Lab 01 Backpropagation

邱以中

311551040

1. Introduction

這次的作業是要實作 neural network，經由前向 forward 的過程得到答案，再計算與真實值之間的 loss，之後經過 backpropagation 更新所有權重的參數，在不斷更新的過程當中，讓預測出來的結果越接近真實值越好。除此之外，我們還會去調整 neural network 的 learning rate、hidden units、activation function，看他們對神經網路的影響。

2. Experiment setups

A. Sigmoid functions

```python
    def activate(self,x):
        if self.activate_function=="sigmoid":
            return 1.0/(1.0+np.exp(-x))
    def derivative_activate(self,x):
        if self.activate_function=="sigmoid":
            return np.multiply(x,1.0-x)

```

附圖為我實作的 sigmoid function 與 derivative sigmoid function，sigmoid 用在 forward 過程中，derivative sigmoid 用在 backpropagation 上。

B. Neural networks

```python
class MLP():
    def __init__(self,hidden_layer=2,hidden_size=100,activate_funtion="sigmoid"):
        self.hidden_layer = hidden_layer
        self.hidden_size = hidden_size
        self.input_size = 2
        self.output_size = 1
        self.activate_function = activate_funtion

        self.weight = {}
        self.out = {}
        self.gradient = {}
        self.build()

    def build(self):

        for i in range(self.hidden_layer+1):
            if i==0:
                self.weight[f'B{i+1}'] = np.zeros((1,self.hidden_size))
                self.weight[f'W{i+1}'] = np.random.randn(self.input_size,self.hidden_size)
            elif i==self.hidden_layer:
                self.weight[f'B{i+1}'] = np.zeros((1,self.output_size))
                self.weight[f'W{i+1}'] = np.random.randn(self.hidden_size,self.output_size)
            else:
                self.weight[f'B{i+1}'] = np.zeros((1,self.hidden_size))
                self.weight[f'W{i+1}'] = np.random.randn(self.hidden_size,self.hidden_size)

```

```python
    def forward(self,x):

        self.out[f'{0}']=x
        for i in range (self.hidden_layer+1):
            x = np.matmul(x,self.weight[f'W{i+1}'])
            x = x + self.weight[f'B{i+1}']
            x = self.activate(x)
            self.out[f'{i+1}']=x

        return x

```

附圖為我實作的 neural network，我將它包成一個 MLP class，在 initial function 建立網路，並初始化 weight 權重為 gaussian noise、bias 為 0，並在 forward function 進行前向傳遞。

C. Backpropagation

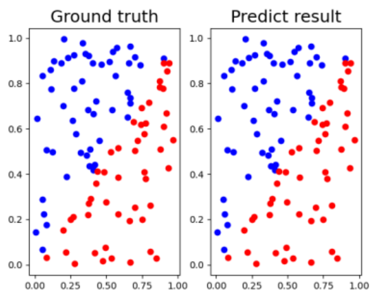
```python
    def backward(self,predict,y):
        dL = predict-y
        for i in range(self.hidden_layer+1,0,-1):
            dL = self.derivative_activate(self.out[f'{i}'])*dL
            self.gradient[f'B{i}'] = np.sum(dL,axis=0)
            self.gradient[f'W{i}'] = np.matmul(self.out[f'{i-1}'].T,dL)
            dL = np.matmul(dL,self.weight[f'W{i}'].T)

    def update_weight(self,lr = 0.001):
        for i in range(1,self.hidden_layer+2,1):
            self.weight[f'W{i}'] = self.weight[f'W{i}'] - lr*self.gradient[f'W{i}']
            self.weight[f'B{i}'] = self.weight[f'B{i}'] - lr*self.gradient[f'B{i}']
```

附圖為實作的 backpropagation，會在 backward function 去計算每個權重的 gradient，並在 update weight function 根據不同的 learning rate 去更新權重。

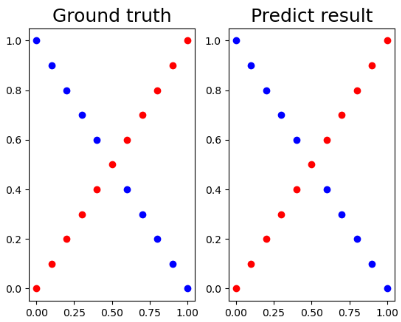
3. Results of testing

A. Screenshot and comparison figure

- Linear result



- Xor result



B. Show the accuracy of your prediction

- Linear

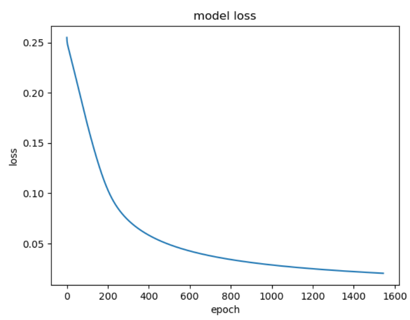
```
[1.24161431e-01]
[9.75394793e-01]
[9.97907927e-01]
[1.34542650e-02]
[9.99375897e-01]
[9.78506921e-01]
[1.24141083e-03]
[1.87855745e-03]
[7.59492261e-02]]

accuracy: 100.0
```

- Xor

```
[0.89106861]
[0.1291556 ]
[0.97650699]
[0.08029898]
[0.98904431]
[0.05046952]
[0.99172816]]

accuracy: 100.0
```

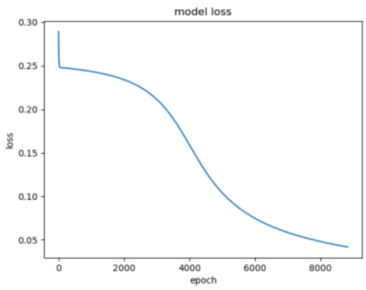
C. Learning curve (loss, epoch curve)

- Linear



```
epoch 1200: loss = 0.024919126598239208 , accuracy = 99.0 %
epoch 1300: loss = 0.023407966703550023 , accuracy = 99.0 %
epoch 1400: loss = 0.02208249362082566 , accuracy = 99.0 %
epoch 1500: loss = 0.020907476910294934 , accuracy = 99.0 %
epoch 1545: loss = 0.020420501452618183 , accuracy = 100.0 %
```

- Xor

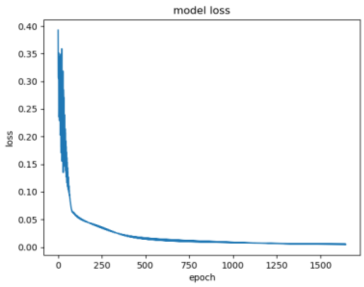


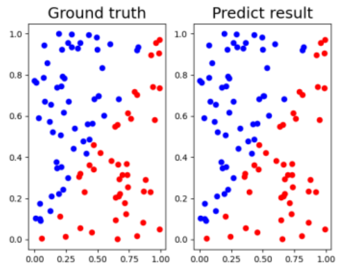

```
epoch 8500: loss = 0.0443091674403539 , accuracy = 95.23809523809524 %
epoch 8600: loss = 0.043583049312918995 , accuracy = 95.23809523809524 %
epoch 8700: loss = 0.04287650519807276 , accuracy = 95.23809523809524 %
epoch 8800: loss = 0.042188381756767875 , accuracy = 95.23809523809524 %
epoch 8841: loss = 0.04191132401621384 , accuracy = 100.0 %
```

4. Discussion

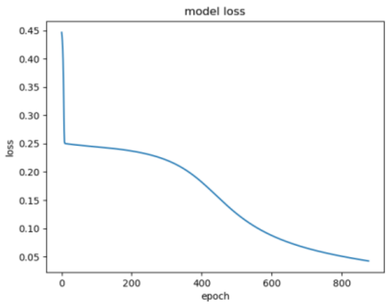
A. Try different learning rates

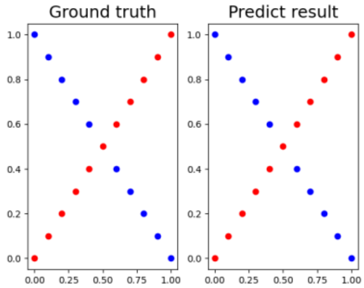
- Learning rate = 0.1
  - ✧ Linear



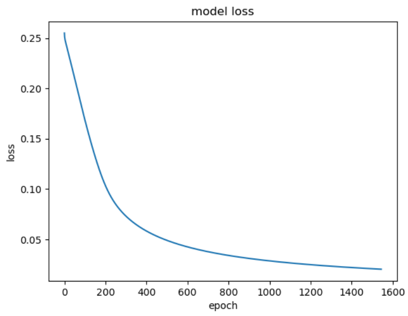


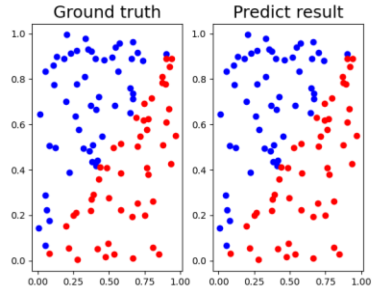
  - ✧ Xor



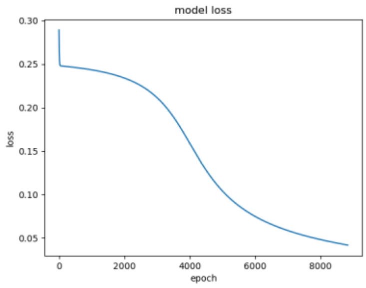


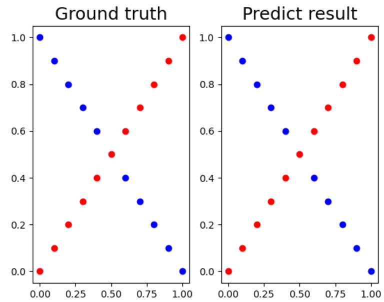
- Learning rate = 0.01
  - ✧ Linear



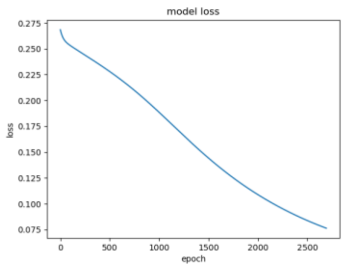


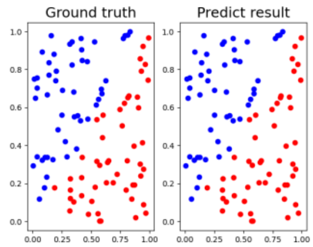
  - ✧ Xor



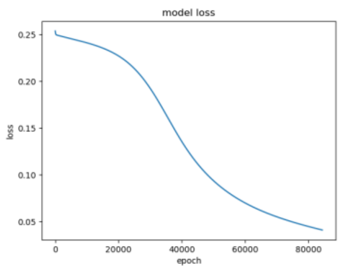


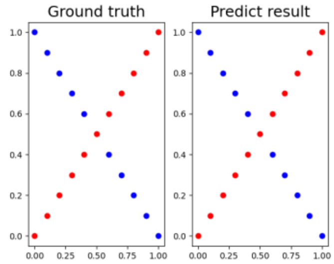
- Learning rate = 0.001
  - ✧ Linear





  - ✧ Xor

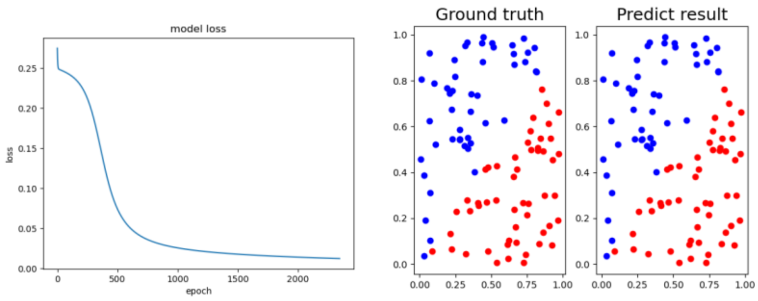




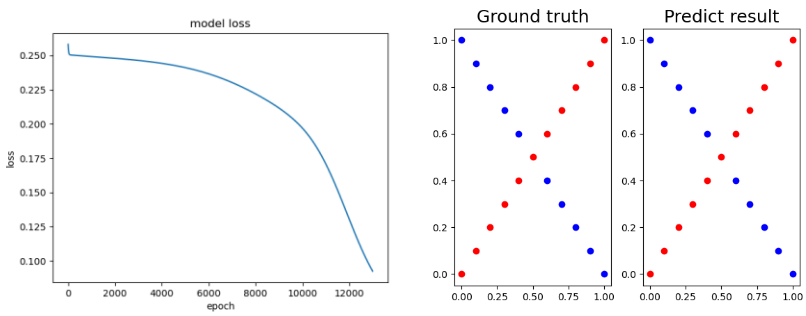
從以上的圖來看，可以發現在其他條件不變的情況下，若是我們去調整 learning rate，lr 越大 loss 的震盪也會變大，讓 loss 沒辦法很穩定的收斂，相反的 lr 越小 loss 的變化相對平滑，但是因為 lr 太小所以可能就會需要更長的時間來進行收斂。

B. Try different numbers of hidden units

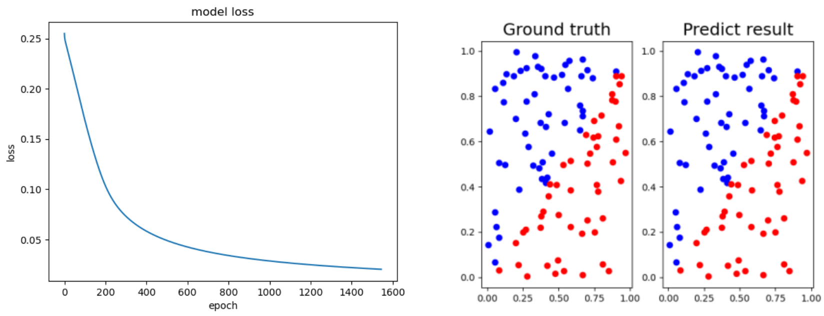
- Units = 5
  - ✧ Linear



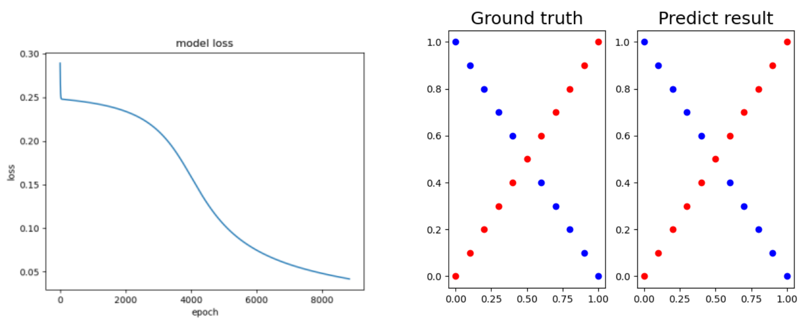
  - ✧ Xor



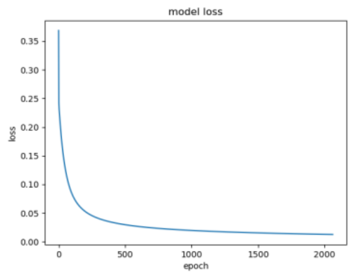
- Units = 10
  - ✧ Linear

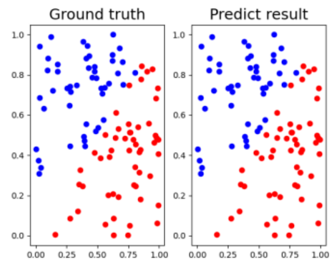


  - ✧ Xor

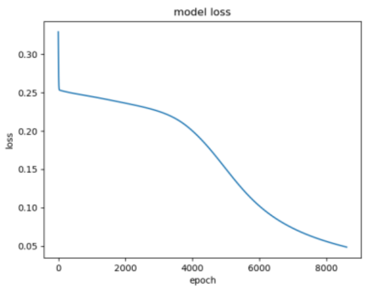


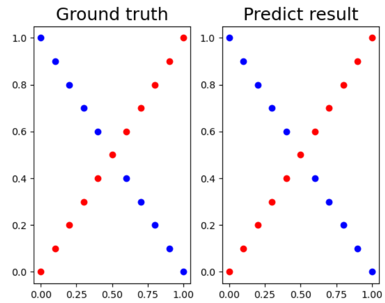
- Units = 20
  - ✧ Linear





✧ Xor

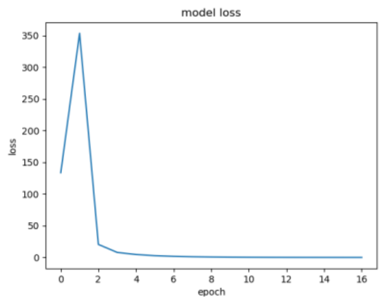


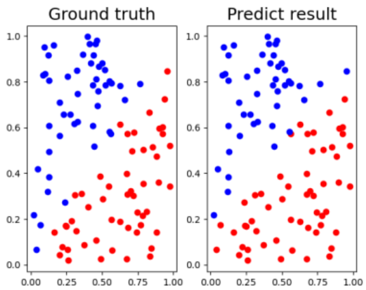


可以發現隨著 hidden unit 的增加，收斂的時間有略為縮短，推測可能的原因是 model 表徵能力更強，能夠更好的去擬合目標的數據，或是受到不同初始權重誤差的影響。

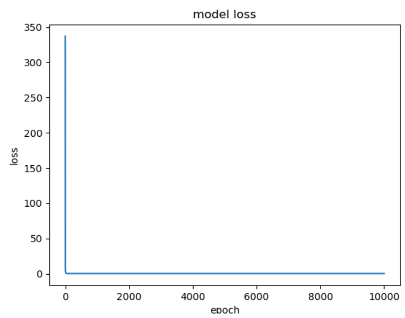
C. Try without activation functions

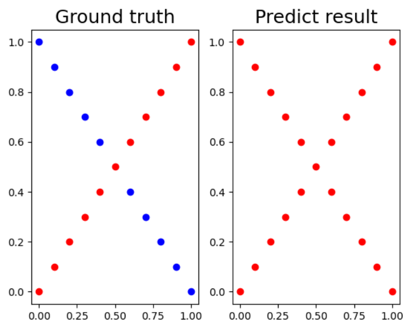
● Linear





● Xor





可以發現，如果沒有加 activation function，model 就不預測不出 nonlinear 的結果，所以在 Xor 這個 data 上就會全部預測出一樣的結果。

D. Anything you want to share

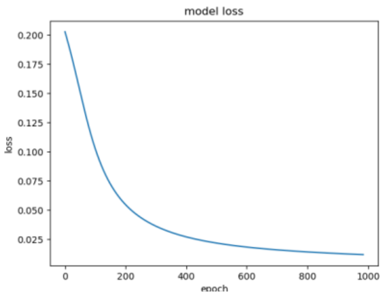
在實驗的過程中，我發現 initial weight 對訓練影響很大，一個好的 initial weight 可以讓 loss 快速的收斂，相反的如果 initial weight 選的不好，可能就會讓 loss 收斂的非常慢。而因為我的 weight 初始為隨機的 gaussian noise，所以在每次訓練時的結果也都會略有不同。

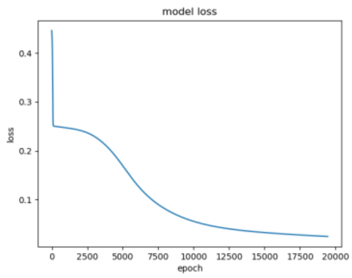
5. Extra

A. Implement different activation functions

在這裡我有嘗試將中間兩層 hidden layer 的 activation function 更改成 relu，下圖左邊為 linear 的 loss curve，右邊為 Xor 的 loss curve。
可以發現在 linear 跟 xor 的 task 中，使用 relu 都能夠比較快達到收斂，原因可能是因為 sigmoid 的倒函數在頭尾兩端會趨近於 0，導致 gradient 消失，而 relu 則不會有這個問題。

- Sigmoid





- Relu