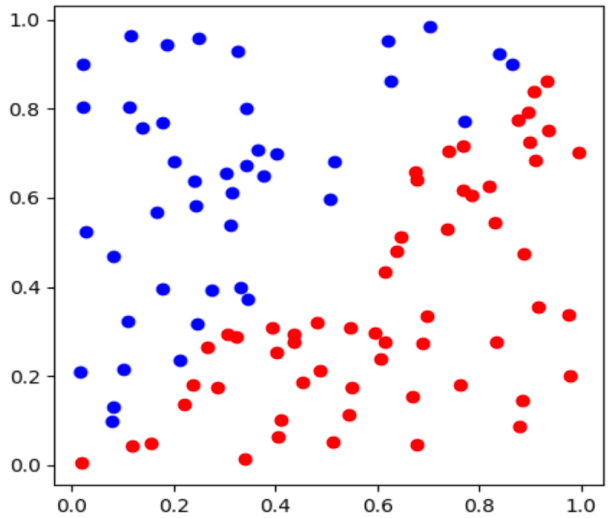
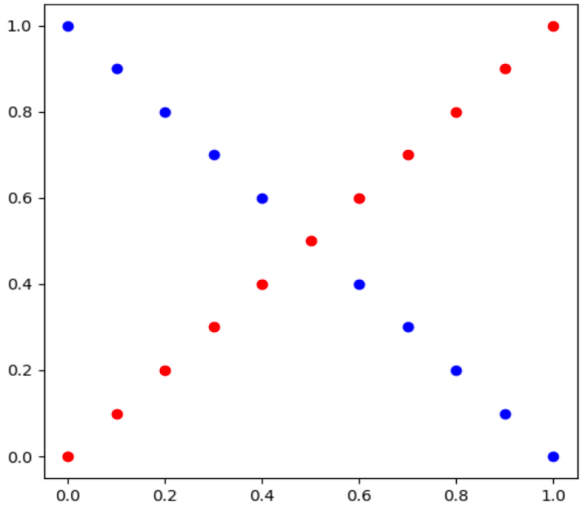
1 Introduction

這次的 LAB 主要目標是要建構一個擁有兩層 Hidden Layer 的全連接層神經網路(Fully connected Neural Network)，分別給予線性(Linear)以及互斥或(XOR)資料集作為訊練，而目標則是二維平面上的紅藍兩類，接著在訓練階段時，我們要能夠區分輸入進來的二維資料點在二維平面上是屬於哪一類別。

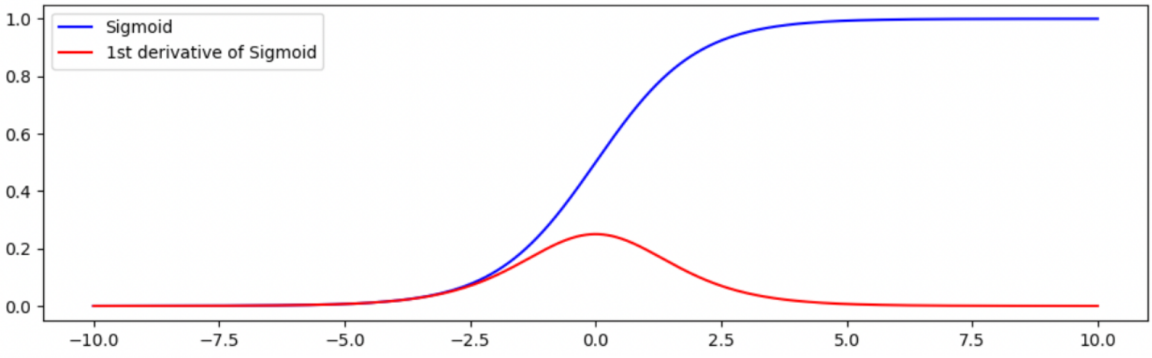




2 Experiment setups

2.1 Sigmoid function

激活函數之一，因為有了激活函數 Neural Network 才能克服 XOR 這種資料集的分類問題，也因為激活函數在數學上有很好的連續性質，因此激活函數可以說是 Neural Network 相當重要的推手。
底下的 Sigmoid function 是這次 LAB 會頻繁使用到的函數，也因為 Backpropagation 會需要 Sigmoid 的微分，因次也順便把微分後的樣子放上來。



$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad \frac{d\sigma(x)}{d(x)} = \sigma(x)\cdot(1-\sigma(x)).$$

2.2 Neural Network

兩層的 Hidden Layer 是使用 10 個 neuron，Learning rate 設置為 0.1
並且有三個權重矩陣(Weight Matrix)，分別為 w1(2*10), w2(10*10), w3(10*1)

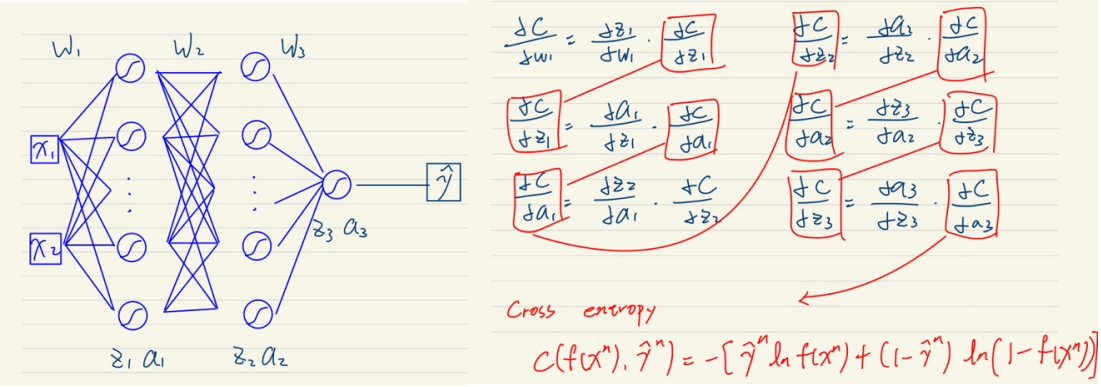
並且在 Forward pass 的時候把各個 neuron 的 input(z1~z3)以及 output(a1~a3)記錄起來。以便於 Backward pass 計算 Gradient。

```
for i, x in enumerate(inputs):
    a0 = np.array(x).reshape((1,2)) #要記得都轉換成numpy的型態
    y = np.array(labels[i]).reshape((1,1))
    #forward pass
    z1 = a0 @ w1
    a1 = sigmoid(z1)
    z2 = a1 @ w2
    a2 = sigmoid(z2)
    z3 = a2 @ w3
    a3 = sigmoid(z3)
    predict = a3
```

2.3 Backpropagation

```
initialize network weights (often small random values)
do
   forEach training example named ex
      prediction = neural-net-output(network, ex)  // forward pass
      actual = teacher-output(ex)
      compute error (prediction - actual) at the output units
      compute Δw_h for all weights from hidden layer to output layer  // backward pass
      compute Δw_i for all weights from input layer to hidden layer   // backward pass continued
      update network weights // input layer not modified by error estimate
until all examples classified correctly or another stopping criterion satisfied
return the network
```

根據上面的演算法流程，我們最重要的事情就是每一輪 input 送進來得到一個 output 時，我們就可以根據這筆 input 得到的 Loss 再利用 Backpropagation 去跟新權重，根據 Chain rule 把我們想求的 w 對整個 Loss 的微分展開，得到下面我手寫的算式。
而在程式碼當中我也用相同的變數名稱去表示，以利助教 trace code。
例如 C_z3 代表的就是 Loss function “C”對 z3 進行偏微分。

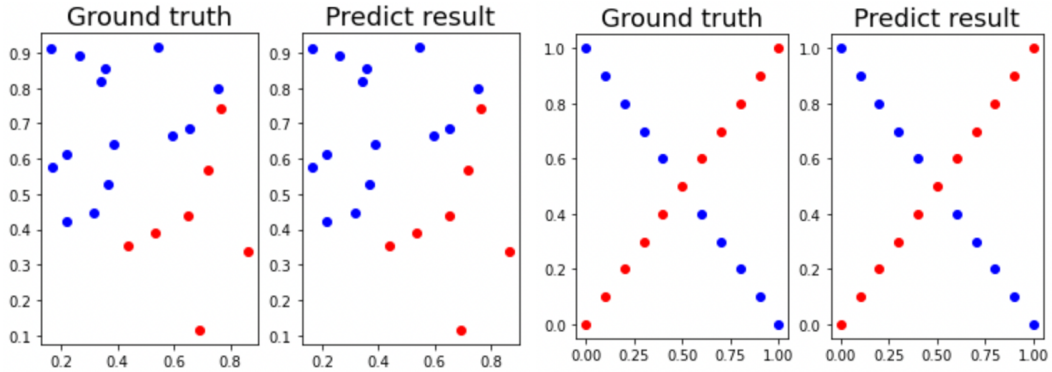


```
#backward pass
C_z3 = derivative_sigmoid(a3) * -(y/(predict)-(1-y)/(1-predict))
C_a2 = (C_z3 @ w3.T)
w3_gradient = (a2.T @ C_z3)
w3 = w3 - learning_rate * w3_gradient

C_z2 = derivative_sigmoid(a2) * C_a2
C_a1 = (C_z2 @ w2.T)
w2_gradient = (a1.T @ C_z2)
w2 = w2 - learning_rate * w2_gradient

C_z1 = derivative_sigmoid(a1) * C_a1
C_a0 =  (C_z1 @ w1.T)
w1_gradient = (a0.T @ C_z1) #Gradient是要乘input,而不是weight
w1 = w1 - learning_rate * w1_gradient
```

3 Results of testing

3.1 Screenshot and comparison figure

基本上 1000 個 epoch 過後都可以達到 100%的分類正確率



3.2 Show the accuracy of prediction

Linear Dataset Accuracy: 100%

XOR Dataset Accuracy: 100%

左邊是每 100 個 epoch 顯示一次當前的 loss 是多少，右邊為 21 筆測試資料的預測結果。

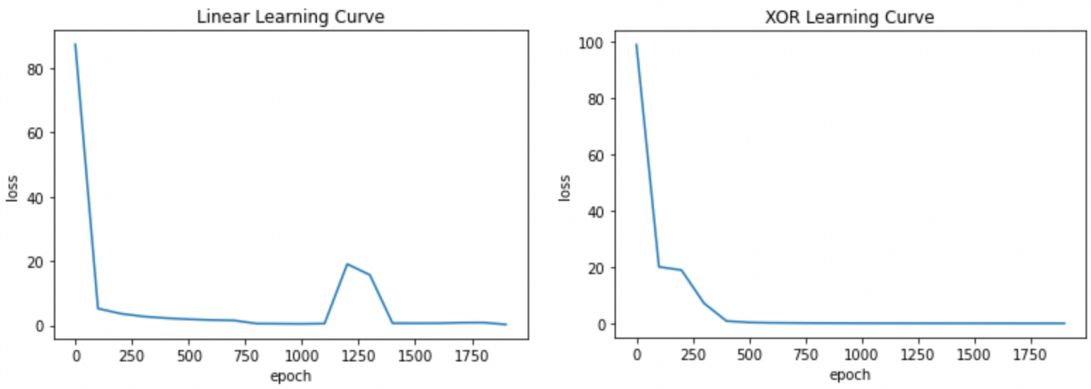
Loss per 100 epoch:

```
epoch:0 total_loss:[[80.60676374]]
epoch:100 total_loss:[[5.96354269]]
epoch:200 total_loss:[[4.31429373]]
epoch:300 total_loss:[[3.48402145]]
epoch:400 total_loss:[[3.04816546]]
epoch:500 total_loss:[[2.75561226]]
epoch:600 total_loss:[[2.52558023]]
epoch:700 total_loss:[[2.32289956]]
epoch:800 total_loss:[[2.12797821]]
epoch:900 total_loss:[[1.93076033]]
epoch:1000 total_loss:[[1.73378577]]
epoch:1100 total_loss:[[1.55317875]]
epoch:1200 total_loss:[[1.25993329]]
epoch:1300 total_loss:[[1.55060379]]
epoch:1400 total_loss:[[0.65877306]]
epoch:1500 total_loss:[[0.87047663]]
epoch:1600 total_loss:[[0.82118685]]
epoch:1700 total_loss:[[3.17579082]]
epoch:1800 total_loss:[[0.46886722]]
epoch:1900 total_loss:[[0.21990438]]
```

Testing predict result:

```
[[3.84144865e-07]]
[[0.99999673]]
[[0.9999992]]
[[0.82820547]]
[[3.75996173e-07]]
[[0.99999913]]
[[3.73301415e-07]]
[[0.99088845]]
[[3.83258453e-07]]
[[3.69359904e-07]]
[[0.99999915]]
[[0.99999922]]
[[3.86604772e-07]]
[[0.99999911]]
[[4.37437536e-07]]
[[0.99300962]]
[[0.00014587]]
[[4.60783116e-07]]
[[0.99999919]]
[[4.1258118e-07]]
[[3.86836565e-07]]
```

3.3 Learning curve



3.4 Anything wants to present

可能因為這次的分類任務比較簡單，所以不需要增加 Bias 的參數也可以訓練得很好，甚至不需要像我一樣設定成 2000 個 epoch，應該 1500 左右的 epoch 就可以得到 100%的正確率了。

還有一點小錯誤也提一下希望自己下次不要再犯，因為我們在計算的時候其實大部分是用矩陣運算，有時候會把注意力放太多在數字的對齊上，因為我們都知道 10x1 的矩陣是不能跟 10x10 的矩陣去做運算的，但我反而忽略一個很致命的點是資料型態，舉例來說[1, 2, 3]跟[[1, 2, 3]]在運算的過程當中根據計算方式的不同會有不一樣的結果發生，所以如果有用 Enumerate 要記得再把型態轉換成 numpy array 免得像我一樣 Debug 半死。

4 Discussion

4.1 Try different learning rate

比較大的 Learning rate 會讓整體 loss 下降速度快一些，但其實整體幅度差距沒有很大，但如果不小心調太大的話，可能會在 Learning curve 上看到起起落落的曲線，我想原因應該是在 Gradient 調整的過程中不小心一步走太遠，導致不小心走上比較陡峭的山坡，但或許是因為這次的分類任務比較簡單，整體來說 Learning rate 影響並不會太嚴重。

4.2 Try different numbers of hidden units

跑過二十輪的實驗後發現，如果把每層的 units 數設定為 4 以下，即使跑了 10,000 個 epoch，XOR 的資料集都沒辦法成功分類，但當 units 數設定為 4 以上（包含 4），僅僅只需要 2,000 個 epoch 就可以成功訓練分類 XOR 資料集的神經網路。目前還在試圖找尋原因，我知道 Gradient descent 很看初始值，所以我在想會不會是我做的實驗次數不夠多，說不定 units 數在 4 以下是可以成功訓練成功的也不一定，因為就老師上課所舉的例子，僅需要兩層的 perceptron 應該就可以成功分類 XOR 資料集。

4.3 Try without activation functions

如果不用 activation function 是沒有辦法成功辨識 XOR 的資料集，因為就算我們用再多層的 hidden layer 但如果沒有加上 activation function 的話其實跟 single layer 的神經網路是一樣的，多層的矩陣運算其實可以合併成一個矩陣，而 XOR

的資料及必須要兩層以上，也可以想像成是兩條 decision boundary 才能夠正確的分類。

4.4 Anything wants to share

當我在實作 Backpropagation 在計算 Gradient 的時候不小心誤把 neuron 的輸入(z)對 weight(w)的偏微分算成 weight(w)，正確來說偏微分的結果應為 neuron 的 input(x or a)，導致我 debug 也找了非常久…
建議以後可以先把公式推導整個寫一遍，而不是看完投影片就開始埋頭苦尻。

$$\partial z/\partial w_1 =?\ x_1$$

$$\partial z/\partial w_2 =?\ x_2$$

5 Extra

將 Sigmoid function 替換成 ReLU function，但因為 ReLU 出來的值不會介於 0~1 之間，因此 Cross entropy 在計算 math.log(1-predict) 的時候需要留意（因為 log 不能取負值），因此我這邊的做法是在最後一層的時候使用 Sigmoid function 來避免預測的結果沒有介於 0~1 之間，才能使用 Cross entropy 來當作 Loss function。

```python
def relu(x):
    for i in range(x.shape[1]):
        if x[0, i] > 0:
            continue
        else:
            x[0, i] = 0
    return x
```

```python
def der_relu(x):
    for i in range(x.shape[1]):
        if x[0, i] > 0:
            x[0, i] = 1
        else:
            x[0, i] = 0
    return x
```

```python
for i, x in enumerate(inputs):
    a0 = np.array(x).reshape((1,2)) #要記得都轉換成numpy的型態
    y = np.array(labels[i]).reshape((1,1))
    #forward pass
    z1 = a0 @ w1
    a1 = relu(z1)
    z2 = a1 @ w2
    a2 = relu(z2)
    z3 = a2 @ w3
    a3 = sigmoid(z3)
    predict = a3
```

```
    #backward pass
    C_z3 = der_sigmoid(a3) * -(y/(predict)-(1-y)/(1-predict))
    C_a2 = (C_z3 @ w3.T)
    w3_gradient = (a2.T @ C_z3)
    w3 = w3 - learning_rate * w3_gradient

    C_z2 = der_relu(z2) * C_a2
    C_a1 = (C_z2 @ w2.T)
    w2_gradient = (a1.T @ C_z2)
    w2 = w2 - learning_rate * w2_gradient

    C_z1 = der_relu(z1) * C_a1
    C_a0 =  (C_z1 @ w1.T)
    w1_gradient = (a0.T @ C_z1) #Gradient是要乘input,而不是weight
    w1 = w1 - learning_rate * w1_gradient
    loss = -(y * math.log(predict) + (1-y) * math.log(1-predict))
    total_loss += loss
```