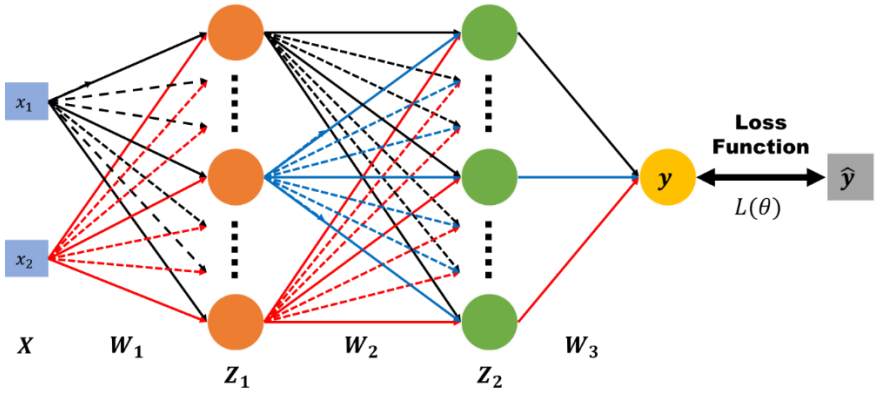
Lab2 : back-propagation

310605005 王映勻

1. Introduction



這次 lab 我們需要實作出一個包含兩層 hidden layers 的神經網路架構，對兩種二元分類問題進行測試，將分類結果及學習曲線等繪製成圖表，比較調整不同參數及激活函數(Activation functions)的結果並探討其原因。

實作流程：

[1] Network Initialization

[2] Forward propagation

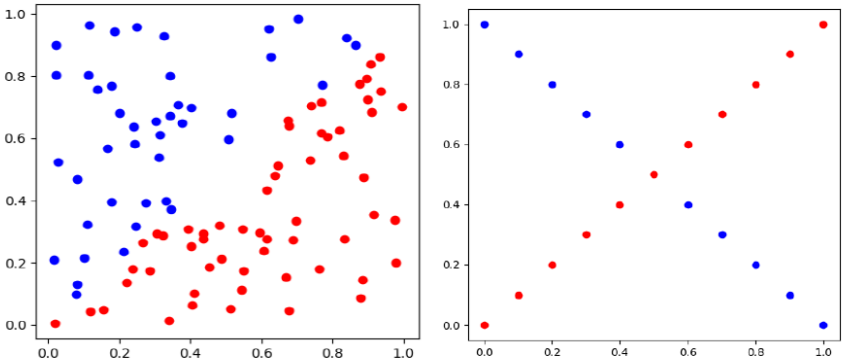
[3] Backpropagation

[4] Weight update

Repeat until the performance of the network is satisfactory

[5] Predictition

Input data：



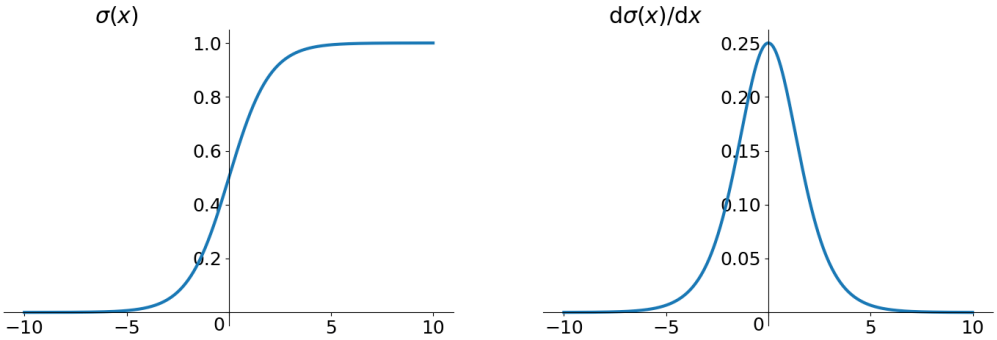
(1) Linear　　(2) XOR

2. Experiment setups

A. Sigmoid functions

Sigmoid functions 是最常見的一種激活函數，激活函數的目的是為了讓神經網路能實現非線性輸出，使其能學習更複雜的問題。

$$Sigmoid：f(x)=\frac{1}{1+e^{-x}}\quad，\quad f'(x)=f(x)*(1-f(x))$$



Sigmoid functions 把輸入的連續實值變換為 0 和 1 之間的輸出，是便於求導數之平滑函數。但這種指數函數在計算上較耗損資源，且因其導數在極值上會趨近於 0，在 Backpropagation 與各層導數的數值進行乘積時，結果就會趨近於 0，容易造成梯度消失 (gradient vanishing) 的問題。

B. Neural network

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 的神經網路模型由一個輸入層、兩個隱藏層，以及一個輸出層所組成。將輸入 x 乘上權重 w 後會得到一個 score ($s = \sum w_i x_i$)，將這個 score 輸入到激活函數(本次設定為 sigmoid funstions)計算後，判斷其是否大於設定的閾值以向後傳遞，此即為一個神經元的運算。將多組神經元做線性組合最後透過 loss function 來衡量出最後的答案。有了這個模型架構後，即可以透過 gradient descent 的方式來對參數做最佳化，此方法即為後面所提到的 backpropagation

C. Backpropagation

Backpropagation 的概念是將誤差值往回傳遞資訊，透過連鎖律(chain rule)可計算出損失函數對每一個節點的梯度，再進一步算出對每個權重的梯度，使模型可以利用這些資訊進行梯度下降法來更新權重以最小化誤差。

以這次的雙層神經網路模型為例，為了讓模型能自動地從資料中學到神經網路的權重，我們需要得到 loss 對權重 w 的偏微分，也就等於把每個參數的 loss 對特定參數 w 的微分加總：

$$Gradient\ of\ weight:\ \hat{w} = w - \eta \frac{\partial L(\theta)}{\partial w} \quad (\eta: learning\ rate)$$

$$Loss\ function:\ L(\theta)=\sum_{n=1}^{N} l_n(\theta) \rightarrow \frac{\partial L(\theta)}{\partial w}=\sum_{n=1}^{N}\frac{\partial l_n(\theta)}{\partial w}$$

$$l_n(\theta): error\ function\ of\ y_n\ \&\ \hat{y}_n\ (In\ this\ case, N=2)$$

利用 chain rule 得出 $\frac{\partial l(\theta)}{\partial w}=\frac{\partial z}{\partial w}*\frac{\partial l(\theta)}{\partial z}$ $(z: a\ neuron\ of\ hidden\ layers)$

接著就可以分為以下兩部份推導：

- Forward pass → $\frac{\partial z}{\partial w}$

$$\because\ z=x_1w_{n1}+x_2w_{n2} \qquad \therefore\ \frac{\partial z}{\partial w_{n1}}=x_1\ ,\ \frac{\partial z}{\partial wn_2}=x_2$$

因此可以得出一個結論，$\frac{\partial z}{\partial w}$ 即為該神經元的輸入 x。

- Backward pass → $\frac{\partial l(\theta)}{\partial z}$

$$\frac{\partial l(\theta)}{\partial z}=\frac{\partial a}{\partial z}\frac{\partial l(\theta)}{\partial a}\ (a: output\ of\ each\ hidden\ layer)$$

假設每層 hidden layer 只有兩個神經元 $z'\ \&\ z''$，且每層的輸出 z 都需再經過激活函數 σ 的運算得到 $a=\sigma(z)$：

$$\frac{\partial a}{\partial z}=\sigma'(z)\ ,\quad \frac{\partial l(\theta)}{\partial a}=\frac{\partial z'}{\partial a}\frac{\partial l(\theta)}{\partial z'}+\frac{\partial z''}{\partial a}\frac{\partial l(\theta)}{\partial z''}$$

$$\because \frac{\partial z}{\partial a}=w \quad \therefore \frac{\partial l(\theta)}{\partial z}=\sigma'(z)*[w'\frac{\partial l(\theta)}{\partial z'}+w''\frac{\partial l(\theta)}{\partial z''}]$$

由上式得出，目前的未知數僅剩下 $\frac{\partial l(\theta)}{\partial z'}$ & $\frac{\partial l(\theta)}{\partial z''}$ 。

若是最後一層 output layer 的話 →

$$\frac{\partial l(\theta)}{\partial z'}=\frac{\partial y_1}{\partial z'}\frac{\partial l(\theta)}{\partial y_1}\ ,\quad \frac{\partial l(\theta)}{\partial z''}=\frac{\partial y_2}{\partial z''}\frac{\partial l(\theta)}{\partial y_2}$$

(最後的輸出 y_1, y_2 已經透過正向運算得到，且 $\frac{\partial l(\theta)}{\partial y_1}, \frac{\partial l(\theta)}{\partial y_2}$ 也可利用 loss function 來決定)

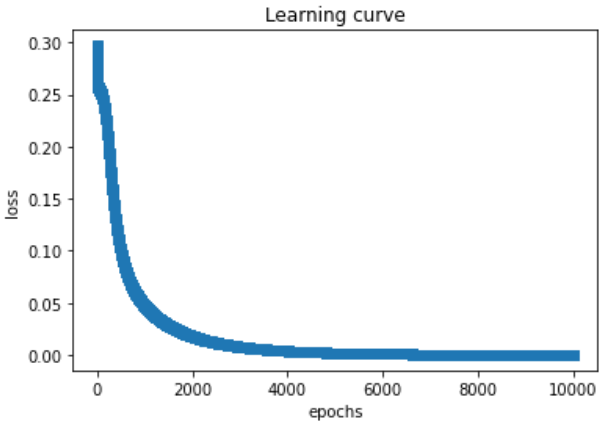
若不是 output layer 的話 →

就得一路往後推到 output layer 才能得到確切的值，再從結果一層一層回推所有的 $\frac{\partial l(\theta)}{\partial z}$

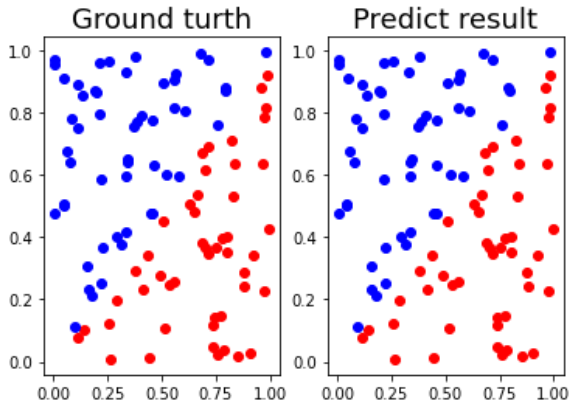
3. Results of your testing

(1) Linear (learning rate = 0.01, n_epoch = 10000)

```
epoch    0    loss : 0.29704
epoch  500    loss : 0.10009
epoch 1000    loss : 0.04742
epoch 1500    loss : 0.02833
epoch 2000    loss : 0.01792
epoch 2500    loss : 0.01175
epoch 3000    loss : 0.00793
epoch 3500    loss : 0.00548
epoch 4000    loss : 0.00387
epoch 4500    loss : 0.00278
epoch 5000    loss : 0.00203
epoch 5500    loss : 0.00151
epoch 6000    loss : 0.00113
epoch 6500    loss : 0.00086
epoch 7000    loss : 0.00067
epoch 7500    loss : 0.00052
epoch 8000    loss : 0.00041
epoch 8500    loss : 0.00033
epoch 9000    loss : 0.00027
epoch 9500    loss : 0.00022
```

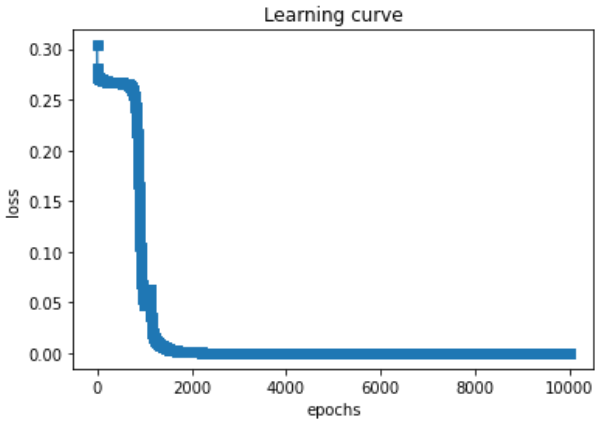



```
0.99918898   label = 1  predict = 1
0.08553790   label = 0  predict = 0
0.00051500   label = 0  predict = 0
0.99904851   label = 1  predict = 1
0.00031254   label = 0  predict = 0
0.87299754   label = 1  predict = 1
0.99880661   label = 1  predict = 1
0.00580453   label = 0  predict = 0
0.76166223   label = 1  predict = 1
0.67883747   label = 1  predict = 1
0.00069167   label = 0  predict = 0
0.00034529   label = 0  predict = 0
0.99901158   label = 1  predict = 1
0.99902390   label = 1  predict = 1
0.00026575   label = 0  predict = 0
0.99929975   label = 1  predict = 1
0.01482909   label = 0  predict = 0

Accuracy :  1.0
```

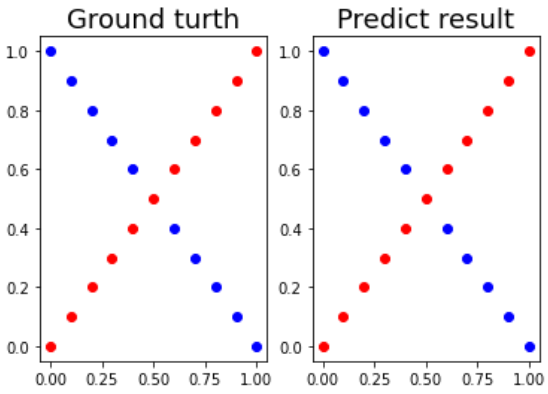


(2) XOR (learning rate = 0.3, n_epoch = 10000)

```
epoch    0    loss : 0.30356
epoch  500    loss : 0.26656
epoch 1000    loss : 0.04911
epoch 1500    loss : 0.00430
epoch 2000    loss : 0.00100
epoch 2500    loss : 0.00047
epoch 3000    loss : 0.00030
epoch 3500    loss : 0.00021
epoch 4000    loss : 0.00016
epoch 4500    loss : 0.00013
epoch 5000    loss : 0.00011
epoch 5500    loss : 0.00009
epoch 6000    loss : 0.00008
epoch 6500    loss : 0.00007
epoch 7000    loss : 0.00006
epoch 7500    loss : 0.00006
epoch 8000    loss : 0.00005
epoch 8500    loss : 0.00005
epoch 9000    loss : 0.00004
epoch 9500    loss : 0.00004
```



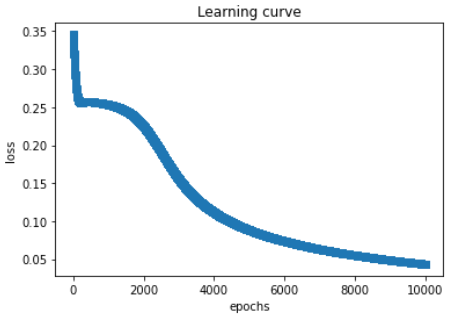
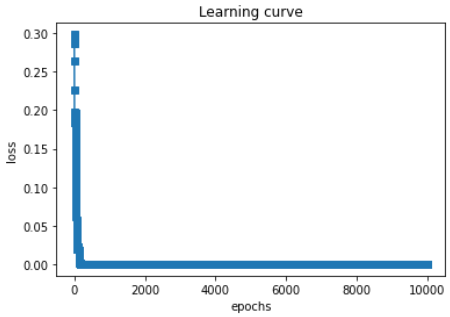

```
0.01242833   label = 0  predict = 0
0.99785588   label = 1  predict = 1
0.01240263   label = 0  predict = 0
0.99757498   label = 1  predict = 1
0.01237699   label = 0  predict = 0
0.96702376   label = 1  predict = 1
0.01235142   label = 0  predict = 0
0.01232593   label = 0  predict = 0
0.97155993   label = 1  predict = 1
0.01230050   label = 0  predict = 0
0.99863877   label = 1  predict = 1
0.01227515   label = 0  predict = 0
0.99880293   label = 1  predict = 1
0.01224986   label = 0  predict = 0
0.99880791   label = 1  predict = 1
0.01222464   label = 0  predict = 0
0.99880806   label = 1  predict = 1

Accuracy :  1.0
```



4. Discussion

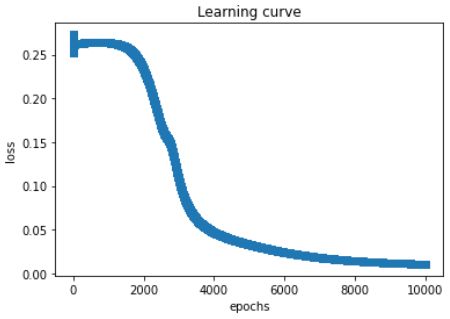
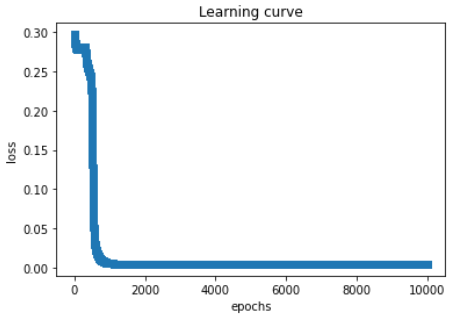
A. Try different learning rates (n_epoch = 10000, l : learning rate)

(1) Linear

l=0.001	l=0.8
	

由於輸入為簡單的線性模型，因此不論 learning rate 數值高低都有機會收斂，而從上圖也可以看出，當 learning rate 較低時，收斂速度及程度都明顯下降，反之訓練效果較好。

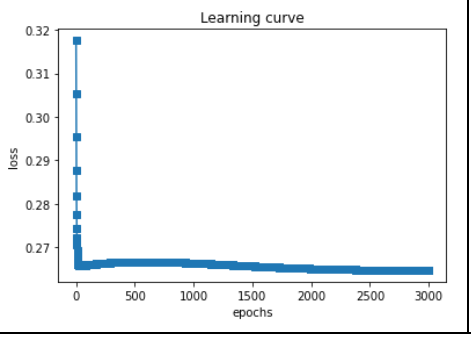
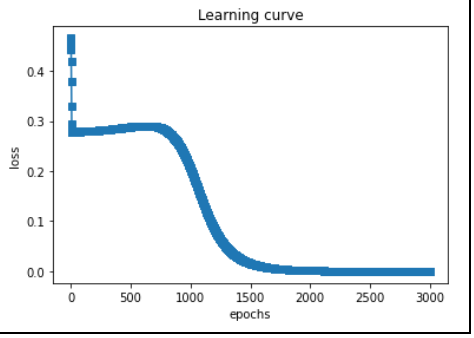
(2) XOR

l=0.05	l=0.8
	

XOR 模型則較為複雜，訓練時嘗試了很多組 learning rate 才訓練到全部正確。而從上圖同樣可看出當 learning rate 較低時，收斂速度明顯下降，而當 learning rate 較高時，雖然從上圖看不出來，但仔細觀察數值，可發現其最終收斂的 loss 值較高，推測可能是因為 learning rate 太高，導致收斂時不斷振盪，無法收斂至更小的 loss。

B. Try different numbers of hidden units

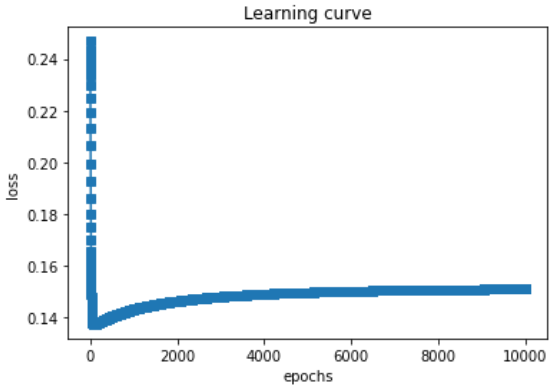
以 XOR 模型為例 (learning rate = 0.1, n_epoch = 3000)：

n_hidden=2	n_hidden=10
	

XOR 模型本身就非線性模型，因此若在隱藏層利用較多神經元訓練效果應該會較好，從上圖比較即可看出，當一層使用 2 個神經元時，雖然能收斂到一個數值但效果不好，而當提高神經元的數量後，模型則有機會收斂至較低的 loss。

C. Try without activation functions

若不使用 activation function 的話，整個神經網路即只是一個線性模型，難以訓練去解決各種非線性問題。以下圖為例，可看出模型是無法收斂的：



5. Extra

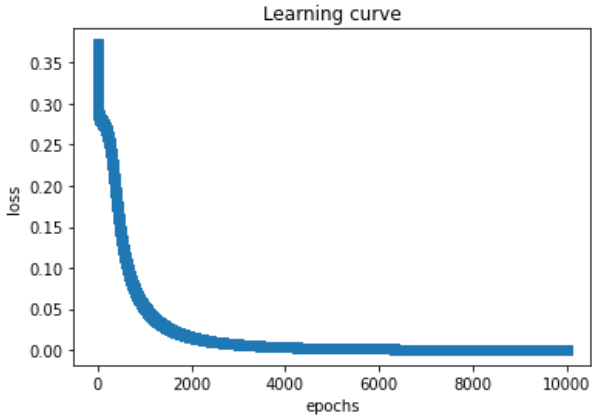
A. Implement different activation functions.

以線性模型為例 (learning rate = 0.01, n_epoch = 10000)：

✓ Sigmoid

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

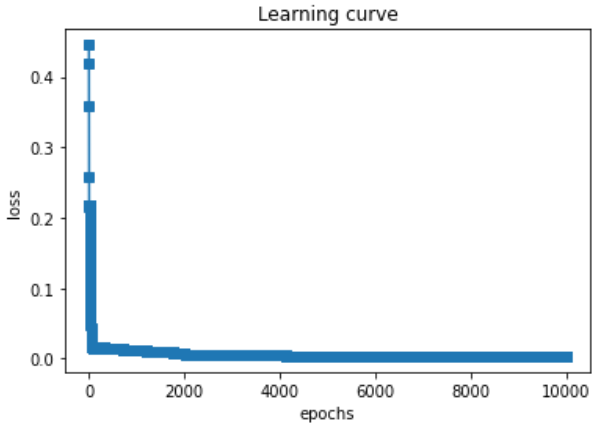
```
epoch    0    loss : 0.37304
epoch  500    loss : 0.14101
epoch 1000    loss : 0.05189
epoch 1500    loss : 0.02640
epoch 2000    loss : 0.01545
epoch 2500    loss : 0.00985
epoch 3000    loss : 0.00665
epoch 3500    loss : 0.00469
epoch 4000    loss : 0.00342
epoch 4500    loss : 0.00257
epoch 5000    loss : 0.00197
epoch 5500    loss : 0.00154
epoch 6000    loss : 0.00122
epoch 6500    loss : 0.00098
epoch 7000    loss : 0.00080
epoch 7500    loss : 0.00066
epoch 8000    loss : 0.00055
epoch 8500    loss : 0.00046
epoch 9000    loss : 0.00039
epoch 9500    loss : 0.00033
```



✓ Tanh

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

```
epoch    0    loss : 0.44571
epoch  500    loss : 0.01445
epoch 1000    loss : 0.01158
epoch 1500    loss : 0.01012
epoch 2000    loss : 0.00639
epoch 2500    loss : 0.00521
epoch 3000    loss : 0.00476
epoch 3500    loss : 0.00442
epoch 4000    loss : 0.00415
epoch 4500    loss : 0.00394
epoch 5000    loss : 0.00378
epoch 5500    loss : 0.00366
epoch 6000    loss : 0.00356
epoch 6500    loss : 0.00349
epoch 7000    loss : 0.00343
epoch 7500    loss : 0.00337
epoch 8000    loss : 0.00332
epoch 8500    loss : 0.00328
epoch 9000    loss : 0.00324
epoch 9500    loss : 0.00320
```



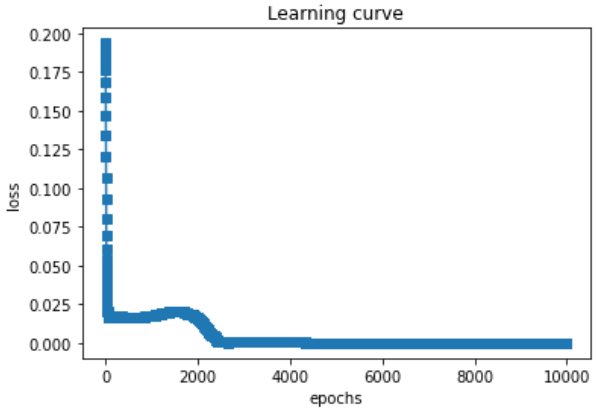
Tanh 的輸出在 -1~1 之間，均值為 0，理論上會較 Sigmoid

更方便下一層網路的學習

✓ ReLU

$$f(x) = \max(0, x) \ , \ f'(x) = 1\ (x \geq 0)\ or\ 0\ (x < 0)$$

```
epoch    0    loss : 0.19241
epoch  500    loss : 0.01683
epoch 1000    loss : 0.01764
epoch 1500    loss : 0.02072
epoch 2000    loss : 0.01588
epoch 2500    loss : 0.00079
epoch 3000    loss : 0.00066
epoch 3500    loss : 0.00057
epoch 4000    loss : 0.00049
epoch 4500    loss : 0.00040
epoch 5000    loss : 0.00031
epoch 5500    loss : 0.00023
epoch 6000    loss : 0.00017
epoch 6500    loss : 0.00013
epoch 7000    loss : 0.00010
epoch 7500    loss : 0.00008
epoch 8000    loss : 0.00006
epoch 8500    loss : 0.00004
epoch 9000    loss : 0.00003
epoch 9500    loss : 0.00002
```

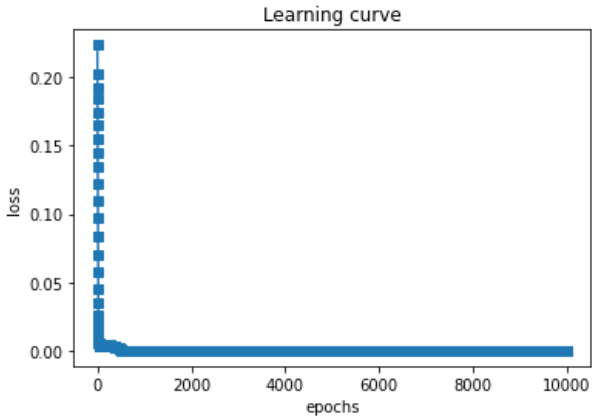


因數學運算簡單，計算速度較 tanh 和 sigmoid 快

✓ Leaky ReLU

$$f(x) = x\ (x > 0)\ or\ \alpha x (x \leq 0)\ ,\ f'(x) = 1\ (x > 0)\ or\ \alpha (x \leq 0)$$
$$(\alpha\ is\ a\ small\ value)$$

```
epoch    0    loss : 0.22376
epoch  500    loss : 0.00035
epoch 1000    loss : 0.00029
epoch 1500    loss : 0.00025
epoch 2000    loss : 0.00021
epoch 2500    loss : 0.00019
epoch 3000    loss : 0.00018
epoch 3500    loss : 0.00017
epoch 4000    loss : 0.00018
epoch 4500    loss : 0.00019
epoch 5000    loss : 0.00020
epoch 5500    loss : 0.00022
epoch 6000    loss : 0.00023
epoch 6500    loss : 0.00024
epoch 7000    loss : 0.00025
epoch 7500    loss : 0.00026
epoch 8000    loss : 0.00027
epoch 8500    loss : 0.00028
epoch 9000    loss : 0.00029
epoch 9500    loss : 0.00030
```



因 x<0 時不為 0，因此可以解決 ReLu dying 的問題