Deep Learning and Practice Lab 1

學號: 311605015 姓名:張哲源

1. Introduction:

本次 lab 要實作雙隱藏層類神經網路，並不能使用相關神經網路套件，利用 forward propagation 得到輸出並預測答案，並計算他與 ground truth 之 loss 並透過 backpropagation 演算法更新權重，並透過調整不同 learning_rate 及改變不同隱藏層之神經元去探討對學習過程及準確度的影響

2. Experiment setups

A. **Sigmoid function**:

sigmoid 的公式為 $\sigma(s)=1/(1+e^s)$，而其微分為$-(1+e^{-s})^{-2} * -e^{-s}$，可以簡化成 $\sigma(x)(1-\sigma(x))$，而 sigmoidfunction 在 forward propagation 時能縮小數據保證數據幅度不會有太大問題，但也會有容易出現梯度消失及運算耗時較久等問題。

```python
def sigmoid(self,s,derive=False):
    if (derive==True):
        return s*(1-s)
    return 1/(1+np.exp(-s))
```

B. Neural network

流程為:

(1)初始化神經網路所有權重

(2)將資料由 input layer 往 output layer 向前傳遞(forward pass)， 並計算出所有神經元的 output

(3)誤差由 output layer 往 input layer 向後傳遞(backward propagation)，並算出每個神經元對誤差的影響

(4)用誤差影響去更新權重(weights)

(5)重複步驟(2)~(4)直到誤差收斂夠小

初始化神經網路的基本特性如 input 層之神經元數，各個 Hidden layer 的神經元及 learning_rate 等，並將所需要的 weight 矩陣透過高斯分布平均值為 0 標準差為 1，而我們預設 hidden layer1 及 hidden layer2 其 neurons 數都為 2,learning rate 則是 0.01

```python
class NeuralNetwork(object):
    def __init__(self,inputlayer=2,hiddenlayer1=2,hiddenlayer2=2,outputlayer=1,lr=0.01):
        #parameter
        self.inputlayer=inputlayer
        self.hiddenlayer1=hiddenlayer1
        self.hiddenlayer2=hiddenlayer2
        self.outputlayer=1
        #weight
        self.w1=np.random.normal(size=(self.inputlayer,self.hiddenlayer1))#(2*2)
        self.w2=np.random.normal(size=(self.hiddenlayer1,self.hiddenlayer2))#(2*2)
        self.w3=np.random.normal(size=(self.hiddenlayer2,self.outputlayer))#(2*1)
        self.lr=lr
```

並接著定義 forward_propagation 方法，在此我預設 batch_size 為 100，相當於將所有資料透過矩陣方式的方式丟進去，因此資料數量 100，batch_size 也是 100，輸出的 output 就是 y_pred

```python
def forward(self,x):
    self.z=np.dot(x,self.w1)#(100,2)dot(2,2)=100,2
    self.a=self.sigmoid(self.z) #through activation funtion
    self.z2=np.dot(self.a,self.w2)#(100,2)dot(2,2)=100,2
    self.a2=self.sigmoid(self.z2)
    self.z3=np.dot(self.a2,self.w3)#100,2dot(2,1)=100,1個output
    output=self.sigmoid(self.z3)
    return output
```

C. Backpropagation

接著計算 backpropagation 算法，backpropagation 會從 outputlayer 往回推，我在此使用的 loss function 為 L=(1/2)(y- yt)^2，其微分為-2*(y-yt), gradient decent 的公式為:

$Wij\ (l) \leftarrow Wij\ (l) - \eta \times \partial L\ \partial Wij\ (l)$

而整個 LOSS 對 WEIGHT 的偏為可以透過連鎖率寫成

$$\frac{\partial L(\theta)}{\partial w_1} = \frac{\partial y}{\partial w_1}\frac{\partial L(\theta)}{\partial y} = \frac{\partial x''}{\partial w_1}\frac{\partial y}{\partial x''}\frac{\partial L(\theta)}{\partial y} = \frac{\partial z}{\partial w_1}\frac{\partial x''}{\partial z}\frac{\partial y}{\partial x''}\frac{\partial L(\theta)}{\partial y}$$

$$= \frac{\partial x'}{\partial w_1}\frac{\partial z}{\partial x'}\frac{\partial x''}{\partial z}\frac{\partial y}{\partial x''}\frac{\partial L(\theta)}{\partial y}$$

而其中麻煩的點在於因為有許多權重，因此矩陣較為複雜，程式碼如下

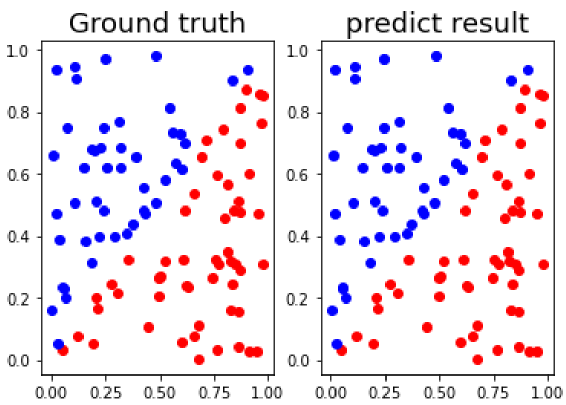
```python
        return output
    def backward(self,x,y,output):#backward propagate through the network
        self.output_error=-2*(y-output)#error in output derivative(yt-y)^2-> -2(yt-y)
        self.output_delta=self.output_error*self.sigmoid(output,derive=True)

        self.z3_error=self.output_delta.dot(self.w3.T)
        self.z3_delta=self.z3_error*self.sigmoid(self.a2,derive=True)

        self.z2_error=self.z3_error.dot(self.w2.T)
        self.z2_delta=self.z2_error*self.sigmoid(self.a,derive=True)

        self.w1-=x.T.dot(self.z2_delta)*self.lr
        self.w2-=self.a.T.dot(self.z3_delta)*self.lr
        self.w3-=self.a2.T.dot(self.output_delta)*self.lr
```

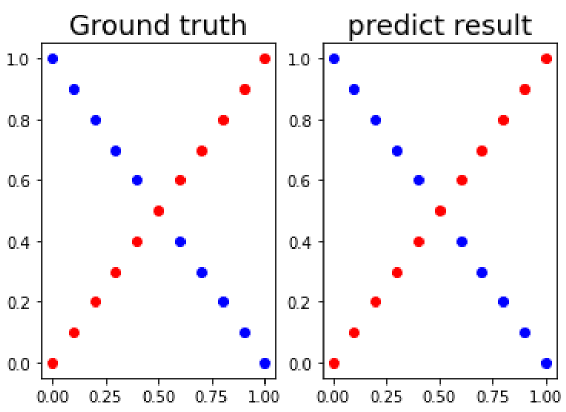
概念是由輸出一層一層往回推其 gradient 之後，再透過 gradient decent 之更新方式去更新每一個權重，在此寫法相當於 batch_size=100,一次把所有輸入丟進去並更新一次權重

3.Results of your testing

A. Screenshot and comparison figure

Linear vs xor





在比較簡單的輸入時分類分得很乾淨，預測的結果與實際的結果吻合

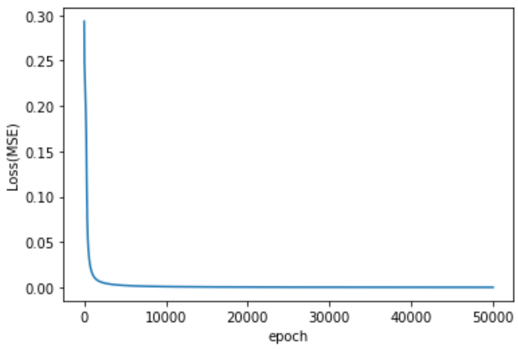
B. show the accuracy of my prediction

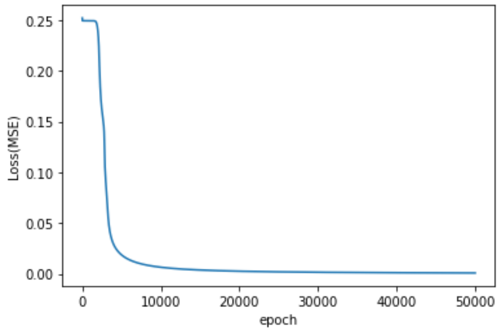
Linear	Xor
[5.32780857e-06] [9.99937909e-01] [5.23224896e-06] [1.09194135e-05] [9.99999304e-01] [9.99999834e-01] [5.31782008e-06] [9.23084712e-06] [5.51576548e-03] [2.63228464e-05] [9.99999835e-01] [5.17087098e-06] [5.16454224e-06] [5.17140858e-06] [9.99999801e-01] [9.99999834e-01]] 準確率 100.0 %	[9.59893461e-01] [1.62599659e-05] [9.72950992e-01] [1.19709030e-05] [2.06901287e-05] [9.79269236e-01] [6.99419176e-05] [9.85023509e-01] [3.59025421e-04] [9.85068130e-01] [2.16011733e-03] [9.85068528e-01] [1.23605557e-02] [9.85068532e-01]] 準確率 100.0 %

可以看出在資料量較簡單的情況下，準確率都可以到達 100%

C.Learning curve (loss, epoch curve)

Linear	XOR
epoch: 0 loss:0.293303 epoch: 5000 loss:0.002087 epoch: 10000 loss:0.000855 epoch: 15000 loss:0.000485 epoch: 20000 loss:0.000320 epoch: 25000 loss:0.000231 epoch: 30000 loss:0.000177 epoch: 35000 loss:0.000141 epoch: 40000 loss:0.000116 epoch: 45000 loss:0.000098	epoch: 0 loss:0.252164 epoch: 5000 loss:0.018669 epoch: 10000 loss:0.006510 epoch: 15000 loss:0.003770 epoch: 20000 loss:0.002611 epoch: 25000 loss:0.001982 epoch: 30000 loss:0.001590 epoch: 35000 loss:0.001325 epoch: 40000 loss:0.001133 epoch: 45000 loss:0.000989





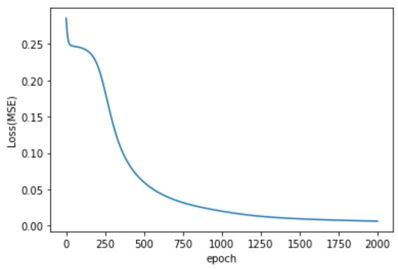
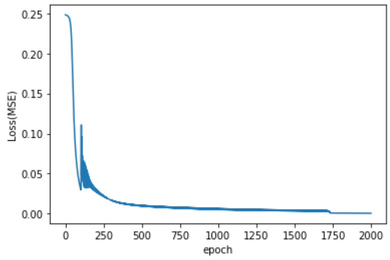
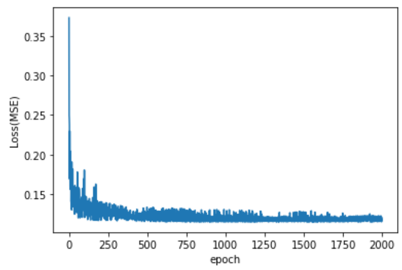
可以看出 xor 在一開始有一小段訓練不太下去，可能還停留在 local minimun，因此較晚收斂到全域最小值上，而 linear 在整體訓練的較快最後 loss 也更低。

4.Discussion

A. Try different learning rates

Linear

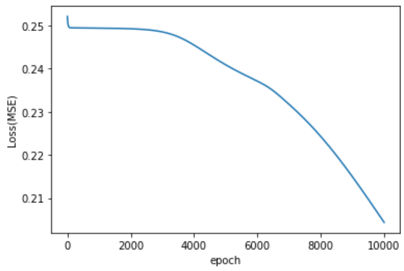
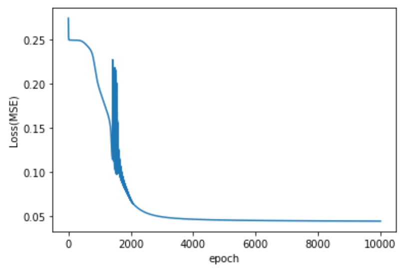
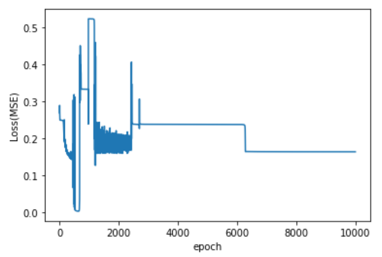
在 linear 的分類中為方便比較對我們將 epoch 縮減到 2000

Learning rate=0.01	Learning rate=0.1	Learning rate=1
epoch: 0 loss:0.285479 epoch: 200 loss:0.221262 epoch: 400 loss:0.085493 epoch: 600 loss:0.044925 epoch: 800 loss:0.028808 epoch: 1000 loss:0.020044 epoch: 1200 loss:0.014078 epoch: 1400 loss:0.010678 epoch: 1600 loss:0.008703 epoch: 1800 loss:0.007379	epoch: 0 loss:0.248296 epoch: 200 loss:0.027351 epoch: 400 loss:0.012199 epoch: 600 loss:0.009038 epoch: 800 loss:0.007448 epoch: 1000 loss:0.006191 epoch: 1200 loss:0.005217 epoch: 1400 loss:0.004530 epoch: 1600 loss:0.004043 epoch: 1800 loss:0.000616	epoch: 0 loss:0.372717 epoch: 200 loss:0.130258 epoch: 400 loss:0.121818 epoch: 600 loss:0.117788 epoch: 800 loss:0.127835 epoch: 1000 loss:0.118559 epoch: 1200 loss:0.118376 epoch: 1400 loss:0.121524 epoch: 1600 loss:0.122534 epoch: 1800 loss:0.119728
		

由圖可知在 learning rate 比較小時，學習曲線較為平滑，而 learning rate 為 0.1 的時候時因為學習率太較高因此會產生震盪的效果，learning rate 為 1 時震盪效果更加明顯，最終 loss 到 0.1 時也降不下去。

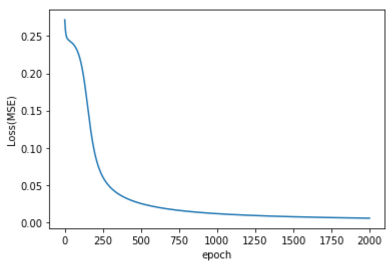
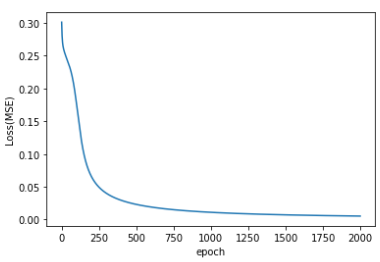
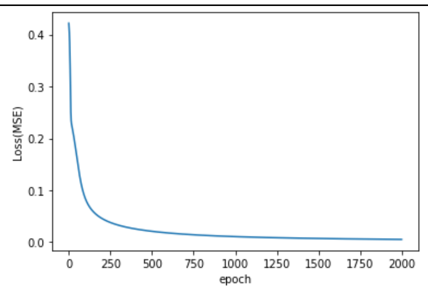
Xor

在 Xor 的分類中我將 epoch 提升自 10000 以方便進行比對

Learning rate=0.01	Learning rate=0.1	Learning rate=1
epoch: 0 loss:0.252087 epoch: 1000 loss:0.249378 epoch: 2000 loss:0.249228 epoch: 3000 loss:0.248499 epoch: 4000 loss:0.245445 epoch: 5000 loss:0.240864 epoch: 6000 loss:0.237082 epoch: 7000 loss:0.231705 epoch: 8000 loss:0.224257 epoch: 9000 loss:0.214917	epoch: 0 loss:0.274197 epoch: 1000 loss:0.193204 epoch: 2000 loss:0.067642 epoch: 3000 loss:0.048749 epoch: 4000 loss:0.046016 epoch: 5000 loss:0.045092 epoch: 6000 loss:0.044626 epoch: 7000 loss:0.044342 epoch: 8000 loss:0.044149 epoch: 9000 loss:0.044009	epoch: 0 loss:0.269120 epoch: 1000 loss:0.523673 epoch: 2000 loss:0.192988 epoch: 3000 loss:0.238166 epoch: 4000 loss:0.238094 epoch: 5000 loss:0.238066 epoch: 6000 loss:0.237922 epoch: 7000 loss:0.164068 epoch: 8000 loss:0.163917 epoch: 9000 loss:0.163851
		

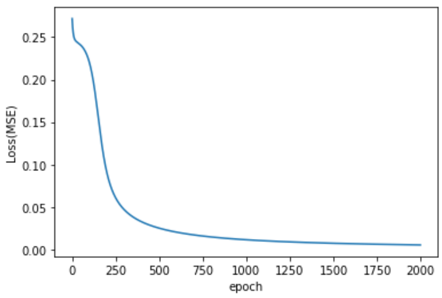
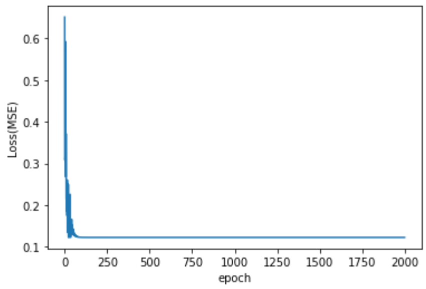
從圖可以看出 learning rate 較小時當他找到合適的梯度時他會往該方向去進行更新，而當 learning rate 較高時震盪也較明顯，當他訓練到一定程度時也會更難降下去。

B.Try different numbers of hidden units

Learning rate=0.01	Learning rate=0.1	Learning rate=1
epoch: 0 loss:0.271794 epoch: 200 loss:0.089910 epoch: 400 loss:0.032543 epoch: 600 loss:0.020412 epoch: 800 loss:0.014749 epoch: 1000 loss:0.011486 epoch: 1200 loss:0.009388 epoch: 1400 loss:0.007933 epoch: 1600 loss:0.006867 epoch: 1800 loss:0.006050	epoch: 0 loss:0.301247 epoch: 200 loss:0.065332 epoch: 400 loss:0.029284 epoch: 600 loss:0.019073 epoch: 800 loss:0.014034 epoch: 1000 loss:0.011053 epoch: 1200 loss:0.009102 epoch: 1400 loss:0.007733 epoch: 1600 loss:0.006718 epoch: 1800 loss:0.005935	epoch: 0 loss:0.421833 epoch: 200 loss:0.045450 epoch: 400 loss:0.025466 epoch: 600 loss:0.017634 epoch: 800 loss:0.013344 epoch: 1000 loss:0.010677 epoch: 1200 loss:0.008884 epoch: 1400 loss:0.007602 epoch: 1600 loss:0.006641 epoch: 1800 loss:0.005893
		

可以看出 epochs 為 2000 時，其最終的 Loss 並不會差太多，
hidden unit 越大其下降的速率也稍微更快一點，可能需要透過較為複雜的輸入來探討隱藏層對 loss 的影響較為方便看出影響結果。

C. Try without activation functions

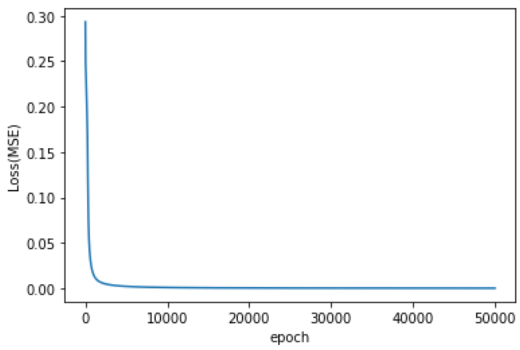
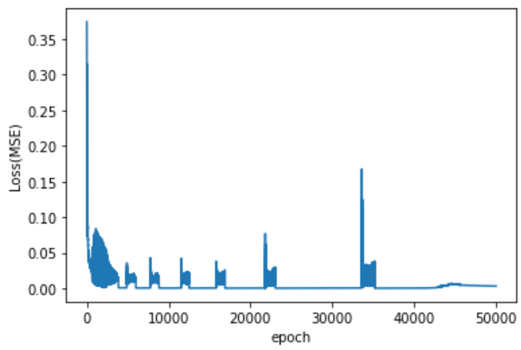
<table>
<tr><th>Activation=sigmoid</th><th>No activation function</th></tr>
<tr><td>epoch: 0 loss:0.271794
epoch: 200 loss:0.089910
epoch: 400 loss:0.032543
epoch: 600 loss:0.020412
epoch: 800 loss:0.014749
epoch: 1000 loss:0.011486
epoch: 1200 loss:0.009388
epoch: 1400 loss:0.007933
epoch: 1600 loss:0.006867
epoch: 1800 loss:0.006050</td><td>epoch: 0 loss:0.650870
epoch: 200 loss:0.122511
epoch: 400 loss:0.122511
epoch: 600 loss:0.122511
epoch: 800 loss:0.122511
epoch: 1000 loss:0.122511
epoch: 1200 loss:0.122511
epoch: 1400 loss:0.122511
epoch: 1600 loss:0.122511
epoch: 1800 loss:0.122511</td></tr>
<tr><td>
</td><td>
</td></tr>
</table>

可以看出在沒有 activation function 的情況下容易產生梯度消失的問題，當 loss 到 0.12 附近時梯度已經幾乎消失。

5.Extra

B.Implement different activation functions. (3%)

我接下來使用 tanh 去進行測試比對，其 LOSS 與分類結果如下

sigmoid	tanh
epoch: 0 loss:0.293303 epoch: 5000 loss:0.002087 epoch: 10000 loss:0.000855 epoch: 15000 loss:0.000485 epoch: 20000 loss:0.000320 epoch: 25000 loss:0.000231 epoch: 30000 loss:0.000177 epoch: 35000 loss:0.000141 epoch: 40000 loss:0.000116 epoch: 45000 loss:0.000098	epoch: 0 loss:0.374289 epoch: 5000 loss:0.007494 epoch: 10000 loss:0.000215 epoch: 15000 loss:0.000169 epoch: 20000 loss:0.000125 epoch: 25000 loss:0.000083 epoch: 30000 loss:0.000125 epoch: 35000 loss:0.015689 epoch: 40000 loss:0.000116 epoch: 45000 loss:0.005522
	
準確率:100%	準確率:100%

可以看出在資料量不大的時候，兩者皆能很好的分類資料，但當 activation 為 sigmoid 時的 losss 下降效果更為平滑，tanh 產生較大的震盪最終 loss 也較高。

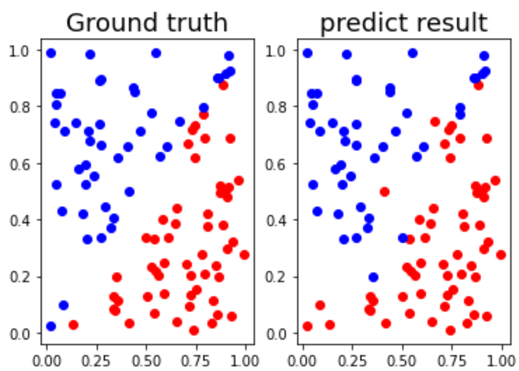
C.Implement convolutional layers. (5%)

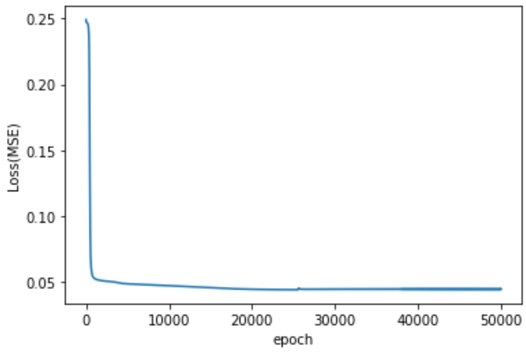
透過 convolution 的定義，我決定先建照一個 2*1 的遮罩，Stride 及 padding 設定為 1，每個 x 的輸出都會受到前一個 x 值的影響，其程式如下

```
maskh=np.array([0.2,0.8])
xnew=np.zeros((100,2))

for i in range(100):
    if (i==0 or i==99):
        xnew[i][0]=x1[i][0]
        xnew[i][1]=x1[i][1]

    else:
        xnew[i][0]=x1[i-1][0]*maskh[0]+x1[i][0]*maskh[1]
        xnew[i][1]=x1[i-1][1]*maskh[0]+x1[i][1]*maskh[1]
```

而由下面圖表可以看出，在資料量較小的情況下，捲積可能無法帶來太好的協助，loss 到 0.04 附近之後變較難持續下降，最後的準確率也僅有 93%。

```
epoch: 0 loss:0.249219
epoch: 5000 loss:0.048224
epoch: 10000 loss:0.046843
epoch: 15000 loss:0.045225
epoch: 20000 loss:0.044091
epoch: 25000 loss:0.043719
epoch: 30000 loss:0.044145
epoch: 35000 loss:0.044184
epoch: 40000 loss:0.044180
epoch: 45000 loss:0.044156
```





準確率 93.0 %