DLP lab01

1.Introduction

在這次的 lab 中，需要實作 neural network，不過為了瞭解神經網路如何 update 參數，使 model 能夠有效的預測分類，因此，在這次的作業中，必須了解 backpropagation 的運算邏輯，僅使用 NumPy 完成程式，其中，我使用了 SGD 來優化參數，cost function 的部分則使用了 MSE。

2.Experiment setups

A. Sigmoid functions

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

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

在這次的作業中，我首先嘗試了使用 sigmoid 作為 activate function，根據定義，$\sigma(x) = \frac{1}{1+e^{-x}}$ ，而在 backpropagation 的部分，需要對 sigmoid function 進行一階微分，公式如下$\frac{d\sigma}{dx} = \sigma(x) \times \big(1 - \sigma(x)\big)$，而受惠於我們已經在 feed forward 的階段就對$\sigma(x)$進行計算，因此在 derivate_sigmoid(x)的 function 中，只需要帶入$\sigma(x)$的值就好，不必重新計算。

B. Neural network

至於神經網路的架構，依照作業的要求，我使用了兩層的 hidden layer，因此會有 3 個 weight 參數需要被訓練，而 cost function 的部分我則使用了 Mean Square Error ，公式如下: MSE $= \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$ ，因為 MSE 計算時有平方項，所以$\hat{y}$ 和 y的順序並沒有影響，但是 back propagation 就要注意相減的順序，來決定要不要負號。

C. Backpropagation

為了得到$\frac{dL}{dW_i}$(where i = 1,2,3)的值來對 weight 進行優化，我們需要一連串的 chain rule，如下圖所示。

```
def back_propagation(self , y_gt):
    self.dc_dy = derivate_MSE(y_gt, self.y_pred)
    self.dy_dz3 = derivate_sigmoid(self.y_pred)
    self.dz3_dw3 = self.a_2
    self.dc_dz3 =self.dy_dz3 * self.dc_dy
    self.dc_dw3 = self.dz3_dw3.T @ self.dc_dz3

    self.dz2_dw2 = self.a_1
    self.dz3_da2 = self.weight_3
    self.dc_da2 = self.dz3_da2 @ self.dc_dz3.T
    self.dc_dz2 = self.derivate_activate_function(self.a_2) * self.dc_da2.T
    self.dc_dw2 = self.dz2_dw2.T @ self.dc_dz2


    self.dz1_dw1 = self.x
    self.dz2_da1 = self.weight_2
    self.dc_da1 = self.dz2_da1 @ self.dc_dz2.T
    self.dc_dz1 = self.derivate_activate_function(self.a_1) * self.dc_da1.T
    self.dc_dw1 = self.dz1_dw1.T @ self.dc_dz1
```

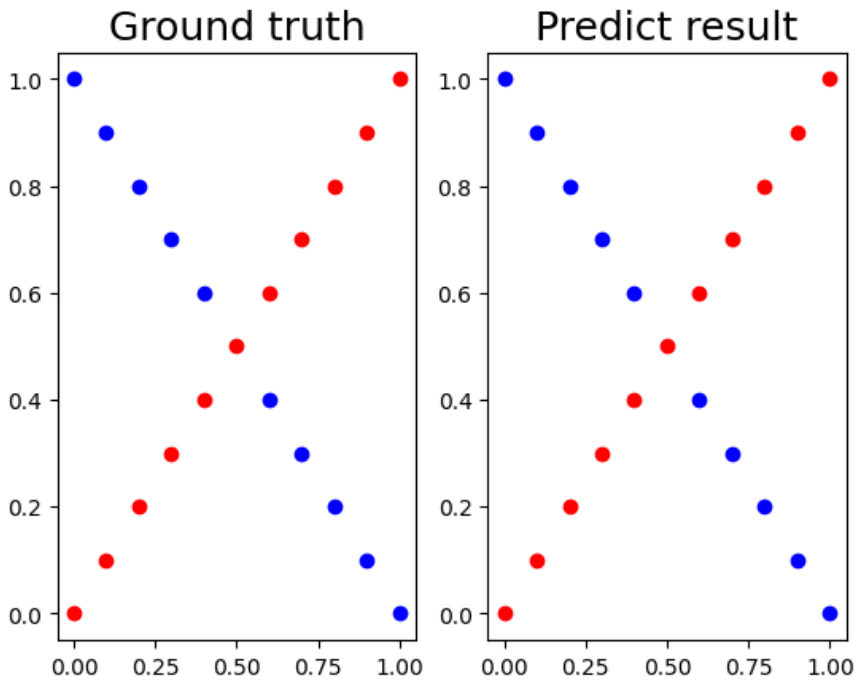
而在計算完$\frac{dL}{dW_i}$後，就可以更新權重了，而在這次的作業中，我使用了 Gradient Descent 作為 Optimizer，在這裡使用 GD 的原因而不是 SGD 的原因是因為這次作業的訓練資料數量相當的小，因此在實作上，我採取了計算完所有訓練資料的 Gradient 後才更新權重。

```
self.weight_1 -= self.lr*self.dc_dw1
self.weight_2 -= self.lr*self.dc_dw2
self.weight_3 -= self.lr*self.dc_dw3
```

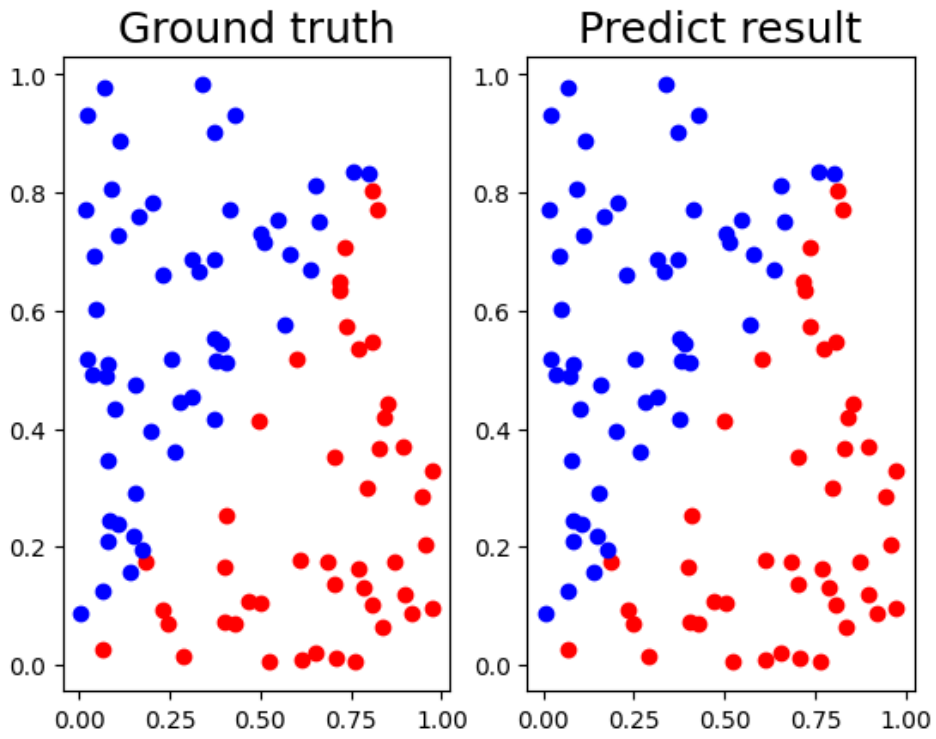
3. Results of your testing

A. Screenshot and comparison figure

XOR DATA:



Linear DATA:



B. Show the accuracy of your prediction

XOR DATA:

Train loss:

```
epoch 5000 loss : 0.000581
epoch 10000 loss : 0.000237
epoch 15000 loss : 0.000146
epoch 20000 loss : 0.000105
epoch 25000 loss : 0.000082
epoch 30000 loss : 0.000067
epoch 35000 loss : 0.000056
epoch 40000 loss : 0.000049
epoch 45000 loss : 0.000043
epoch 50000 loss : 0.000038
epoch 55000 loss : 0.000034
epoch 60000 loss : 0.000031
epoch 65000 loss : 0.000029
epoch 70000 loss : 0.000027
epoch 75000 loss : 0.000025
epoch 80000 loss : 0.000023
epoch 85000 loss : 0.000022
epoch 90000 loss : 0.000020
epoch 95000 loss : 0.000019
epoch 100000 loss : 0.000018
```

Accuracy

```
accuracy 100.0%
```

Linear DATA:

Train loss:

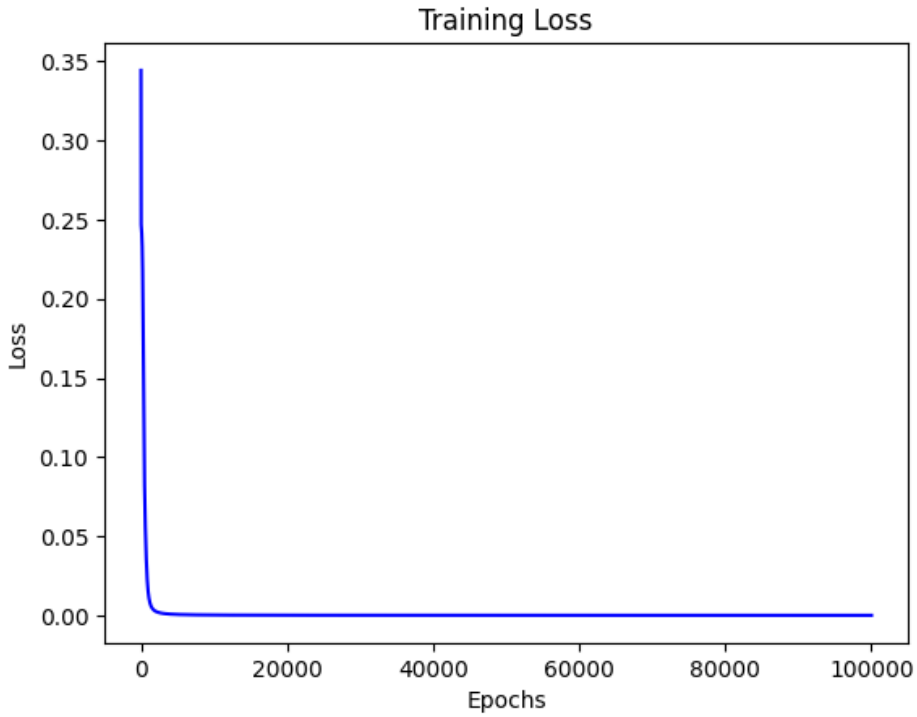
```
epoch 5000 loss : 0.005087
epoch 10000 loss : 0.002745
epoch 15000 loss : 0.001745
epoch 20000 loss : 0.001190
epoch 25000 loss : 0.000850
epoch 30000 loss : 0.000630
epoch 35000 loss : 0.000482
epoch 40000 loss : 0.000378
epoch 45000 loss : 0.000303
epoch 50000 loss : 0.000248
epoch 55000 loss : 0.000205
epoch 60000 loss : 0.000173
epoch 65000 loss : 0.000147
epoch 70000 loss : 0.000126
epoch 75000 loss : 0.000110
epoch 80000 loss : 0.000096
epoch 85000 loss : 0.000085
epoch 90000 loss : 0.000076
epoch 95000 loss : 0.000068
epoch 100000 loss : 0.000062
```

Accuracy

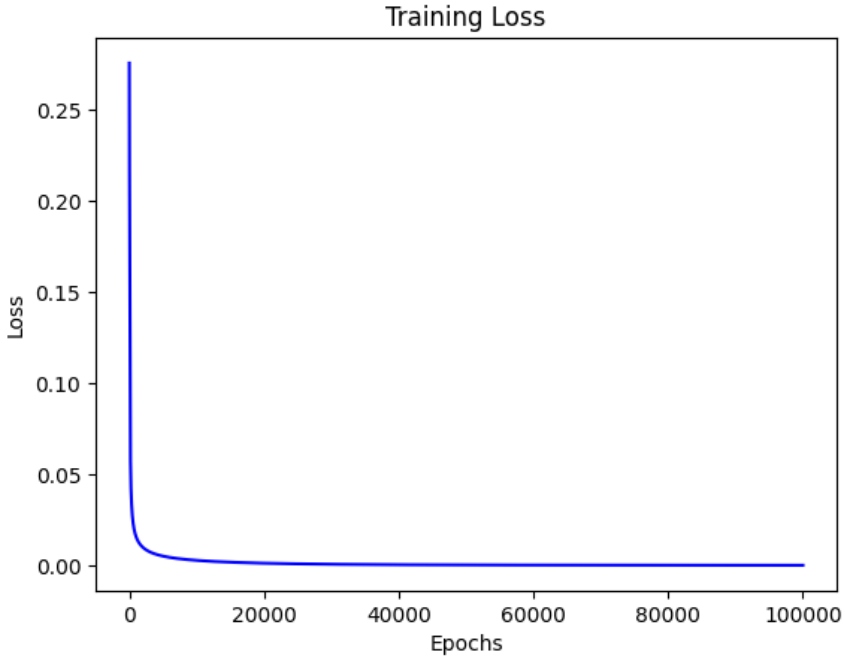
```
accuracy 100.0%
```

C. Learning curve (loss, epoch curve)

XOR DATA (learning rate = 1, optimizer = GD, activate function = sigmoid)



Linear DATA (learning rate = 1, optimizer = GD, activate function = sigmoid)



D. Anything you want to present

可以發現在 XOR data 和 Linear data 上，accuracy rate 都達到了 100% ，也就說

明，我們使用了 sigmoid 等 non-linear transform 方式，能夠成功的將 input data map 到正確的預測值之上。

4. Discussion

A. Try different learning rates

(metric: accuracy, epochs:100000)

	Linear	XOR
0.1	100%	100%
1	100%	100%
10	100%	100%
100	59.0%	52.3%

可以發現當 learning rate 過大時，model 可能無法收斂，進而預測全部紅色或全部藍色。

B. Try different numbers of hidden units

(metric: accuracy, epochs:100000)

(hidden layer1, hidden layer2)

	Linear	XOR
1,1	55.0%	52.3%
3,3	100%	100%
1,9	100%	66.7%
9,1	56%	52.3%
9,9	100%	100%

可以發現如果 hidden units 數量太少，model 無法收斂，此外我也嘗試使用了 (1,9) 和(9,9)兩種配置，結果效果皆比(3,3)差，可見較為對稱的 hidden units 有助於模型收斂。

C. Try without activation functions

(metric: accuracy, epochs:100000)

	Linear	XOR
With activation function	100%	100%
Without activation function	100%	52.3%

在沒有 activation function 的狀況下，可以發現 XOR 完全沒有預測能力，這是因為沒有 activation function 這種非線性轉換，也無法找到一條線能夠將 XOR 這種分布的 data 良好的分類。

D. Anything you want to share

在嘗試不同的 activation function 時，發現在 linear 的 case 中，只使用 Relu 作為 activation function 時，有時候並不會收斂，而一開始以為是隨機性的問題，但透過打印出 predict 的結果時，發現其值並不介於 0-1 之間，這也因此，沒有一個良好的 threshold 將結果切成 0 或 1，因此，在最後一層中，我使用了 sigmoid

function 作為 final activate function。

5. Extra

A. Implement different optimizers.

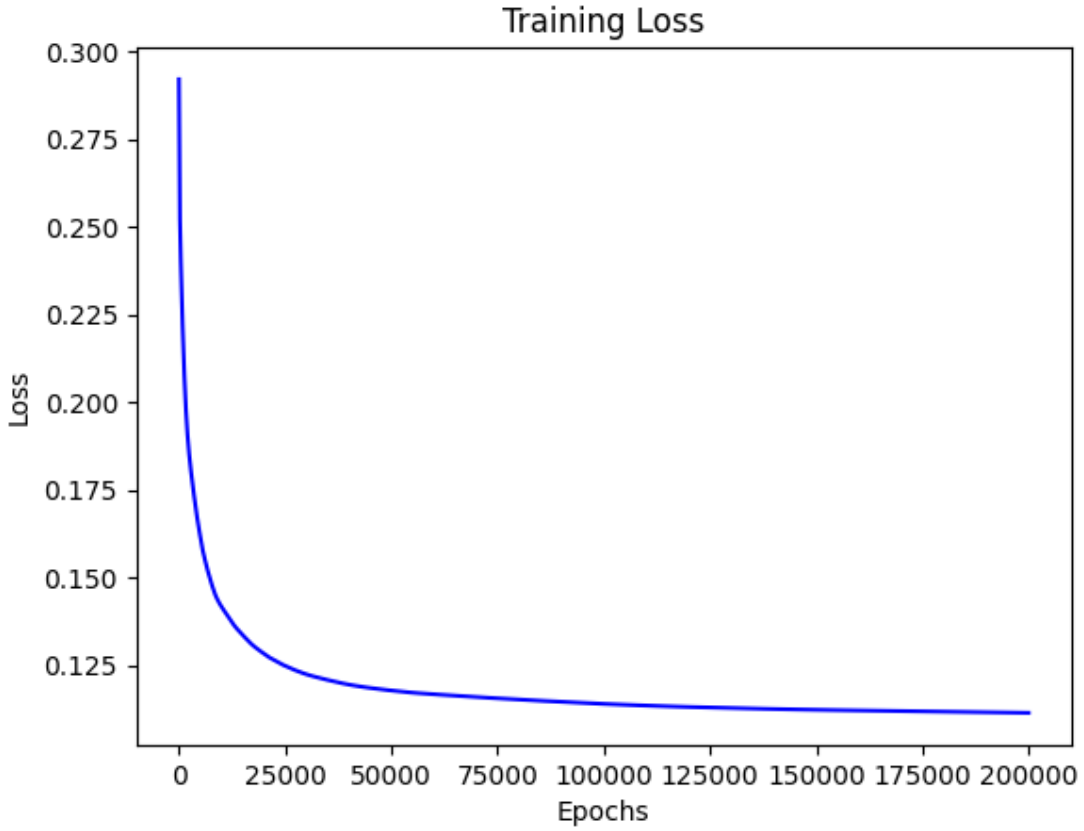
在這次的作業中，我除了實作 Gradient Descent 外，也實作了 Momentum，其公式如下，$V_t \leftarrow \beta V_{t-1} - \eta \frac{\partial L}{\partial W}$ ， $W \leftarrow W + V_t$ ，而這種特性，往往有比 Gradient Descent 更容易收斂，而實作上也相對簡單，只需要記住前一次的梯度值就好，以下為 code 的實作細節。

```
#update
if self.op == 'momentum':
    self.velocity_1 = self.velocity_1 * self.beta - self.lr*self.dc_dw1
    self.weight_1 = self.weight_1 + self.velocity_1
    self.velocity_2 = self.velocity_2 * self.beta - self.lr*self.dc_dw2
    self.weight_2 = self.weight_2 + self.velocity_2
    self.velocity_3 = self.velocity_3 * self.beta - self.lr*self.dc_dw3
    self.weight_3 = self.weight_3 + self.velocity_3
```

下圖為 Gradient Descent 和 momentum 的 learning curve 圖，可以發現 momentum 的收斂速度較快，這也符合原本 momentum 的設計意義。

GD

Momentum:



B. Implement different activation functions.

除了 sigmoid function 我也實作了 Relu activate function，Relu function 只會有正數，因此，只要使用 np.maximum(0,x)就能簡單的表達出來，而其一階導數，在 X>0 的地方為 X 的微分，也就是等於 1，而小於等於 0 的地方則為 0。

```
def relu(x):
    return np.maximum(0,x)

def derivate_relu(x):
    return np.where(x > 0, 1 , 0)
```