# Lab1 back-propagation

資訊科學與工程研究所 313551079 張芷烜

## 1. Introduction

Lab1 目標是在不使用 TensorFlow 或是 PyTorch framework 下，使用 NumPy 和 Python 標準函式庫，實現一個具有兩層 hidden layers 的簡單 neuron network，並且需要完成 model 中 forward propagation 和 backpropagation 的功能，透過設定不同 epoch、learning rate、hidden layer unit numbers 和 activation function 訓練資料，並比較不同設定下的差異與 linear 和 XOR 這兩筆資料的訓練設計差別，透過這項練習可以探討每個參數設定和 loss function 與 activation function 選用的差異性，更了解 backpropagation 中如何透過 gradient descent 去 update 權重，並且比較了 linear data 與 non-linear data 的訓練差異。

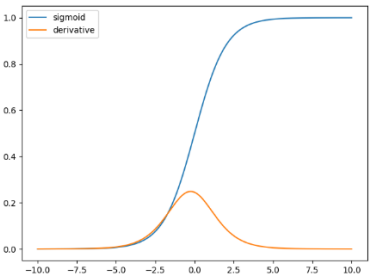
## 2. Implementation Details

A. Sigmoid function

Sigmoid function 是一種常見的 activation function，他可以將數字映射到 [0 , 1] 之間，是一個連續可微分的單調遞增函數，這個函數很適合用在二分類問題上，輸出可以看做機率分佈，在二分問題下以 0.5 作為判斷邊界，缺點是導數的最大值是 0.25，當網路層數較深時，梯度在反向傳播過程中會逐層縮小，其公式如下：

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\sigma'(x) = \frac{d}{dx}\sigma(x) = \sigma(x)\big(1 - \sigma(x)\big)$$




```
def forward(self,x):
    if self.act_type == 'relu':
        return np.maximum(0,x)
    elif self.act_type == 'sigmoid':
        return 1.0/(1.0+np.exp(-x))
    elif self.act_type == 'tanh':
        return np.tanh(x)
    elif self.act_type == 'none':
        return x
```

```
def derivative(self,x):
    if self.act_type == 'relu':
        return (x > 0).astype(float)
    elif self.act_type == 'sigmoid':
        return np.multiply(x,1-x)
    elif self.act_type == 'tanh':
        return 1 - np.tanh(x)**2
    elif self.act_type == 'none':
        return np.ones_like(x)
```

- 這邊實作方法與助教提供的參考相同

B. Neural network architecture

整個 NN 的架構我設定每層 hidden layer 有 10 個神經元，並且中間層可以選用 ReLU、sigmoid 或是 tanh 作為 activation function ，或是不選用 activation function，輸出層則以 sigmoid 作為 activation function。

在初始化架構中，需要先隨機初始化 weight 和 bias，之後進行 forward pass 計算預設結果，根據預測結果與 ground truth 透過 MSE 計算 loss 後對其透過 backward 計算每個節點的 gradient 後根據其設定的 learning rate 更新權重，直至準確率到達 100%或是 epoch 到達設定量。

$$z1 = w1 \cdot x + b1$$
$$a1 = \sigma(z1)$$
$$z2 = w2 \cdot a1 + b2$$
$$a2 = \sigma(z2)$$
$$z3 = w3 \cdot a2 + b3$$
$$a3 = \sigma(z3)$$
$$L = \frac{1}{m}\sum_{i=1}^{m}(y_{pred} - y\_true)^2$$

```
def forward(self,x):
    # z = act(Wx + b)
    self.z1 = np.dot(x,self.W1) + self.b1
    self.a1 = self.act.forward(self.z1)

    self.z2 = np.dot(self.a1,self.W2) + self.b2
    self.a2 = self.act.forward(self.z2)

    self.z3 = np.dot(self.a2,self.W3) + self.b3
    self.a3 = self.out_act.forward(self.z3)

    return self.a3
```

```
def compute_loss(y_pred,y_true):
    # MSE loss
    loss = np.mean((y_pred - y_true)**2)
    return loss
```

這樣最後透過 sigmoid 的 a3 是一個 0~1 之間的值，選用 0.5 最為邊界判斷，小於 0.5 label 為 0，反之 label 為 1，透過這樣一個設計即可很好的訓練和預測資料。

C. Backpropagation

Backpropagation 是此次 lab 的核心，主要概念為上述所提到的透過 loss function 計算 loss 後，利用 gradient descent 來循環更新 W 和 b, 而之所以稱作 backpropagation 是因為計算 gradient 的過程要由最後一層開始往前推。Backpropagation 基於 Chain Rule 反算，下面為反向推導的過程(這邊以 MSE 為例)

根據 gradient descent 我們需要知道 gradient 已更新 w 和 b

$$w = w - lr \cdot \frac{\partial L}{\partial w}, b = b - lr \cdot \frac{\partial L}{\partial b}$$

首先根據 chain rule 為了計算 $\frac{\partial L}{\partial w3} = \frac{\partial L}{\partial a3} \cdot \frac{\partial a3}{\partial z3} \cdot \frac{\partial z3}{\partial w3}$ ，要計算 $\frac{\partial L}{\partial a3} = \frac{2}{m}(a3 - y)$ ，

再來是 $\frac{\partial a3}{\partial z3}$ ，已知 $a3 = \sigma(z3)$ 因此 $\frac{\partial a3}{\partial z3} = a3(1 - a3)$

最後是 $\frac{\partial z3}{\partial w}, z3 = w3 \cdot a2 + b3, \frac{\partial z3}{\partial w3} = a2$

可以得知，$\frac{\partial L}{\partial z3} = (a3 - y)\sigma'(a3)$ 省略常數項 $\frac{\partial L}{\partial w3} = a2^T \cdot \frac{\partial L}{\partial z3}$ 以此類推每一層

```python
def backward(self,x,y):
    m = x.shape[0]
    dz3 = (self.a3 - y)*self.out_act.derivative(self.a3)
    dW3 = np.dot(self.a2.T,dz3)/m
    db3 = np.sum(dz3,axis=0,keepdims=True)/m

    dz2 = np.dot(dz3,self.W3.T)*self.act.derivative(self.a2)
    dW2 = np.dot(self.a1.T,dz2)/m
    db2 = np.sum(dz2,axis=0,keepdims=True)/m

    dz1 = np.dot(dz2,self.W2.T)*self.act.derivative(self.a1)
    dW1 = np.dot(x.T,dz1)/m
    db1 = np.sum(dz1,axis=0,keepdims=True)/m

    gradients = [dW1,db1,dW2,db2,dW3,db3]
    return gradients
```

```python
def update(self,gradients,learning_rate):
    self.W1 -= learning_rate*gradients[0]
    self.b1 -= learning_rate*gradients[1]

    self.W2 -= learning_rate*gradients[2]
    self.b2 -= learning_rate*gradients[3]

    self.W3 -= learning_rate*gradients[4]
    self.b3 -= learning_rate*gradients[5]
```

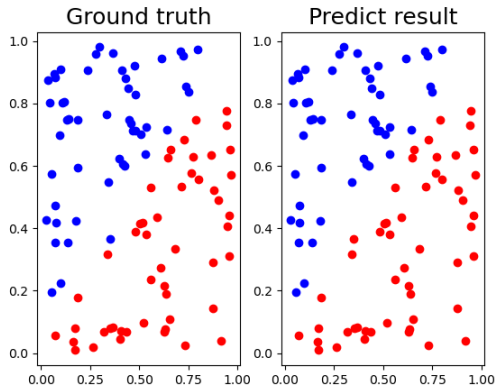
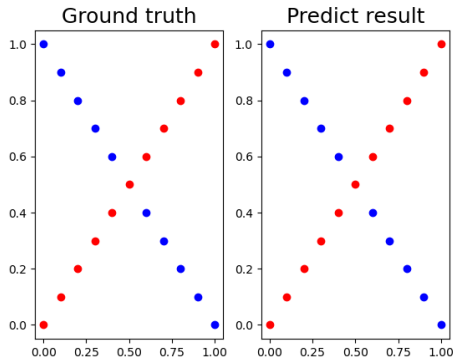
- 最後根據 gradient update 參數

## 3. Experimental Results

A. Screenshot and comparison figure

超參設定: 2 × hidden layer(10)，epoch 100000 (accuracy 到達 100% 提前停止)，learning rate 0.001，output activation function sigmoid(hidden layer use ReLu)，下面分別顯示 linear 和 XOR 的結果

| Linear training steps | Linear testing steps |
|---|---|
| epoch 4700 loss : 0.0562889909531398, acc : 0.9900<br>epoch 4800 loss : 0.0545770208369056, acc : 0.9900<br>epoch 4900 loss : 0.0530146813663937, acc : 0.9900<br>epoch 5000 loss : 0.0515673218788730, acc : 0.9900<br>epoch 5100 loss : 0.0502331715426095, acc : 0.9900<br>epoch 5200 loss : 0.0489965428957398, acc : 0.9900<br>epoch 5300 loss : 0.0478478118056776, acc : 0.9900<br>epoch 5400 loss : 0.0467683955412130, acc : 0.9900<br>epoch 5500 loss : 0.0457598576575927, acc : 0.9900<br>epoch 5600 loss : 0.0448113824228233, acc : 0.9900<br>epoch 5700 loss : 0.0439097619700750, acc : 0.9900<br>epoch 5800 loss : 0.0430504709701624, acc : 0.9900<br>epoch 5900 loss : 0.0422305576578647, acc : 0.9900<br>Training stopped early at epoch 5959 with 100% accuracy | Iter89 \| Ground truth: 0.0 \| prediction: 0.27406<br>Iter90 \| Ground truth: 1.0 \| prediction: 0.80309<br>Iter91 \| Ground truth: 0.0 \| prediction: 0.49999<br>Iter92 \| Ground truth: 1.0 \| prediction: 0.58962<br>Iter93 \| Ground truth: 0.0 \| prediction: 0.04249<br>Iter94 \| Ground truth: 1.0 \| prediction: 0.86636<br>Iter95 \| Ground truth: 1.0 \| prediction: 0.98409<br>Iter96 \| Ground truth: 0.0 \| prediction: 0.10470<br>Iter97 \| Ground truth: 1.0 \| prediction: 0.83070<br>Iter98 \| Ground truth: 0.0 \| prediction: 0.40939<br>Iter99 \| Ground truth: 1.0 \| prediction: 0.68076<br>Iter100 \| Ground truth: 1.0 \| prediction: 0.94507<br><br>loss=0.04176 accuracy=100.00% |
| **XOR training steps** | **XOR testing steps** |
| epoch 5800 loss : 0.0449733987212019, acc : 0.9524<br>epoch 5900 loss : 0.0440890305410489, acc : 0.9524<br>epoch 6000 loss : 0.0432938024163580, acc : 0.9524<br>epoch 6100 loss : 0.0425003813172234, acc : 0.9524<br>epoch 6200 loss : 0.0417147422840187, acc : 0.9524<br>epoch 6300 loss : 0.0409451099259470, acc : 0.9524<br>epoch 6400 loss : 0.0402133061818511, acc : 0.9524<br>epoch 6500 loss : 0.0394553013780882, acc : 0.9524<br>epoch 6600 loss : 0.0387291733642781, acc : 0.9524<br>epoch 6700 loss : 0.0380322024780133, acc : 0.9524<br>epoch 6800 loss : 0.0373315308803435, acc : 0.9524<br>epoch 6900 loss : 0.0365713720127167, acc : 0.9524<br>Training stopped early at epoch 6960 with 100% accuracy | Iter10 \| Ground truth: 1.0 \| prediction: 0.60364<br>Iter11 \| Ground truth: 0.0 \| prediction: 0.16543<br>Iter12 \| Ground truth: 0.0 \| prediction: 0.16548<br>Iter13 \| Ground truth: 1.0 \| prediction: 0.50002<br>Iter14 \| Ground truth: 0.0 \| prediction: 0.16676<br>Iter15 \| Ground truth: 1.0 \| prediction: 0.88533<br>Iter16 \| Ground truth: 0.0 \| prediction: 0.17439<br>Iter17 \| Ground truth: 1.0 \| prediction: 0.98355<br>Iter18 \| Ground truth: 0.0 \| prediction: 0.18242<br>Iter19 \| Ground truth: 1.0 \| prediction: 0.99676<br>Iter20 \| Ground truth: 0.0 \| prediction: 0.19074<br>Iter21 \| Ground truth: 1.0 \| prediction: 0.99934<br><br>loss=0.03617 accuracy=100.00% |

| Linear Result | XOR Result |
| --- | --- |
|  |  |

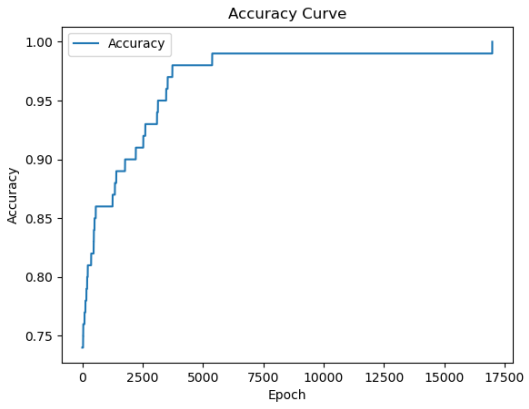
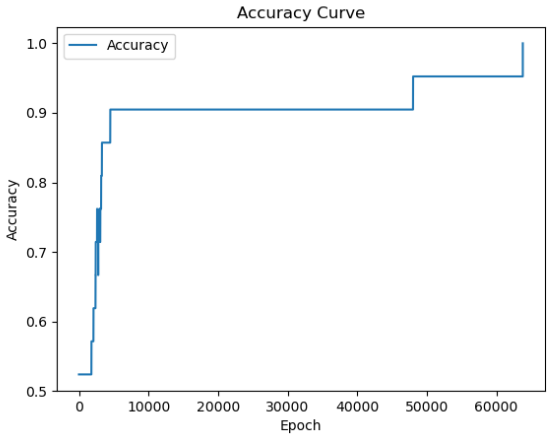


可以看出這兩個 data 最後都有成功收斂完成任務分類。

B. Show the accuracy of your prediction (40%) (achieve 90% accuracy)

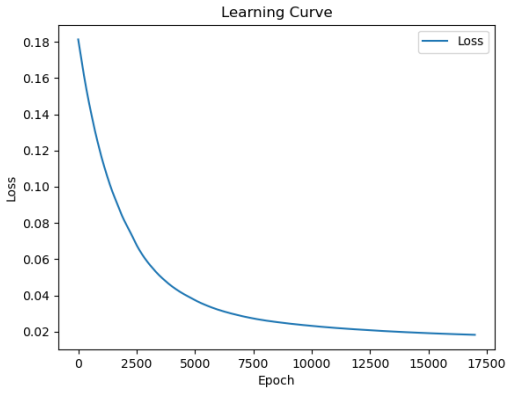
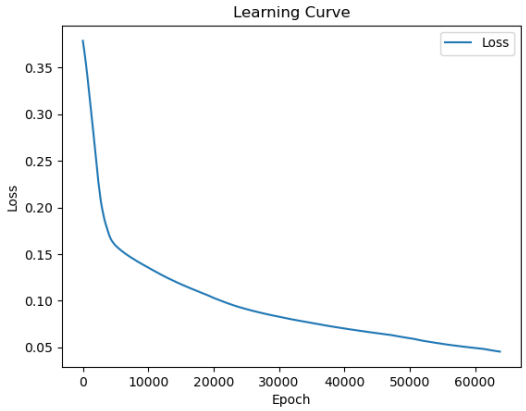
由上表結果可以看出兩者準確率都可以到達 100%，雖然根據不同的超參數設定，不 一定每次都能收斂到 100%，但依舊可以保有 90%以上的準確率

觀察正確率可以發現在前面幾個 epochs 就可以收斂到一定的正確率，推測其可能停在 local minimum 後需要花一定的時間跳出 local minimum，有可能對於這項任務來說 0.01 的 learning rate 過大，或是 loss function 需要重新設計。

| Linear Result | XOR Result |
| --- | --- |
|  |  |

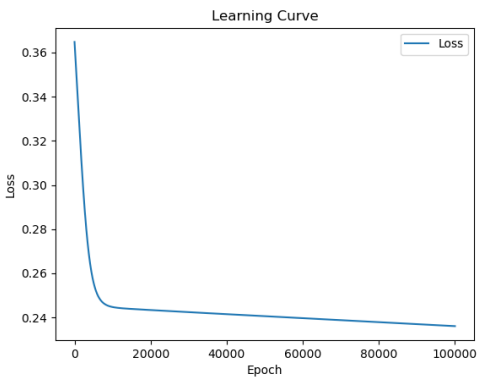
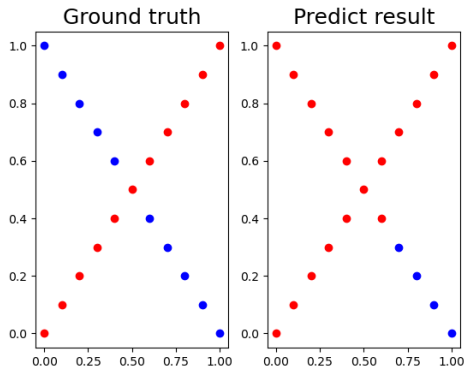
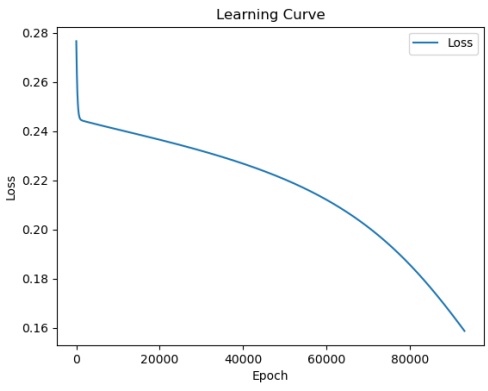
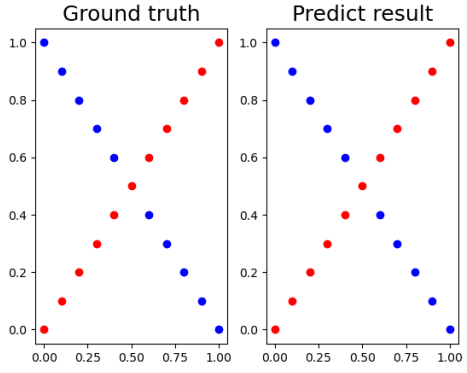
C. Learning curve (loss-epoch curve)

兩者的 loss function 都有穩定下降的趨勢，因此此 loss 設計符合訓練 data

| Linear | XOR |
| --- | --- |
|  |  |

D. Anything you want to present

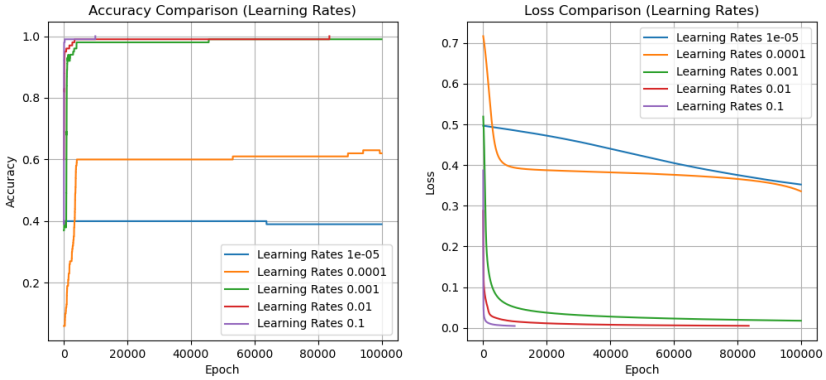
嘗試中間 hidden layer units 改用 sigmoid 作為 activation function，在 learning rate 設為 0.001 的設定下無法好好的收斂結果。Sigmoid 導函數最大為 0.25 的限制下，容易有梯度消失的問題，加上過小的學習率一直無法收斂到結果。調大學習率之後能提高收斂的機率，抑或是轉用 cross entropy 作為損失函數。

| XOR loss curve (lr = 0.001) | XOR result (lr = 0.001) |
| --- | --- |
|  |  |
| XOR loss curve (lr = 0.01) | XOR result (lr = 0.01) |
|  |  |

## 4. Discussion

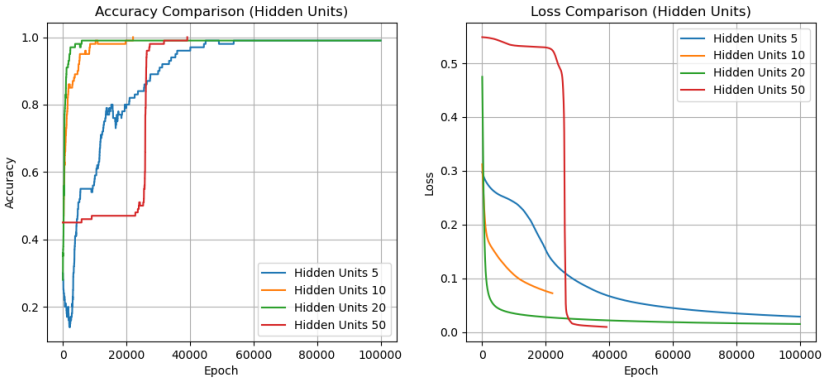
A. Try different learning rates

以 1e5，1e4，0.001，0.01，0.1 分別做測試，太快收斂或無法收斂都不是一個好的學習率，太快收斂可能會造成結果容易停滯在局部最優解，因此選用 0.001 作為最終學習率。

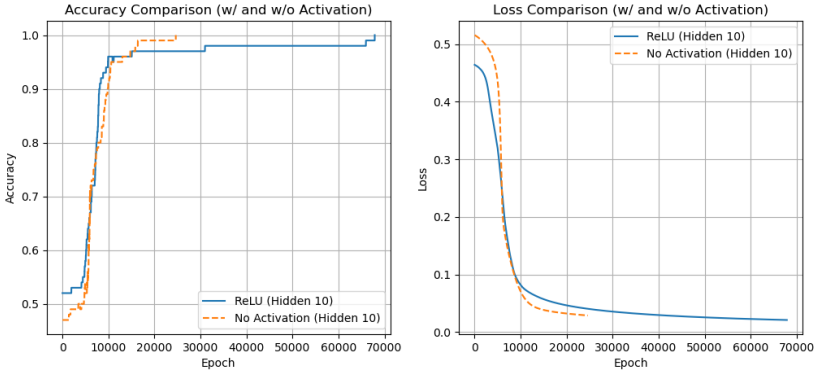


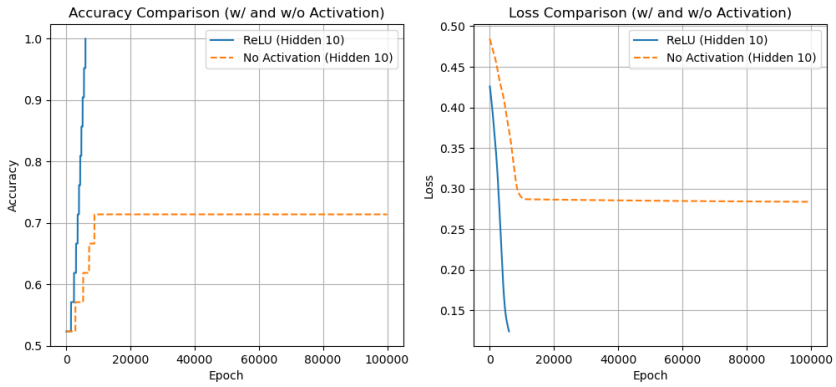

B. Try different numbers of hidden units

嘗試不同後大部分在迭代次數夠多之後都可以收斂到正確率，差別在於 loss 是否有穩定下降，以結果來看選擇 10~20 之間是較好的選擇。




C. Try without activation functions

| Linear : 在 linear data 上，有沒有加 activation function 差異不大 |
| --- |
|  |

| XOR：在 non linear data 上，沒有加 activation function 無法收斂 |
|---|
|  |

## 5. Questions

A. What is the purpose of activation functions? (6%)

activation function 類似於人類大腦的激活電位，決定了訊號是否要傳遞到下一個神經元。在這裡的 activation function 選用 sigmoid 或是 ReLU 這類非線性的轉換除了上述的功能外，也讓訓練中可以輸出非線性的組合，不然原始沒有 activation function 下矩陣相乘的更新只能組合出線性結果，能夠解決更複雜的問題。

B. What might happen if the learning rate is too large or too small? (7%)

| | |
|---|---|
| 學習率過大 | 雖然從上面的實驗結果看不出來，但學習率太大，權重更新幅度會過大，可能導致梯度下降時在損失函數的全局最優解中擺盪，不斷跳做最優解的位置，甚至會使結果發散。模型可能無法收斂，訓練過程不穩定 |
| 學習率過小 | 步伐太小更新權重速度太慢會導致模型收斂速度非常慢，甚至可能停滯在局部極小值或平坦區域，失去有效的動量，訓練效率低下 |

C. What is the purpose of weights and biases in a neural network? (7%)

| | |
|---|---|
| weight | 決定了每個 input feature 的重要性比重，可以將 input 映射到不同的 feature space。權重的大小和方向決定模型更注重什麼特徵，以梯度下降這個優化方式來說，預測輸出 weight 會和 input 相乘，因此 weight 大大影響訓練結果 |
| bias | 讓預設的輸出可以透過 bias 平移，增加模型訓練的靈活性，這樣每個神經元也不會強制一定要通過原點。同時也避免了如果輸入很小，activation function 會沒辦法激活的問題。 |

## 6. Extra

A. Implement different optimizers. (2%)

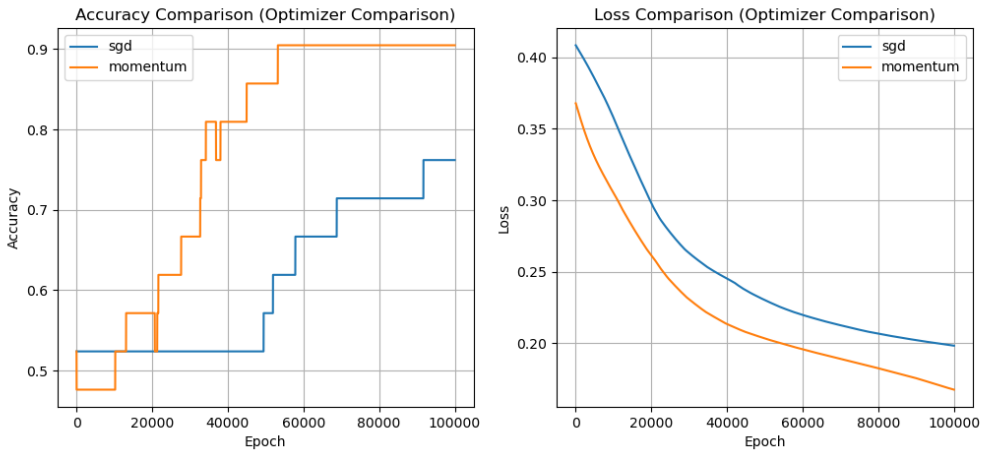
這邊嘗試了改進版本的梯度下降優化法，momentum，引入動量概念，使權重更新時考慮「過去的梯度影響」，讓更新方向更加平滑、穩定，其公式如下

$$V_t = \beta_{v_t-1} + (1-\beta)\nabla L$$
$$W = W - lr \cdot V_t$$

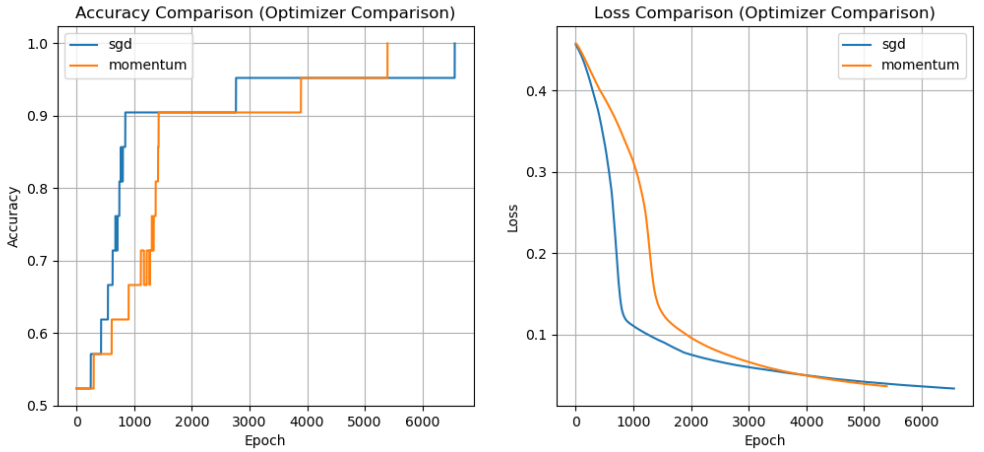
```
elif self.optimizer == 'momentum':
    self.v_W1 = self.beta * self.v_W1 + (1 - self.beta) * dW1
    self.v_b1 = self.beta * self.v_b1 + (1 - self.beta) * db1
    self.v_W2 = self.beta * self.v_W2 + (1 - self.beta) * dW2
    self.v_b2 = self.beta * self.v_b2 + (1 - self.beta) * db2
    self.v_W3 = self.beta * self.v_W3 + (1 - self.beta) * dW3
    self.v_b3 = self.beta * self.v_b3 + (1 - self.beta) * db3

    self.W1 -= learning_rate * self.v_W1
    self.b1 -= learning_rate * self.v_b1
    self.W2 -= learning_rate * self.v_W2
    self.b2 -= learning_rate * self.v_b2
    self.W3 -= learning_rate * self.v_W3
    self.b3 -= learning_rate * self.v_b3
```

XOR data (lr = 0.001/relu/$\beta$ =0.9)



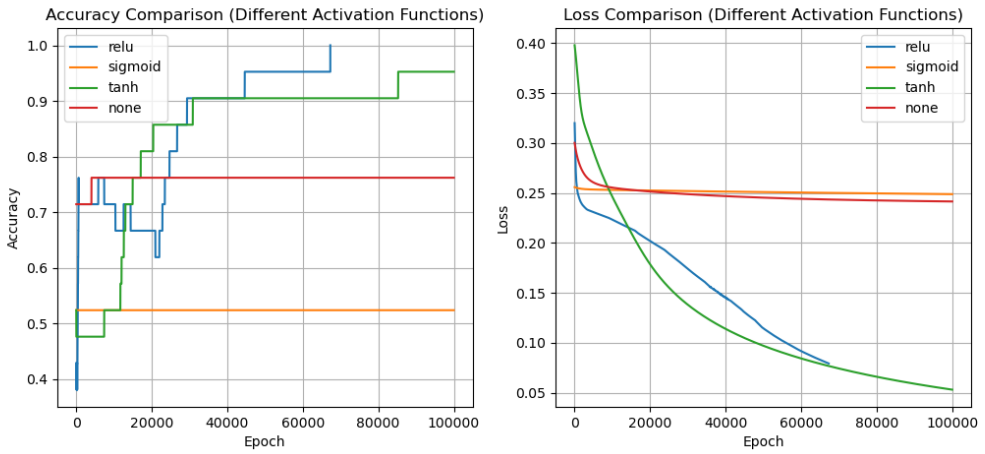

XOR data (lr = 0.01/relu/$\beta$ =0.9)



從實驗結果可以觀察到，在學習率較小的情況下，momentum 確實能夠加速收斂，使模型更快達到較高的準確率。同時，在較複雜或困難的任務（如損失函數存在多個局部最優解，或梯度變化過於平緩的區段）中，momentum 透過累積動量，可以幫助模型突破局部最優解，並保持較為穩定的更新方向。在學習率足夠大的情況下，momentum 雖然仍能加速收斂，但對最終的準確率提升有限，因為主要影響的是訓練的速度，而非模型的最終能力。因此，momentum 的優勢主要體現在加速收斂和改善梯度更新的穩定性，而非顯著提高最終準確率

B. Implement different activation functions. (3%)

嘗試不同的 hidden layer unit activation function 在 XOR 的問題上，發現若 sigmoid 選用作為中間無法很好的ㄏ收斂，推測因為 Sigmoid 的輸出範圍是 (0,1) 之間，導致梯度變得很小，影響學習效率，最後導致無法收斂且降低損失函數。

| XOR data |
| --- |
|  |