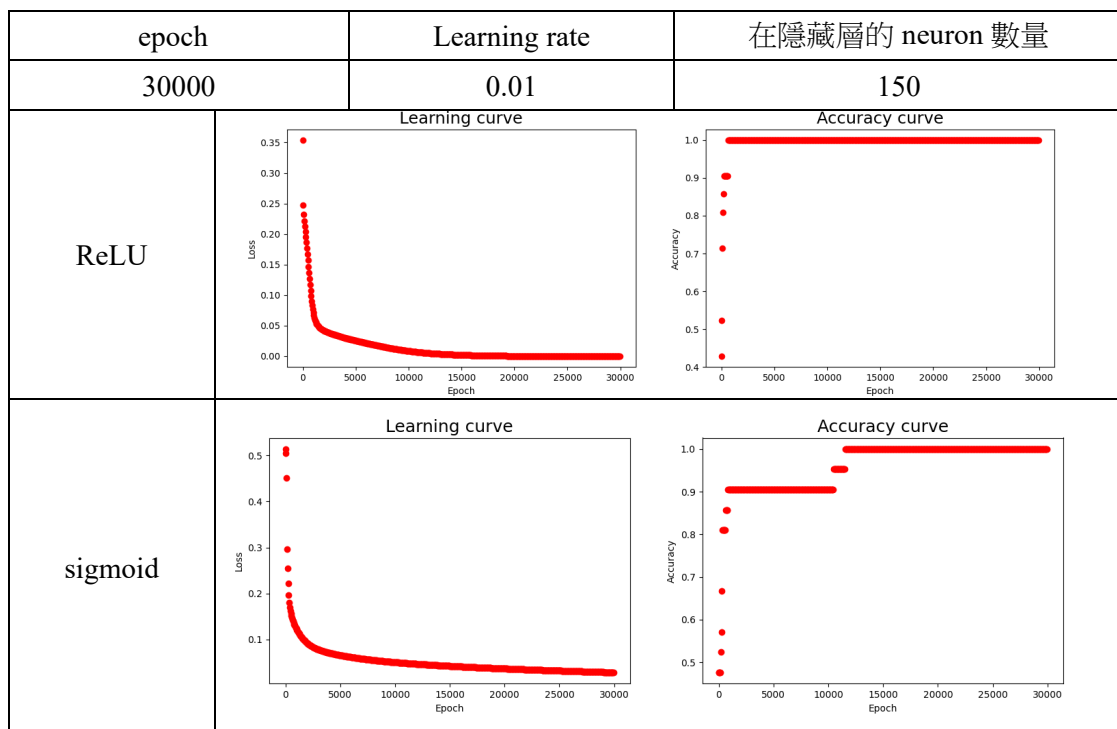
epoch	Learning rate	在隱藏層的 neuron 數量
30000	0.01	100

A. ReLU

$$f(x) = \max(0, x), f'(x) = \begin{cases} 1, & (x \geq 0) \\ 0, & (x < 0) \end{cases}$$

使用 ReLU 這個函數的好處是計算、收斂相當快，不過在使用上有一些需要注意的點，例如因為 ReLU 函數在 0 的地方不可微分，所以我們需要自己設定 $x = 0$ 的微分值，此外在權重的部分不能完全隨機，否則當 output 剛好都是 0 或者負值時，這個函數就不會被激發，這樣的情況也被稱為 Dead ReLU 問題。

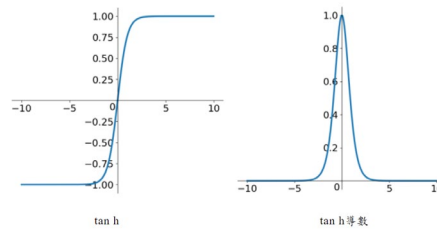
而因為換了激發函數，所以在參數上面我沒辦法直接使用以往的參數，而是重新設計了一組



從兩者的圖表可以看見 ReLU 的模型收斂的比較快，也收斂到比較低的 cost，在訓練上也可以感覺到使用 ReLU 模型的訓練速度較快。

B. Tanh:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, f'(x) = 1 - f(x)^2$$



從上圖當中我們可以看見 tanh 函數解決了 sigmoid 函數中 輸入 $x = 0$ 輸出卻不為 0 的情況，而這次實驗我再次換了一組參數來進行比較

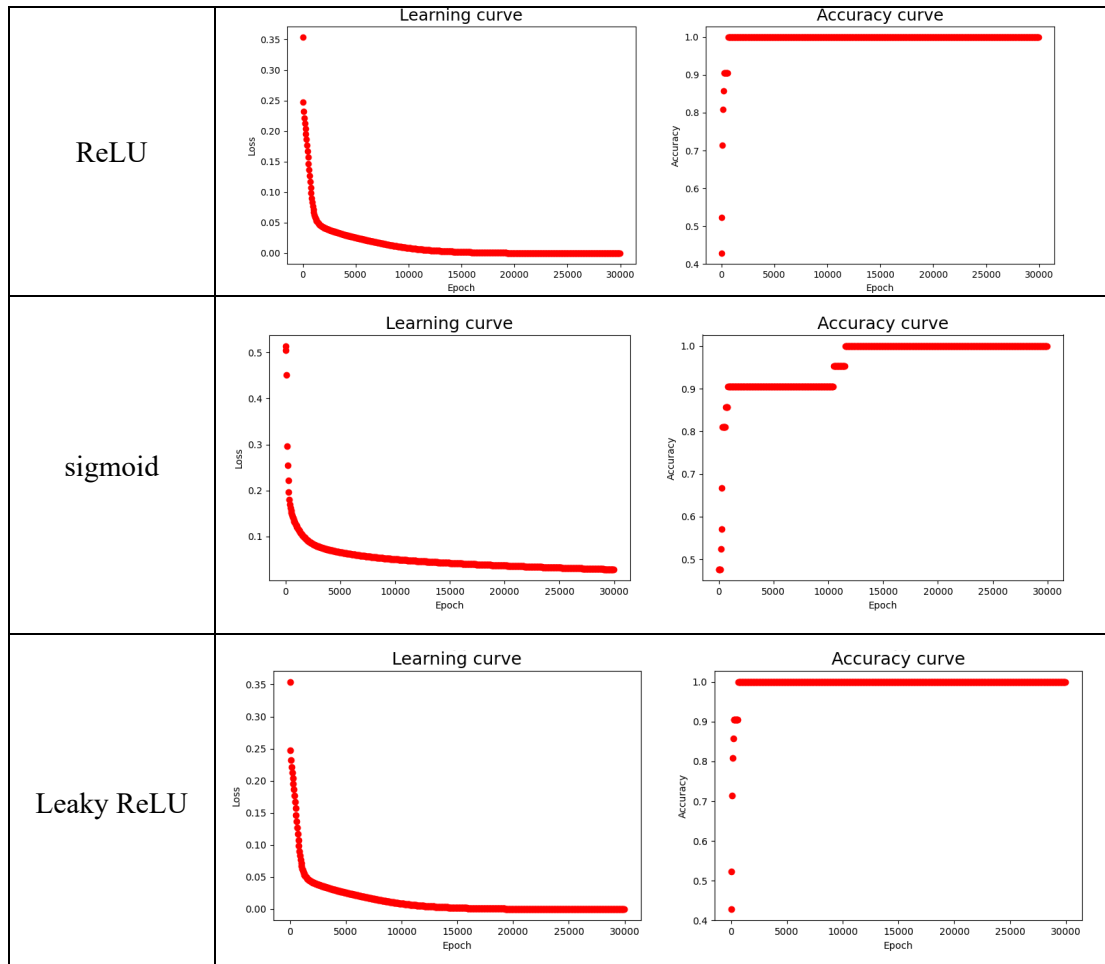
epoch	Learning rate	在隱藏層的 neuron 數量
30000	0.015	100
tanh	<div> <div>Learning curve</div> </div> <div> <div>Accuracy curve</div> </div>	
sigmoid	<div> <div>Learning curve</div> </div> <div> <div>Accuracy curve</div> </div>	

在導數的峰值變高、解決了在函數在 $\text{input} = 0$ 時函數值不為零的兩個問題後，可以看見我們的 loss 收斂的比較快、也收斂到了比較低的點

C. Leaky ReLU

$$f(x) = \max(0, x) + \min(0.001x, 0), f'(x) = \begin{cases} 1, & (x \geq 0) \\ 0.001, & (x < 0) \end{cases}$$

Leaky ReLU 解決了 Dead ReLU 的問題，而這邊我使用與 ReLU 同樣的參數來與 ReLU、sigmoid 進行比較



在這次實驗當中沒辦法很有效地看出 leaky ReLU 與 ReLU 的差別，我推測是因為權重的初始值問題，也因為我沒有碰到 dead ReLU 問題，所以在這邊看不太出差別

2. implement optimizer

1. SGD: SGD 其實就是單純的使用梯度作為修正權重用的數值，也就是我們一般在 lab 中實作的方法
2. Momentum: 在 momentum 當中我們引入了一個新的超參數 β 過去我們在更新權重的時候使用的是

$$W = W - lr * gt$$

$$B = B - lr * gt \quad \text{gt 代表 gradient}$$

而如今我們引入了 β 之後 將我們更新權重的方式改為

$$W = W - lr * V_{dw}$$

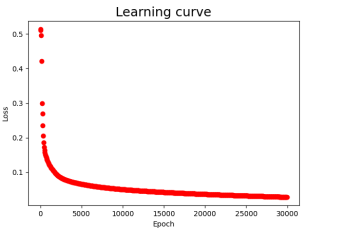
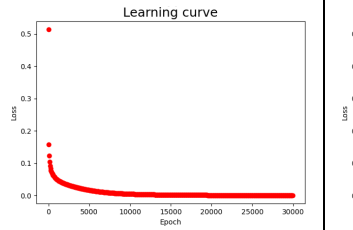
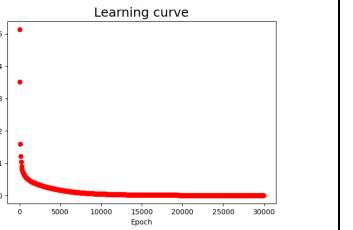
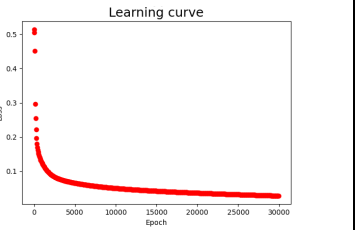
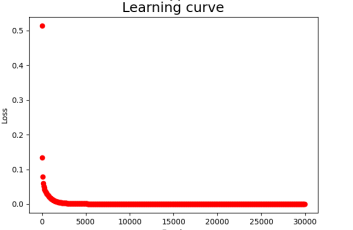
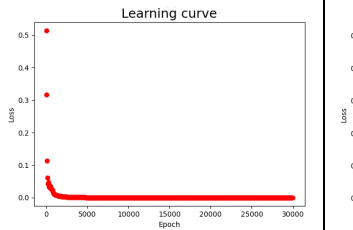
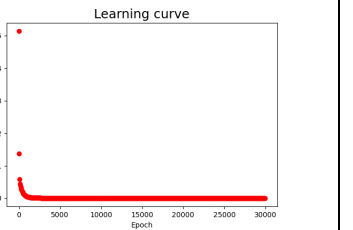
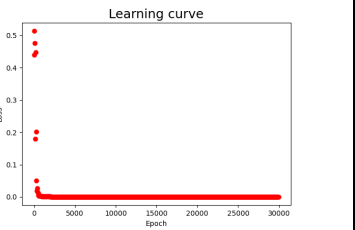
$$B = B - lr * V_{db}$$

$$V_{dw} = \beta * previous V_{dw} + (1 - \beta) * gt$$

$$V_{db} = \beta * previous V_{db} + (1 - \beta) * gt$$

在這一個 optimizer 當中因為我們將權重的變化量套用了 exponential moving average，所以我們可以將以往權重變化時產生的震盪給縮小(可以將 β 這一個超參數視為是運動中的摩擦力，而如此一來我們就可以套用一個更大的 learning rate，來達到更好的效果

XOR model， β :0.95，epoch: 30000，隱藏層神經元: 150，使用 sigmoid 函數激發

Learning rate	0.01		0.1	
Learning curve				
Learning rate	0.5		1	
Learning curve				

左圖為套用了 momentum optimizer 之後，右圖則為沒有套用 optimizer

從圖中我們可以看到，雖然從收斂速度上兩者沒有太大的差別，但正如上面所提到的 β 這一個概念很像摩擦力的超參數，在模型訓練的過程當中降低了震動，讓學習的曲線變得平滑化。

3. Implement convolutional layers

這次無論是 linear 還是 XOR，都是(X1,X2) 這樣的 input，因此 convolutional layer 的部分我採用的是 1D 的形式。

首先，我先將本來 shape = (1,2) 的 input data padding 成(1,4)的 data，如此以來我才有移動的空間。而如此一來，我們的 Input data 的形式變成(0,X1,X2,0)。

接著我設計的 kernel size 是 3。Kernel 的資料是(w₁,w₂,w₃)。

結合上述兩個資訊，這次我新增的 convolutional layer 有以下資訊

1. Input dim = 4

2. Kernel dim = 3

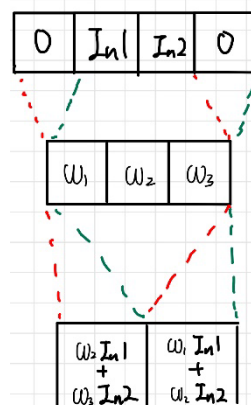
3. Output dim = 2

右圖由上至下分別為:

Input

Kernel

Output



而這次我沒有使用到 max pooling。

而 forward output 的部分就如右上圖所示，所以我們只需要煩惱 backward propagation，而因為這次設計我把 convolutional layer 放在最前面，而且只有一層，所

以我們只要套討到 $\frac{\partial J}{\partial w_1}$ 、 $\frac{\partial J}{\partial w_2}$ 、 $\frac{\partial J}{\partial w_3}$ 三者即可

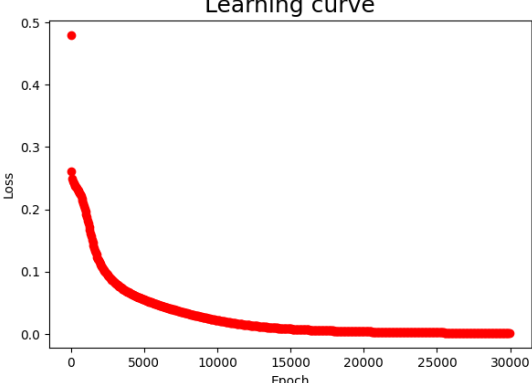
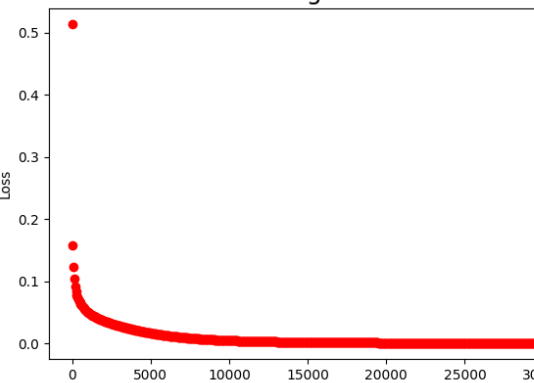
$$\frac{\partial L}{\partial h} \frac{\partial h}{\partial w} = \begin{bmatrix} \frac{\partial L}{\partial h_1} & \frac{\partial L}{\partial h_2} \end{bmatrix}$$

$$\begin{bmatrix} \frac{\partial h}{\partial w_1} & \frac{\partial h_1}{\partial w_1} & \frac{\partial h_2}{\partial w_1} \\ \frac{\partial h}{\partial w_2} & \frac{\partial h_1}{\partial w_2} & \frac{\partial h_2}{\partial w_2} \\ \frac{\partial h}{\partial w_3} & \frac{\partial h_1}{\partial w_3} & \frac{\partial h_2}{\partial w_3} \end{bmatrix} \begin{bmatrix} \frac{\partial \text{loss}}{\partial h_1} & \frac{\partial \text{loss}}{\partial h_2} \end{bmatrix}_{(1 \times 2)} \times \begin{bmatrix} \frac{\partial h_1}{\partial w_1} & \frac{\partial h_1}{\partial w_2} & \frac{\partial h_1}{\partial w_3} \\ \frac{\partial h_2}{\partial w_1} & \frac{\partial h_2}{\partial w_2} & \frac{\partial h_2}{\partial w_3} \end{bmatrix}_{(2 \times 3)}$$

$$= \begin{bmatrix} \quad \quad \quad \end{bmatrix}_{1 \times 3}$$

得到了每次修正的權重如何計算之後，把他串接到本來的 model 上面即可
接著就是看看效果，我使用 XOR 模型來探討這次的結果

使用的參數:

epoch	Learning rate	在隱藏層的 neuron 數量
30000	0.1	150
有無 conv layer	有	無
Learning curve		

在這次的模型當中神經網路並沒有帶來多大的效益，我覺得有個根本的原因是這次的模型並沒有複雜到需要動用捲積層，而在收斂上的速度也比較慢，這部分我覺得是因為我使用了 padding zero 的技巧，對於這次的資料而言沒有多大的幫助，反而是增加了資料的複雜度。