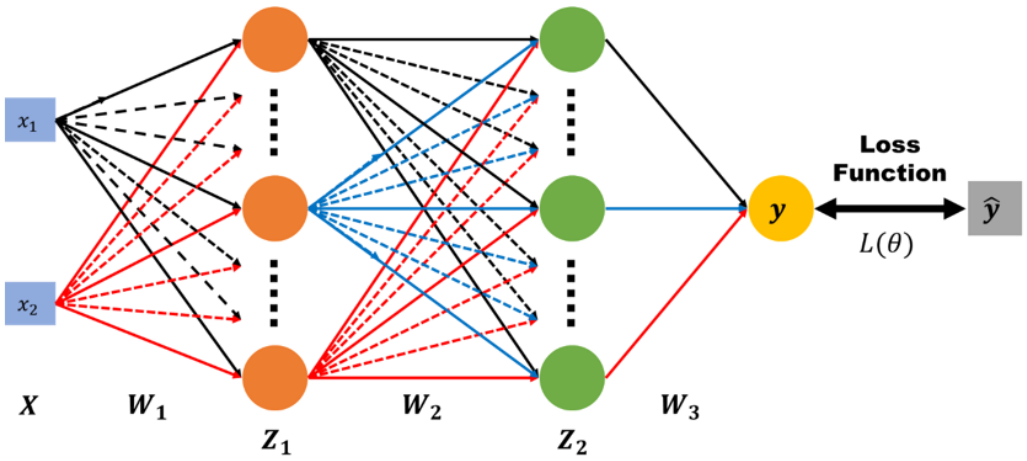
Lab1: Back-propagation

系級:智能系統　學號:312581006　姓名:張宸瑋

1. Introduction

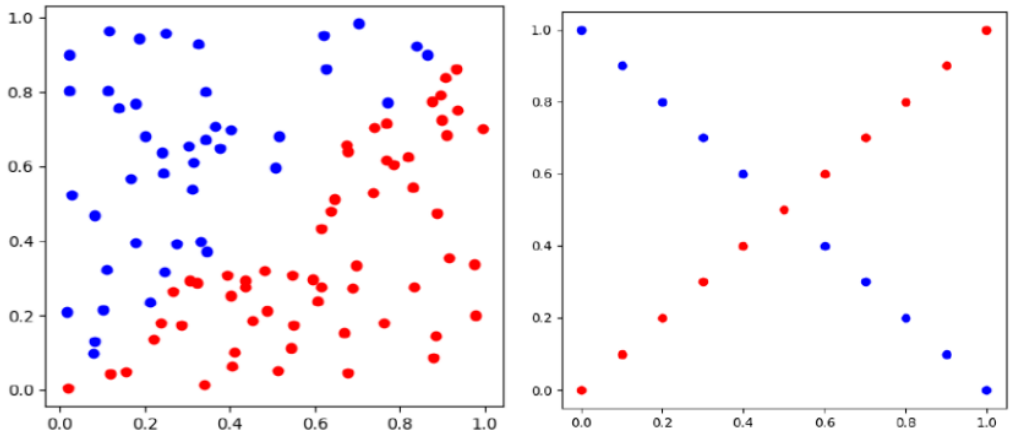


這次 lab 要實作出一個有兩層 hidden layers 的 神經網路，並且分別對兩種二元分類問題進行訓練，將分類結果、訓練過程、預測結果等數據可視化，用來探討神經網路實作中的各項細節，包含 back-propagation 如何實作、如何利用 chain-rule 來計算 gradient 以及 使用不同 optimizer 和 activation functions 會帶來怎麼樣的結果

實作流程:

A. 生成訓練用資料
B. 初始化神經網路(初始化各層權重)
C. 前向傳播（Forward Propagation）
D. 計算損失函數
E. 反向傳播（Backpropagation）
F. 參數更新
G. 重複訓練

實驗中使用的 input data:



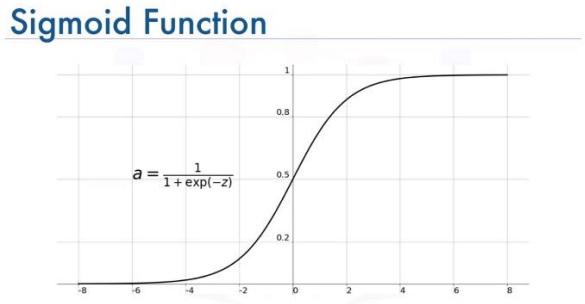
(1) Linear　　(2) XOR

圖中點的顏色分別代表 Y = 0 or 1

2. Experiment setups

A. Sigmoid functions

Sigmoid 函數（也稱為 Logistic 函數）是一個常見的激勵函數，在深度學習和機器學習中廣泛使用。它將輸入值映射到介於 0 和 1 之間的範圍，通常用於處理二元分類問題或產生概率值。



Sigmoid Function 示意圖

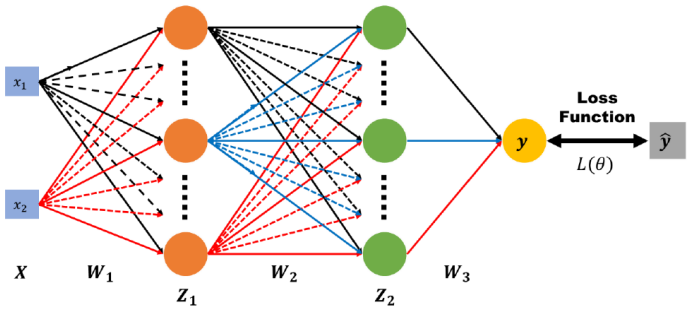
```
def sigmoid(self,x):
    return 1 / (1 + np.exp(-x))

def derivative_sigmoid(self,y):
    return y * (1 - y)
```

程式實作部分

B. Neural networks

這次的實驗的神經網路架構主要是兩層 hidden layer，並且每一層

hidden layer 都必須包含一個轉換以及激勵函數，透過兩層 hidden layer(neuron 數量為 4)的運算後，到達 output layer，透過運算出來的值，搭配我這次使用作為損失函數的 MSE，來評估預測值以及實際值的誤差。



神經網路架構

```
class NN:
    #Y = weight*X + b
    def __init__(self,input_dim, hidden_layer_dim, output_dim, activation_type, learning_rate,optimizer = "gd"):

        self.neural_network =[]
        self.neural_network.append(layers(input_dim,hidden_layer_dim,activation_type,optimizer))
        self.neural_network.append(layers(hidden_layer_dim,hidden_layer_dim,activation_type,optimizer))
        self.neural_network.append(layers(hidden_layer_dim,output_dim,"sigmoid",optimizer))

        self.learning_rate = learning_rate
        self.optimizer = optimizer
        self.epoch ,self.loss = [],[]

    def forward_propagate(self , input):···

    def backward_propagate(self,loss):···

    def update(self):···

    def train(self,epoch,inputs,labels):···

    def predict(self,inputs):···

    def show_learing_curve(self):···

    def show_result(self,x, y, pred_y):···
```

神經網路程式碼架構

C. Backpropagation

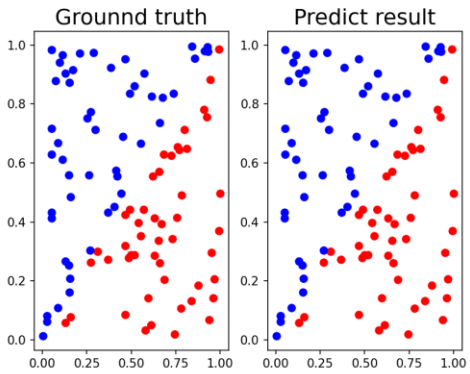
反向傳播(Backpropagation)是一種用於訓練神經網絡的優化算法，用來改善當參數量多的時候，傳統的 gradient descent 會遇到計算複雜的問題，反向傳播通過計算損失函數對於權重的梯度，逐步調整權重以最小化損失，使得預測輸出逼近真實標籤

```
def backward_propagate(self,loss):

    if(self.act_fcn_type == 'sigmoid'):
        self.gradient_matrix = loss * self.derivative_sigmoid(self.output)
    elif(self.act_fcn_type == 'relu'):
        self.gradient_matrix = loss * self.derivative_relu(self.derivate_relu_input)
    elif(self.act_fcn_type == 'leeky_relu'):
        self.gradient_matrix = loss * self.derivative_leaky_relu(self.derivate_relu_input)
    elif(self.act_fcn_type == 'tanh'):
        self.gradient_matrix = loss * self.derivative_tanh(self.derivate_tanh_input)
    elif(self.act_fcn_type == 'none'):
        self.gradient_matrix = loss

    return self.gradient_matrix.dot(self.weight_matrix.T)
```

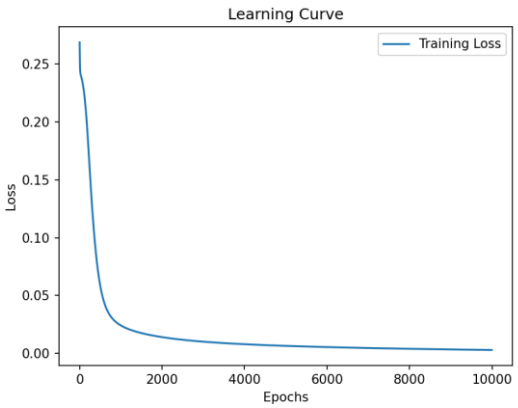
反向傳播程式碼架構

3. Result of testing

1. Linear



Predictions and Ground truth



Learning curve

```
epoch  7500 loss : 0.0018676630
epoch  7600 loss : 0.0018362923
epoch  7700 loss : 0.0018056467
epoch  7800 loss : 0.0017757052
epoch  7900 loss : 0.0017464474
epoch  8000 loss : 0.0017178540
epoch  8100 loss : 0.0016899062
epoch  8200 loss : 0.0016625859
epoch  8300 loss : 0.0016358757
epoch  8400 loss : 0.0016097590
epoch  8500 loss : 0.0015842194
epoch  8600 loss : 0.0015592414
epoch  8700 loss : 0.0015348098
epoch  8800 loss : 0.0015109101
epoch  8900 loss : 0.0014875281
epoch  9000 loss : 0.0014646503
epoch  9100 loss : 0.0014422633
epoch  9200 loss : 0.0014203544
epoch  9300 loss : 0.0013989112
epoch  9400 loss : 0.0013779217
epoch  9500 loss : 0.0013573743
epoch  9600 loss : 0.0013372576
epoch  9700 loss : 0.0013175608
epoch  9800 loss : 0.0012982731
epoch  9900 loss : 0.0012793844
epoch 10000 loss : 0.0012608847
```

Training loss

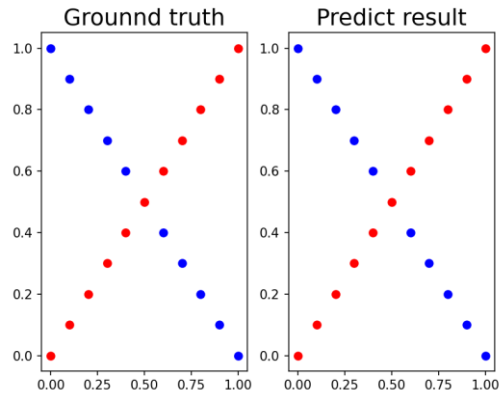
```
[1.09370143e-04]
[4.02049864e-02]
[4.72877326e-05]
[9.99727615e-01]
[7.25493719e-01]
[4.16881788e-05]
[9.92867091e-01]
[5.66738463e-05]
[9.99658786e-01]
[9.93121318e-01]
[5.83777009e-05]
[7.02061521e-05]
[1.29489866e-03]
[9.99686842e-01]
[9.99580076e-01]
[9.99696365e-01]
[4.78485488e-01]
[9.99469182e-01]
[9.99138123e-01]
[9.99728498e-01]
[2.98071764e-04]
[9.98682221e-01]
[9.98474899e-01]
[9.99735709e-01]
[9.90253700e-01]
[2.58019100e-02]
[1.06542667e-04]
[4.59070272e-05]
[9.99730425e-01]
[9.99517086e-01]
[5.24711528e-05]
[7.40412693e-01]
[9.99673960e-01]
[9.99627840e-01]
[9.56488693e-01]
[1.67637941e-04]
[7.39357301e-05]
[3.67337252e-05]
[9.67821818e-01]
[6.30894265e-05]
[9.99716514e-01]]
```

Testing prediction

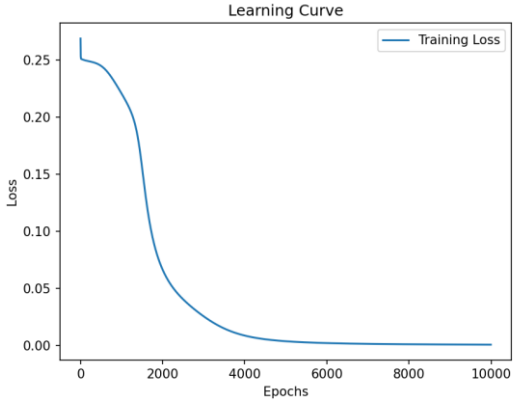
```
Iter62|   Ground truth 1|  prediction 0.999728
Iter63|   Ground truth 1|  prediction 0.725494
Iter64|   Ground truth 0|  prediction 0.000042
Iter65|   Ground truth 1|  prediction 0.992867
Iter66|   Ground truth 0|  prediction 0.000057
Iter67|   Ground truth 1|  prediction 0.999659
Iter68|   Ground truth 1|  prediction 0.993121
Iter69|   Ground truth 0|  prediction 0.000058
Iter70|   Ground truth 0|  prediction 0.000070
Iter71|   Ground truth 0|  prediction 0.001295
Iter72|   Ground truth 1|  prediction 0.999687
Iter73|   Ground truth 1|  prediction 0.999580
Iter74|   Ground truth 1|  prediction 0.999696
Iter75|   Ground truth 0|  prediction 0.478485
Iter76|   Ground truth 1|  prediction 0.999469
Iter77|   Ground truth 1|  prediction 0.999138
Iter78|   Ground truth 1|  prediction 0.999728
Iter79|   Ground truth 0|  prediction 0.000298
Iter80|   Ground truth 1|  prediction 0.998682
Iter81|   Ground truth 1|  prediction 0.998475
Iter82|   Ground truth 1|  prediction 0.999736
Iter83|   Ground truth 1|  prediction 0.990254
Iter84|   Ground truth 0|  prediction 0.025802
Iter85|   Ground truth 0|  prediction 0.000107
Iter86|   Ground truth 0|  prediction 0.000046
Iter87|   Ground truth 1|  prediction 0.999730
Iter88|   Ground truth 1|  prediction 0.999517
Iter89|   Ground truth 0|  prediction 0.000052
Iter90|   Ground truth 1|  prediction 0.740413
Iter91|   Ground truth 1|  prediction 0.999674
Iter92|   Ground truth 1|  prediction 0.999628
Iter93|   Ground truth 1|  prediction 0.956489
Iter94|   Ground truth 0|  prediction 0.000168
Iter95|   Ground truth 0|  prediction 0.000074
Iter96|   Ground truth 0|  prediction 0.000037
Iter97|   Ground truth 1|  prediction 0.967822
Iter98|   Ground truth 0|  prediction 0.000063
Iter99|   Ground truth 1|  prediction 0.999717
Accuracy:1.0
```

Accuracy of model’s prediction

2. XOR



Predictions and Ground truth



Learning curve

```
epoch  7700 loss : 0.0809161848
epoch  7800 loss : 0.0802256060
epoch  7900 loss : 0.0795585322
epoch  8000 loss : 0.0789071371
epoch  8100 loss : 0.0782627157
epoch  8200 loss : 0.0776154999
epoch  8300 loss : 0.0769547308
epoch  8400 loss : 0.0762692108
epoch  8500 loss : 0.0755483750
epoch  8600 loss : 0.0747834228
epoch  8700 loss : 0.0739676316
epoch  8800 loss : 0.0730954093
epoch  8900 loss : 0.0721607262
epoch  9000 loss : 0.0711560622
epoch  9100 loss : 0.0700725633
epoch  9200 loss : 0.0689014740
epoch  9300 loss : 0.0676364437
epoch  9400 loss : 0.0662758307
epoch  9500 loss : 0.0648238249
epoch  9600 loss : 0.0632896281
epoch  9700 loss : 0.0616850427
epoch  9800 loss : 0.0600217393
epoch  9900 loss : 0.0583094396
epoch 10000 loss : 0.0565555525
```

Training loss

```
[[0.02888666]
 [0.99772074]
 [0.02742384]
 [0.99769283]
 [0.02609427]
 [0.99751391]
 [0.02488612]
 [0.99626917]
 [0.02378868]
 [0.94492537]
 [0.02279223]
 [0.021888  ]
 [0.94503673]
 [0.02106809]
 [0.97881063]
 [0.02032534]
 [0.97997886]
 [0.01965331]
 [0.98045855]
 [0.01904619]
 [0.98077948]]
```

Testing prediction

```
Iter 0|   Ground truth 0|  prediction 0.028887
Iter 1|   Ground truth 1|  prediction 0.997721
Iter 2|   Ground truth 0|  prediction 0.027424
Iter 3|   Ground truth 1|  prediction 0.997693
Iter 4|   Ground truth 0|  prediction 0.026094
Iter 5|   Ground truth 1|  prediction 0.997514
Iter 6|   Ground truth 0|  prediction 0.024886
Iter 7|   Ground truth 1|  prediction 0.996269
Iter 8|   Ground truth 0|  prediction 0.023789
Iter 9|   Ground truth 1|  prediction 0.944925
Iter10|   Ground truth 0|  prediction 0.022792
Iter11|   Ground truth 0|  prediction 0.021888
Iter12|   Ground truth 1|  prediction 0.945037
Iter13|   Ground truth 0|  prediction 0.021068
Iter14|   Ground truth 1|  prediction 0.978811
Iter15|   Ground truth 0|  prediction 0.020325
Iter16|   Ground truth 1|  prediction 0.979979
Iter17|   Ground truth 0|  prediction 0.019653
Iter18|   Ground truth 1|  prediction 0.980459
Iter19|   Ground truth 0|  prediction 0.019046
Iter20|   Ground truth 1|  prediction 0.980779
Accuracy:1.0
```

Accuracy of model’s prediction

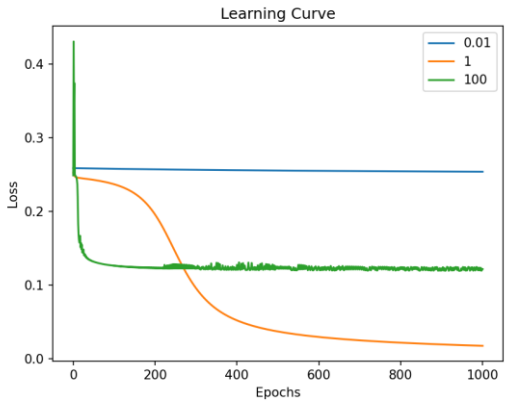
討論:

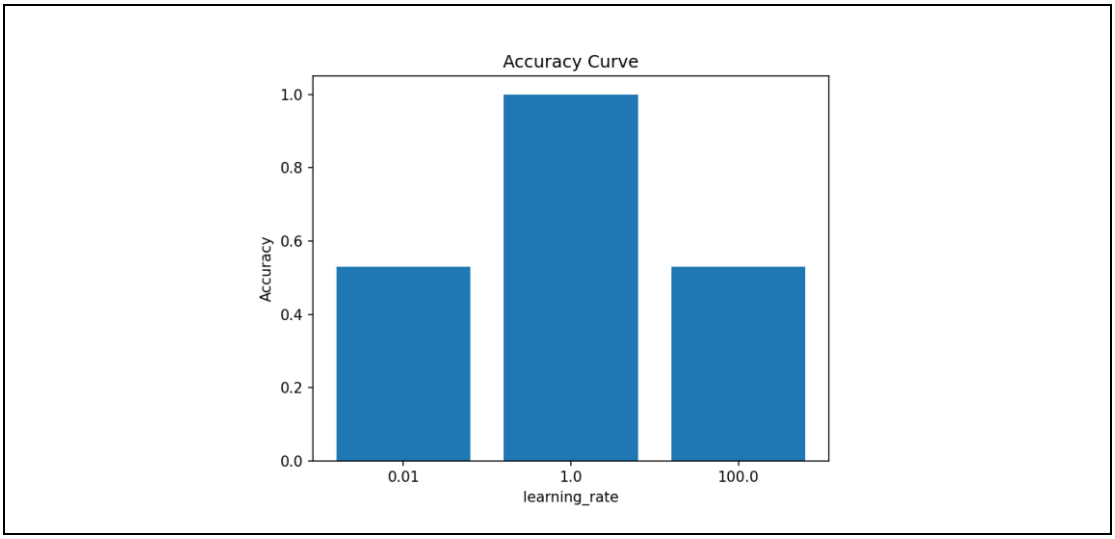
在訓練線性資料集中，loss 的收斂會來得比較快，而且在預設值的部分，線性資料集比較能預測接近 0 或 1 的答案。

4. Discussion

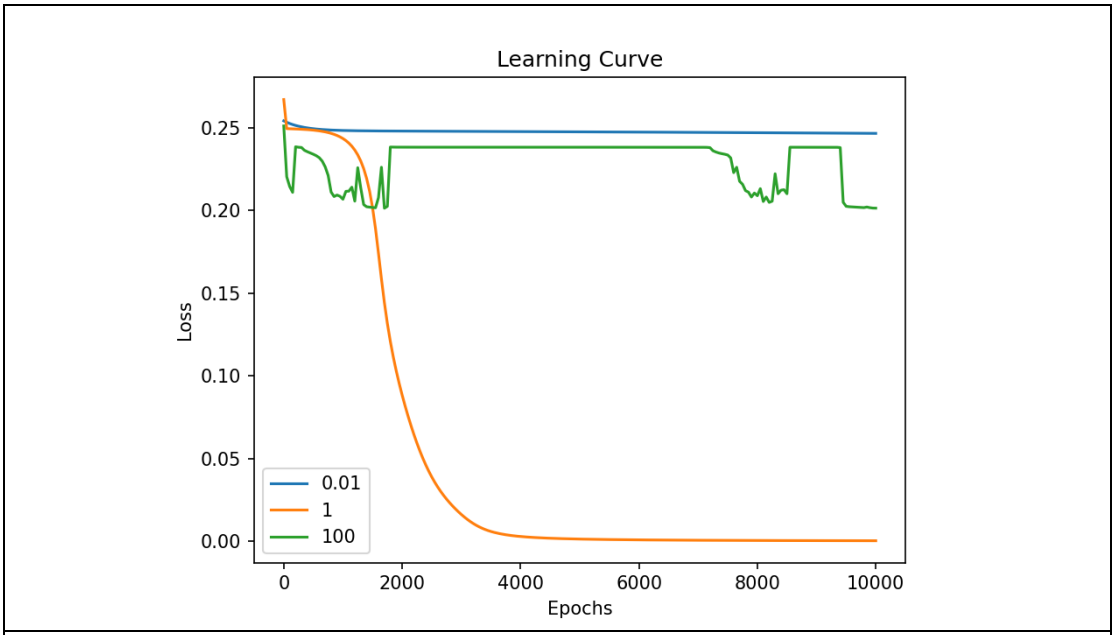
1. 調整成不同的 learning

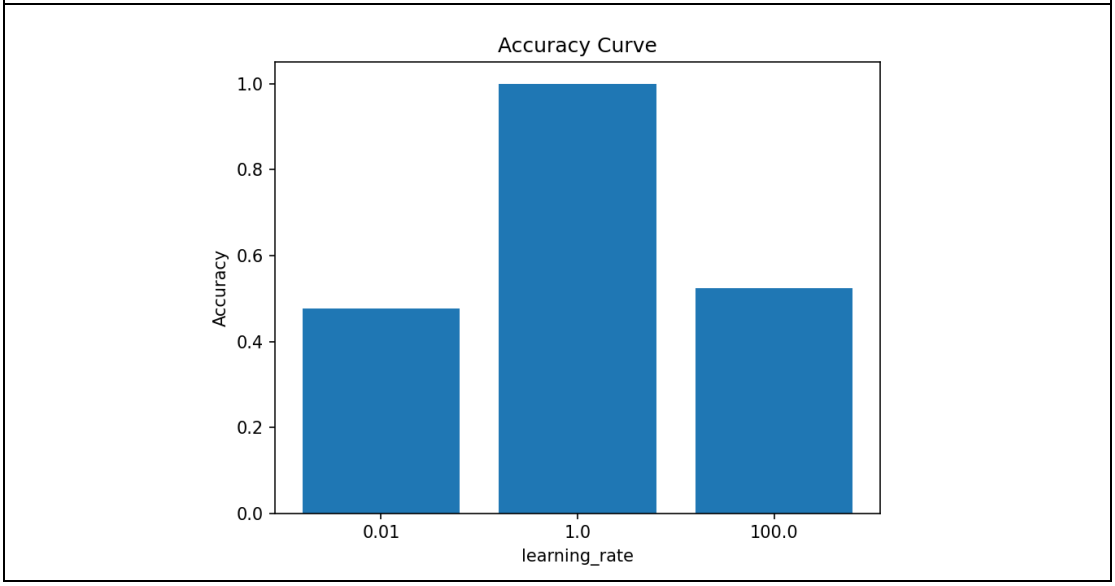
Liner:




Accuracy Curve
Accuracy
1.0
0.8
0.6
0.4
0.2
0.0
0.01
1.0
100.0
learning_rate

XOR:


Learning Curve
Loss
0.25
0.20
0.15
0.10
0.05
0.00
0.01
1
100
0
2000
4000
6000
8000
10000
Epochs

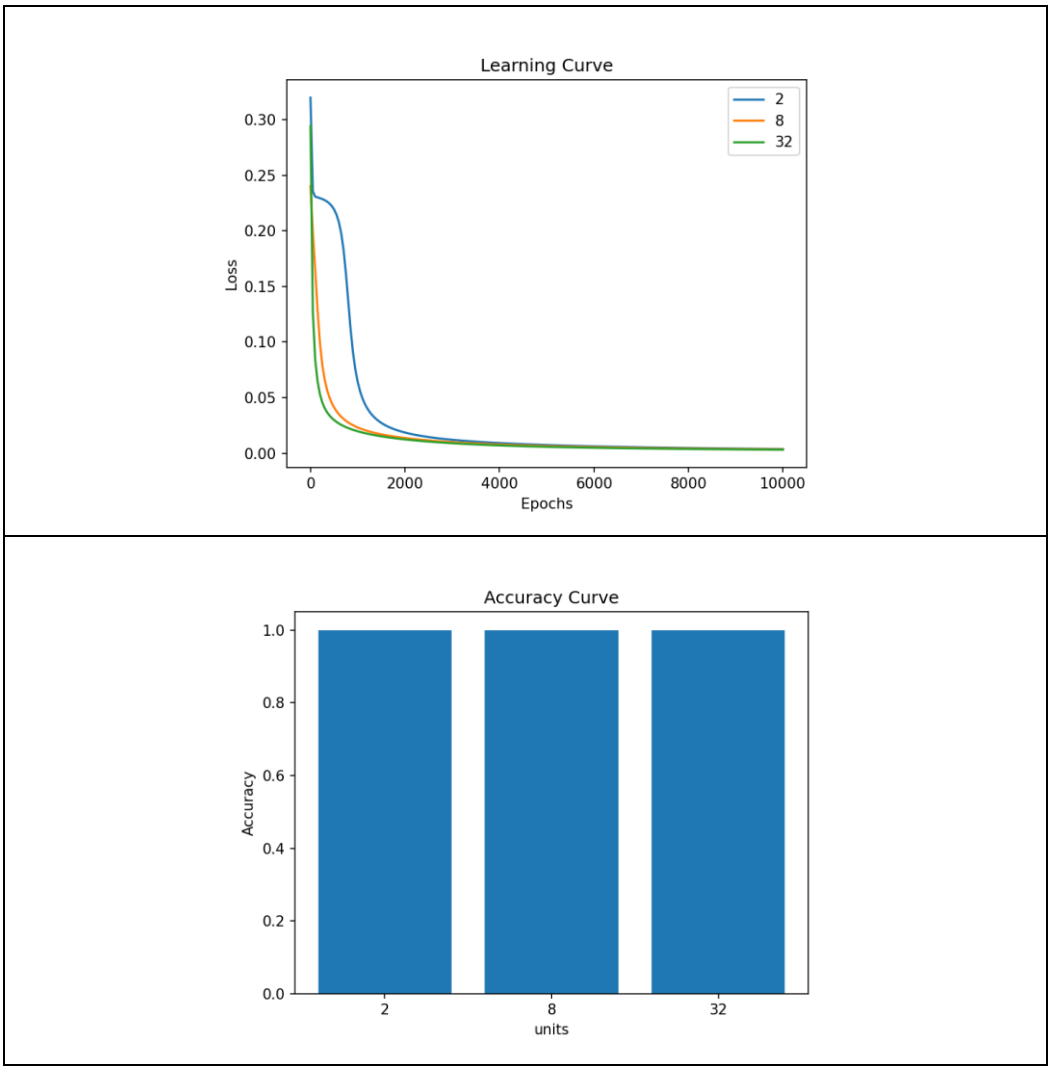

Accuracy Curve
Accuracy
1.0
0.8
0.6
0.4
0.2
0.0
0.01
1.0
100.0
learning_rate

討論:

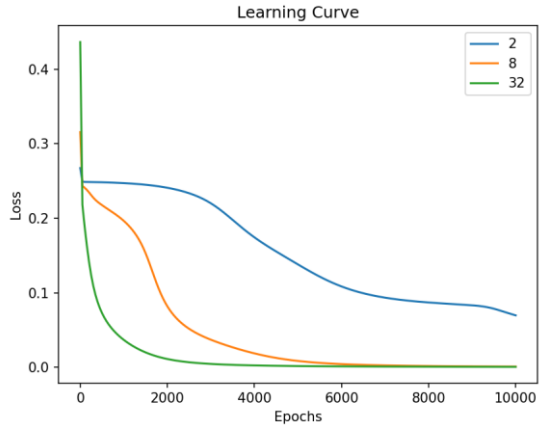
這裡可以很明顯的看到,learning rate 的調整是很重要的,如果 learnging rate 太大的話，可能會一直在最佳解附近徘徊，導致損失函數的曲線看起來一直震動，而如果學習率太小的話，曲線的收斂速度就會很慢，因為每一次的變動都很小,因此適當的學習綠是很重要的這樣才能透過適當的調整,並且更新參數。

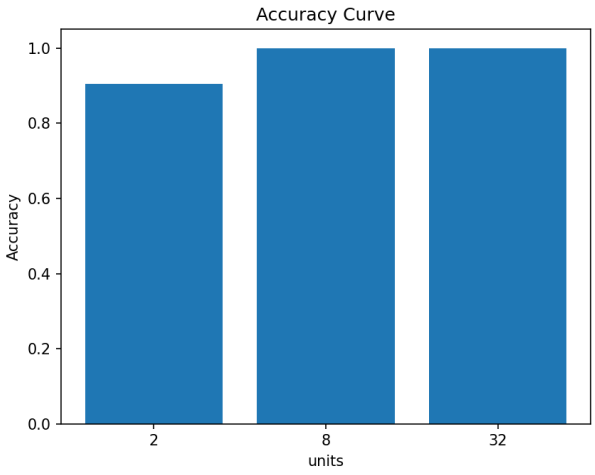
2. 調整 hidden layers 神經元個數

Liner



XOR



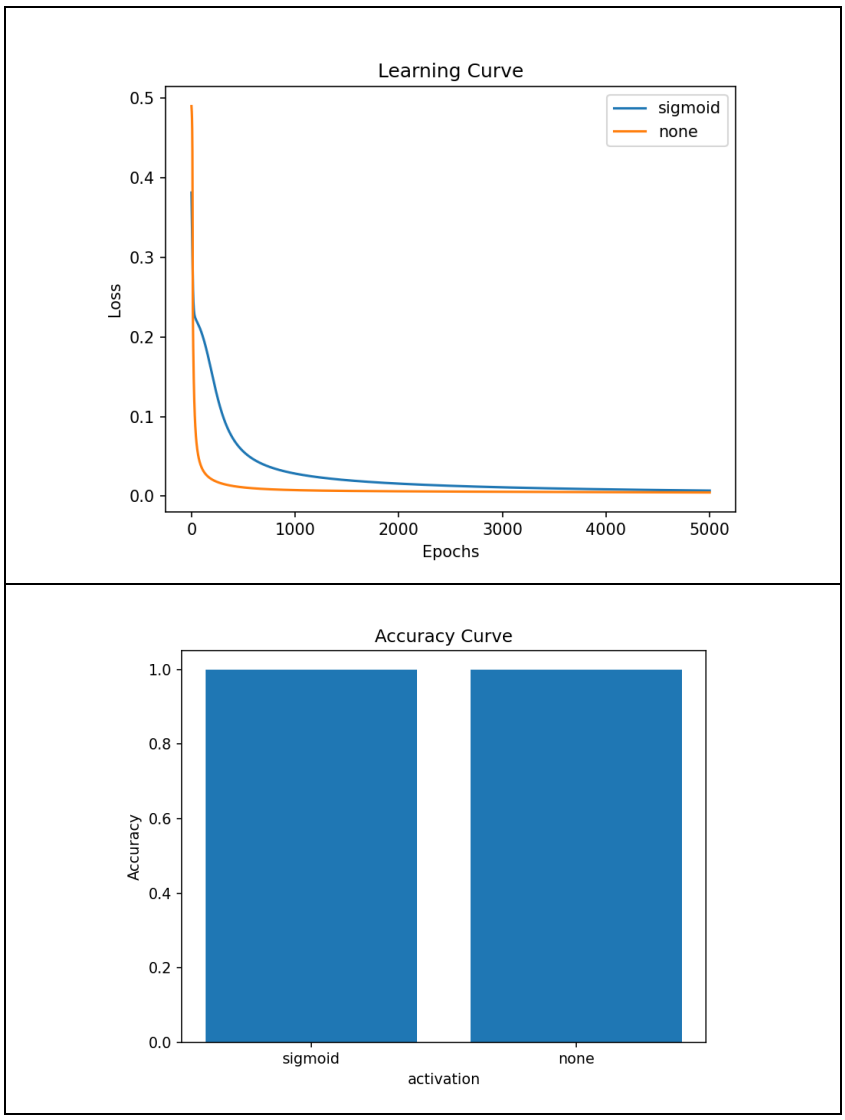


討論:

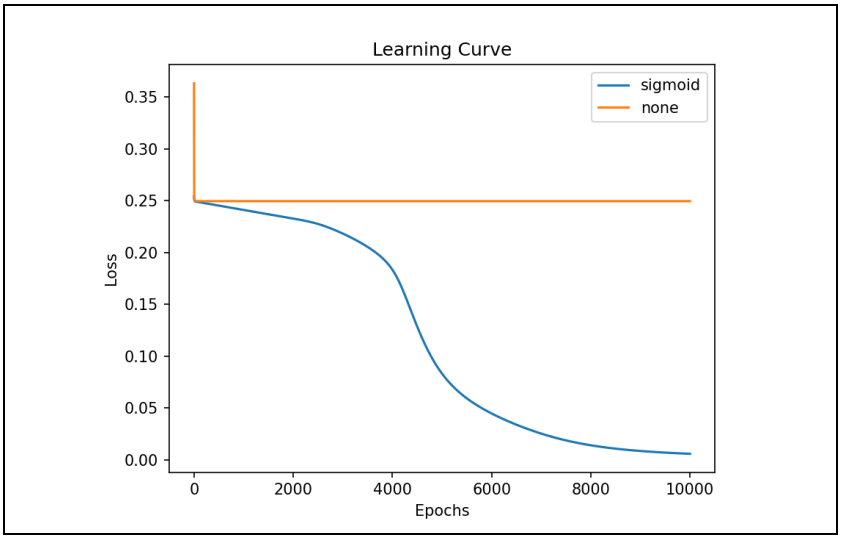
而在可以看到，當神經元個數增加的時候，很明顯的曲線的收斂速度比較快，而在 XOR 資料集中，可以看到神經元較少時，因為模型的表示能力會受到限制，因此對於這種非線性資料的表現會比較差，而因為神經元較多，模型的表示能力會增加，因此在這種非線性的資料集中，表現會較好，但是並不是神經元越大就越好，因為越大就代表著參數量會增加，運算速度可能會變慢，因此選擇合適的神經元是很重要的。

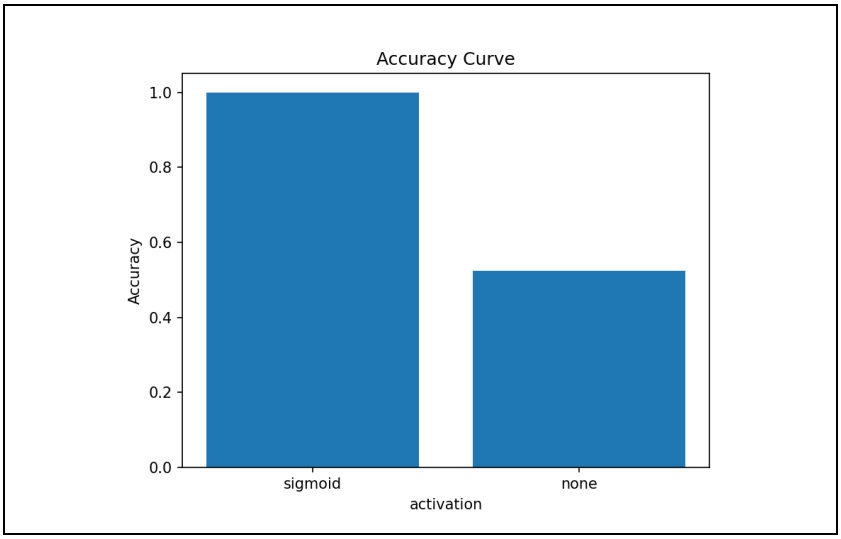
3. 激勵函數比較

Liner



XOR:





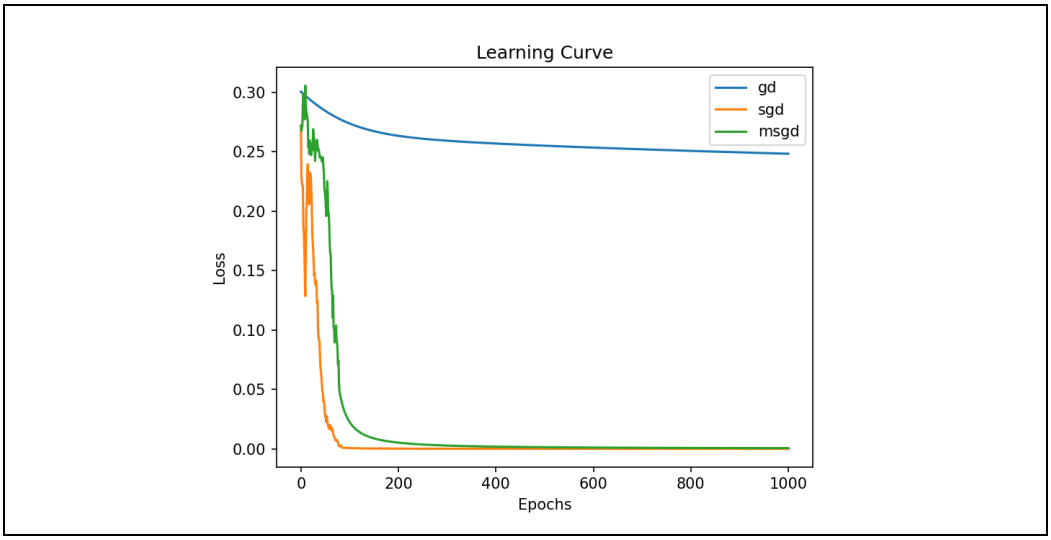
討論:

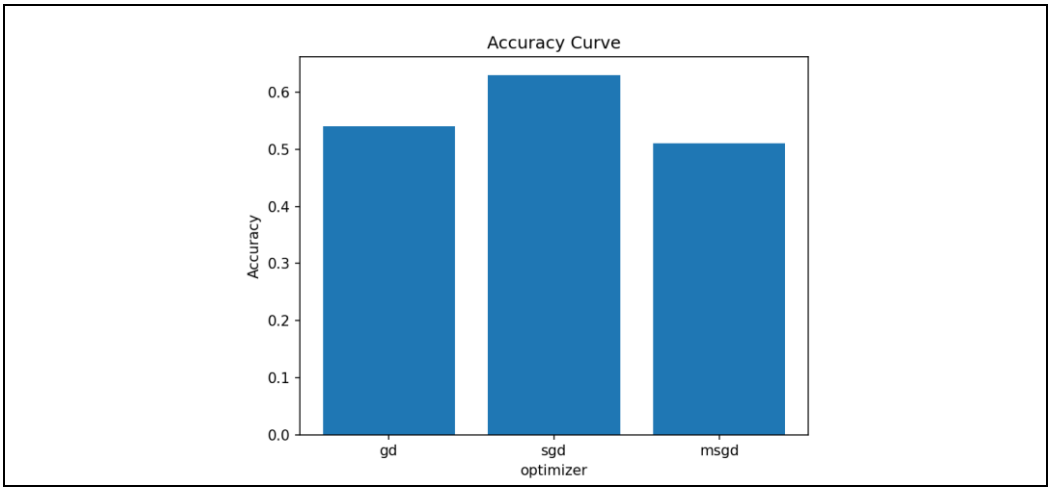
在這裡我們可以看到，在有沒有激勵函數的情況下，在線性資料集的表現上都差不多，我認為這是因為線性資料集中，只需要簡單的線性組合就能夠表示了，所以在表現上看起來都差不多，甚至還更快收斂，但是在非線性的資料集中，表現就非常差，因此適當的加入激勵函數，才能讓模型有更強大的表現能力。

5. Extra

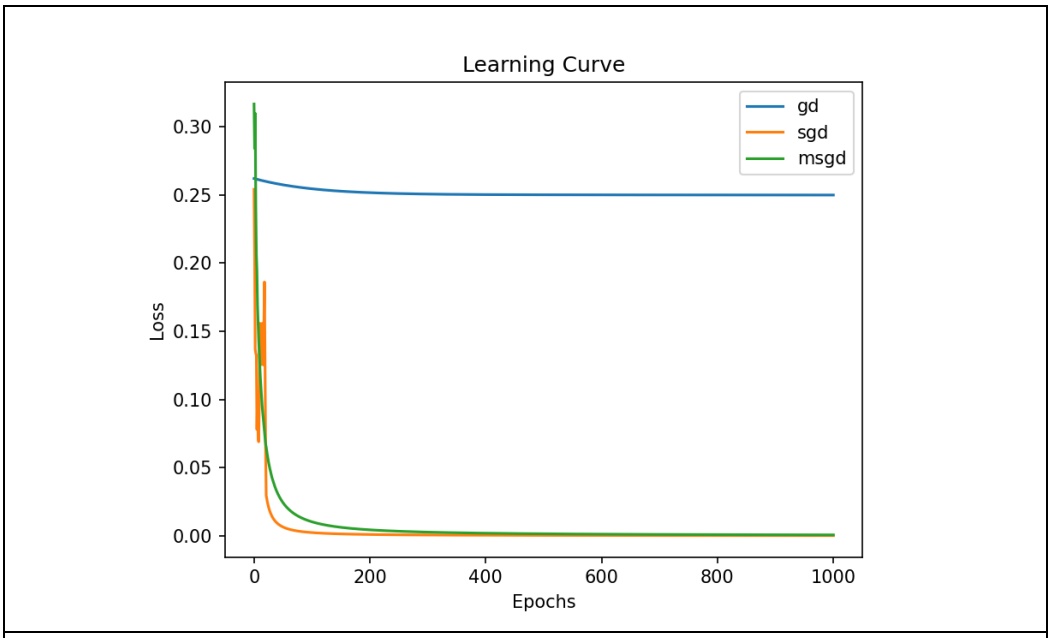
1. 實現不同優化器

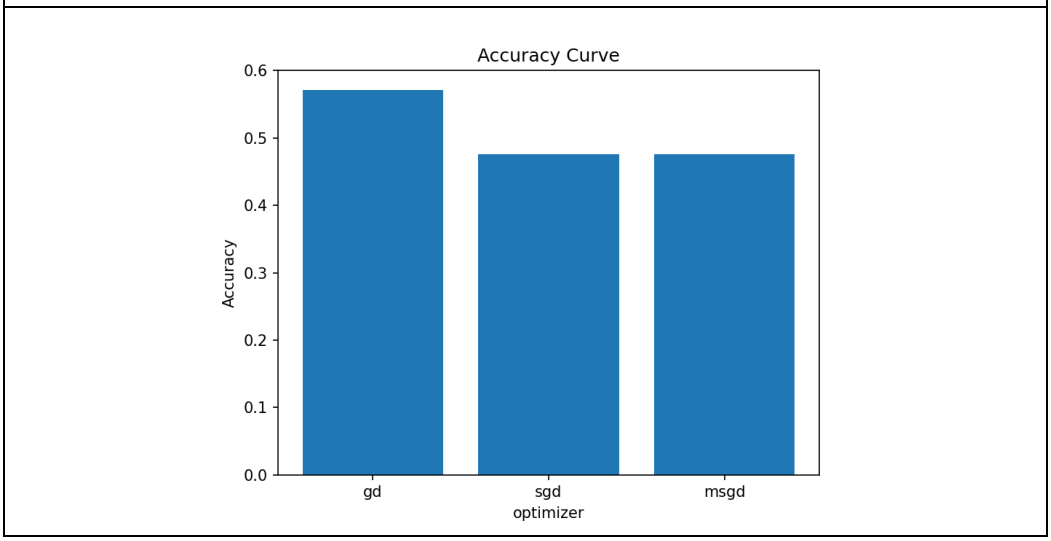
Liner





XOR





討論:

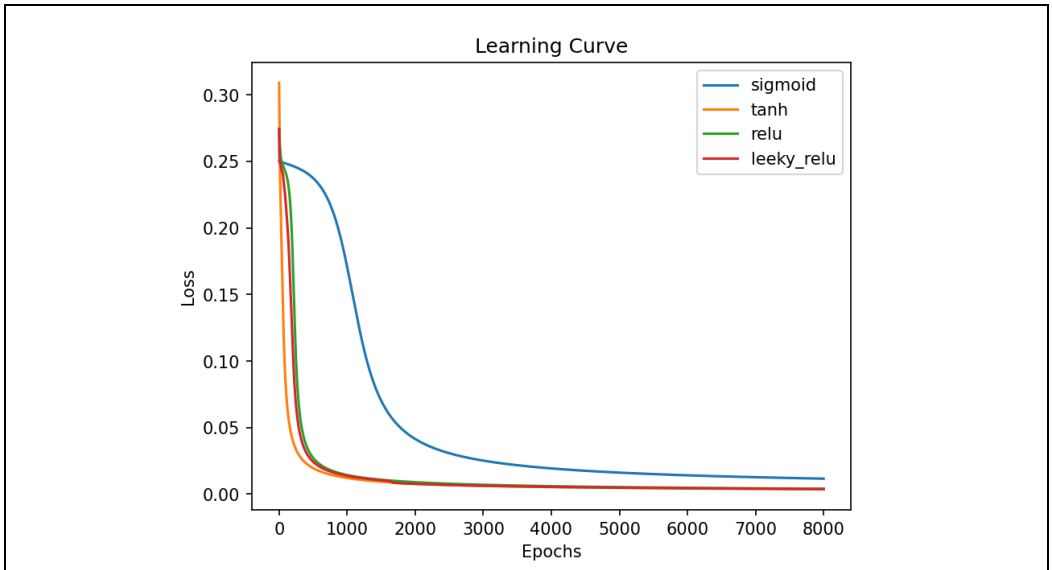
這裡我是用了三種不同的優化器來做比較，第一個是傳統的 Gradient Descent，第二個是 Stochastic Gradient Descent，第三個是 Min-Batch

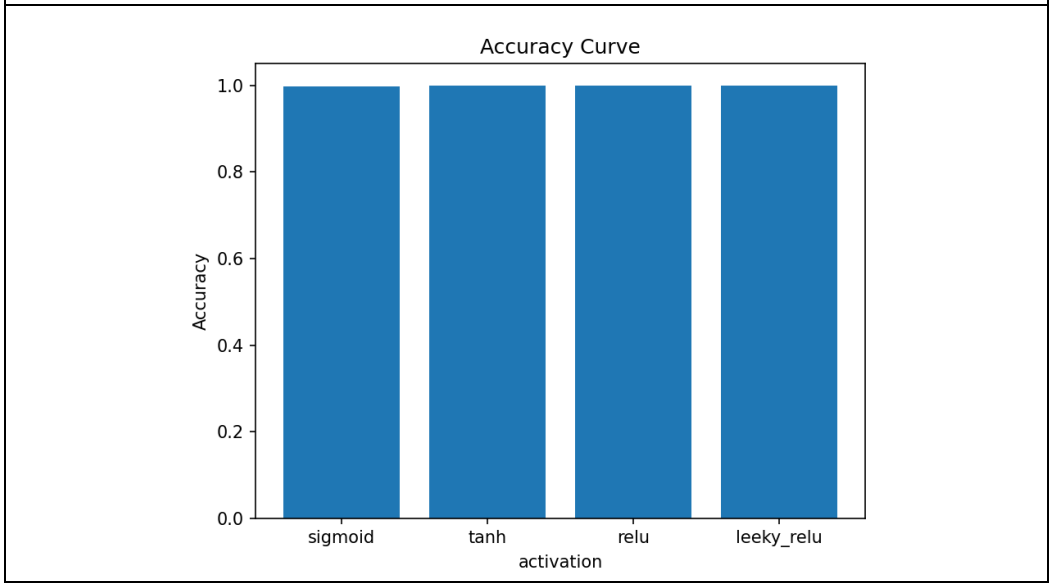
Stochastic Gradient Descent，而後面兩種算法都是為了改善傳統 Gradient Descent 每次更新時，都是使用全部的樣本來做梯度更新，在一些比較大的數據集中，這樣的做法會導致效能降低，而且引入大量的資料集也有可能造成內存不足的問題，而透過 SGD 每次更新時只會使用一個樣本來做更新，這樣做讓計算成本降低，但也很有可能會因為它的隨機性，導致它在最佳解中一直震盪，因此實現了第三種優化器算法 MSGD，這是計算成本以及模型效能的折衷方案，使用小批次數據的梯度來更新模型參數。它介於 GD 和 SGD 之間，每次迭代中使用部分訓練樣本，這保持了部分樣本的隨機性，又減輕了計算成本。

在數據圖中，我們也可以看到，相對於 gd 來說，sgd 跟 msgd 都更容易出現震盪，但是通常在傳統 gd 的作法，比較容易收斂到局部最佳解。

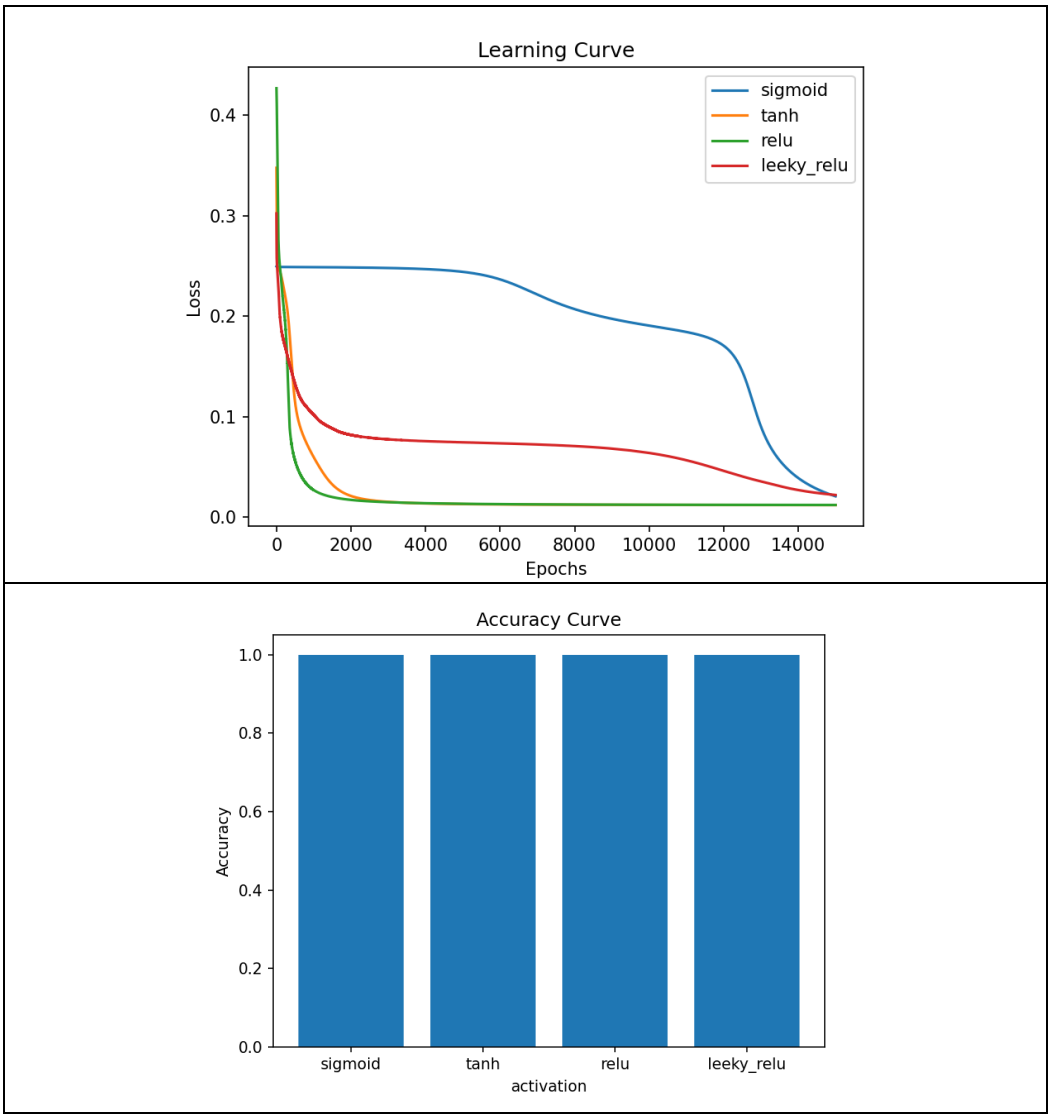
2. 實現不同激勵函數

Liner





XOR



討論:

在這裡我們可以看到 sigmoid，在這兩個資料集中，表現最差，因為它的輸出範圍有限，但這裡比較讓我意外的是 tahh 的表現居然跟 ReLU 以及 leeky ReLU 差不是到太多，不太確定是不是權重初始化的問題，但這裡還是可以看到正常來說，ReLU 跟 leeky ReLU 的表現還是會比較好，因為他們比較不會遭遇到梯度消失的問題，但在實驗的時候，發生一個插曲，訓練時，少數幾次會造成權重不是很明顯地在更新，應該說，可以當成沒更新，在查了資料才發現，這叫 ReLU dead，這種情況通常發生在神經元的輸入小於或等於零的情況下，因為 ReLU 函數在輸入小於等於零時會返回零，導致神經元的權重不會更新，也無法再被激活，因此可能要再學習率跟權重初始化的設置中多加注意。