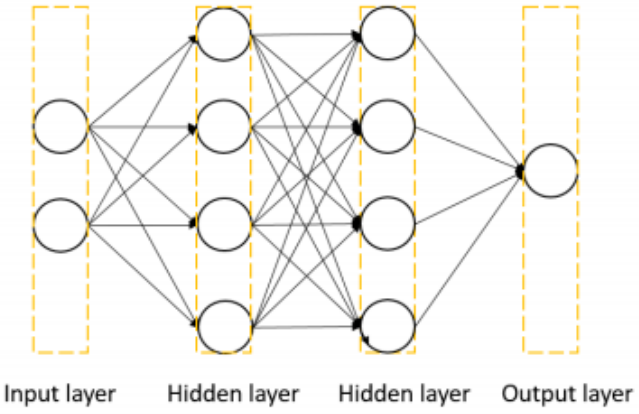
Lab1 : back-propagation

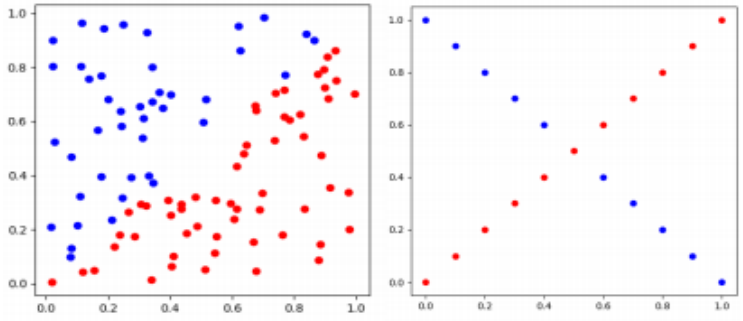
introduction

- 禁止使用pytorch 等framework，實作底下這個Nerual Network



- **Figure 1. Two layers neural network**

- 有兩種不同的DATA 座標 (x1,x2)代表 input_layer 顏色y[0,1] 代表 output layer

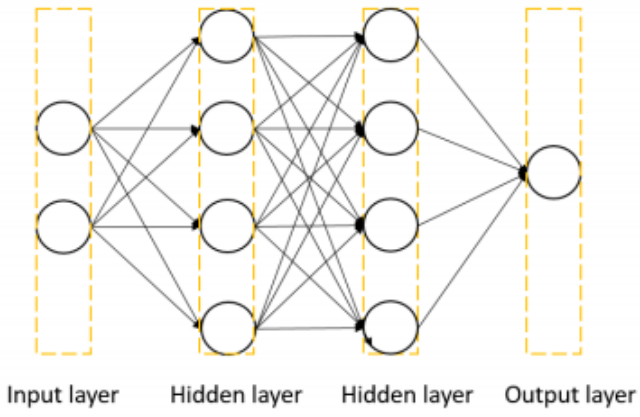


Experiment setups

1. Sigmoid functions
    - 這是 activation_function，引入了非線性
    - f(x) = 1/(1+e^-x)

```
def sigmoid(x):
        return 1.0/(1.0+np.exp(-x))
```

2. Neural network

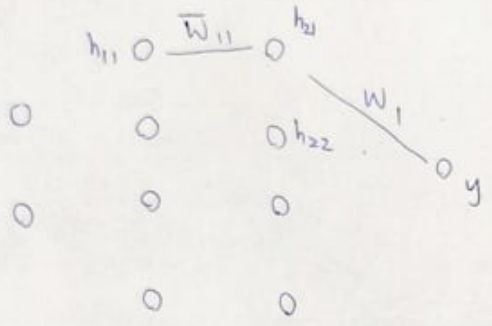


    - **Figure 1. Two layers neural network**

- 我沒有加bias或是增加hidden layer or unit. 僅按照Spec上面所畫的實作
- 簡述各層dimension
  - input layer: 2*1 as x
  - weight 1: 4*2 as w1
  - hidden layer1: 4*1 as h1
  - weight 2: 4*4 as w2
  - hidden layer2: 4*1 as h2
  - weight 3: 1*4 as w3
  - output layer: pred_y 1*1 as pred_y
  - Ue MSE as loss function as L

3. Backpropagation

- Forward Feed:
  - hidden layer1 = sigmoid(w1*x) dim: 4*1
  - hidden layer2 = sigmoid(w2*h1) dim: 4*1
  - pred_y = w3*h3 (dim: 1*1)
  - Loss = 0.5*(pred_y- y)^2
- Backpropagation:
  - 需要計算w1,w2,w3,h1,h2分別對 Loss 的影響力
  - 一開始我不清楚要怎麼整個向量做 chain rule 所以我把weight matrix拆成scalar 再計算偏微分



A = Ground truth

$L = 0.5\,(y - A)^2$ $\quad y = w_1 h_{21} + w_2 h_{22} + w_3 h_{23} + w_4 h_{24}$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial w_1} = (y - A) \cdot h_{21}$$

$$\boxed{\frac{\partial L}{\partial h_{21}}} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial h_{21}} = \boxed{w_1}$$

$$\frac{\partial L}{\partial w_{11}} = \boxed{\frac{\partial L}{\partial h_{21}}} \cdot \frac{\partial h_{21}}{\partial w_{h11}} = \boxed{w_1} \cdot \sigma'(h_{21}) \cdot h_{11}$$

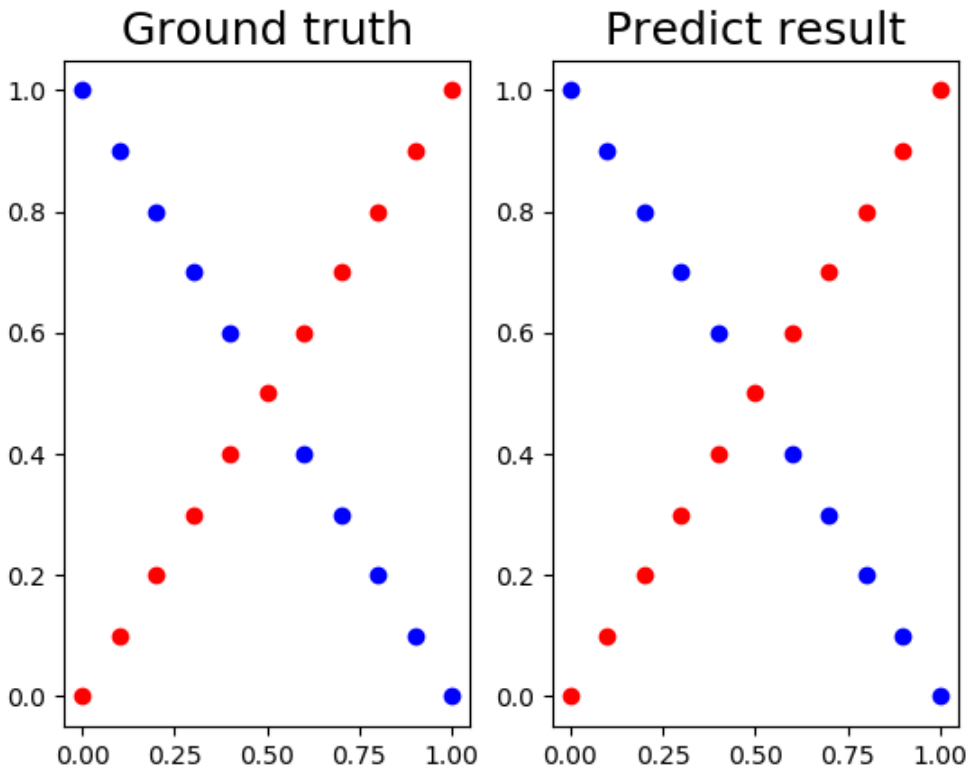
$$\frac{\partial L}{\partial h_{11}} = \boxed{\frac{\partial L}{\partial h_{21}}} \frac{\partial h_{21}}{\partial h_{11}} + \cdots + \boxed{\frac{\partial L}{\partial h_{24}}} \frac{\partial h_{24}}{\partial h_{11}}$$

- 找到規律之後就可以表達成

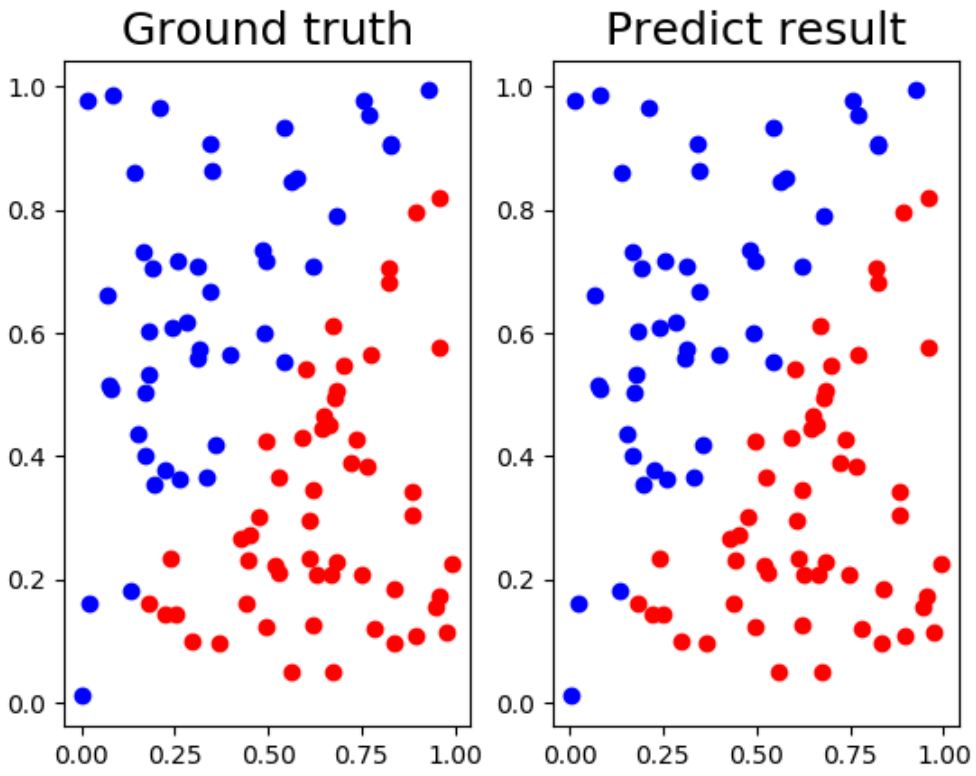
```
# ------------------------------------ #
#    output layer to hidden layer2     #
# ------------------------------------ #

D_W3 =  (2 * (pred_y-y) * H2 )
D_H2 =  (2 * (pred_y-y) * W2)

# -------------- ------------------- #
#    hidden layer2 to hidden layer1    #
# ------------------------------------ #

D_S_H2 = derivative_sigmoid(H2)
D_W2 = H1.T * (D_S_H2 *D_H2)
D_H1 = np.dot( W1 ,  (D_H2*D_S_H2) )

# ------------------------------------ #
#    hidden layer1 to input layer      #
# ------------------------------------ #

D_S_H1 = derivative_sigmoid(H1)
D_W1 =  D_H1 * D_S_H1 * x
```

- Update:
  - 計算完 D_W1,D_W2,D_W3 之後就可以把原本的W 做更新
  - Wi -= learning_rate * D_Wi (i=1,2,3)

Result of your testing



epoch 0 loss : [0.34541456]
epoch 1000 loss : [0.12173195]
epoch 2000 loss : [0.11233186]
epoch 3000 loss : [0.09197941]
epoch 4000 loss : [0.04745425]
epoch 5000 loss : [0.04004195]
epoch 6000 loss : [0.03425453]
epoch 7000 loss : [0.02903704]
epoch 8000 loss : [0.02468709]
epoch 9000 loss : [0.0209268]
-0.07417799559397675
1.1410099486091156
0.009853548471602469
1.1090164141045973
0.09158834915376879
1.0261685856310656
0.151814825581166
0.8373971183849536
0.17973001167712255
0.48163883514397465
0.17539492796657052
0.14646904027422059
0.5651516260834861
0.10316953114684457
1.0983602035372537
0.05466979617698087
1.1222827514005307
0.007609096074737032
1.000905203362808
-0.033974158797656084
0.9152481309175715

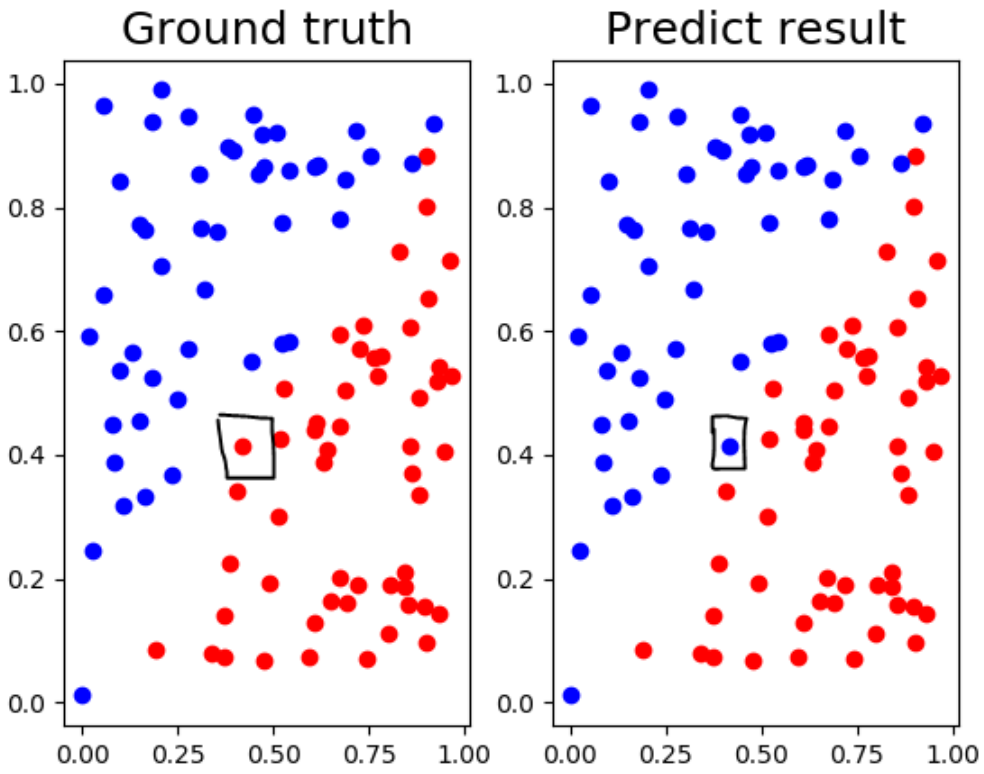


epoch 0 loss : [0.22406848]
epoch 1000 loss : [0.02268306]
epoch 2000 loss : [0.00805109]
epoch 3000 loss : [0.00679366]
epoch 4000 loss : [0.00690809]
epoch 5000 loss : [0.00666526]
epoch 6000 loss : [0.00633083]
epoch 7000 loss : [0.00602168]
epoch 8000 loss : [0.0057407]
epoch 9000 loss : [0.00548266]
0.020093317605886618
0.4301147289381593
1.056069791548032
0.60501290209906
0.9921301999196606
0.9825645415379353
1.0011785263913942
0.9987454634265082
0.04006874658314441
-0.06521526777429187
-0.0276470373869504
0.031711229089665505
-0.060216564025141084
1.103981056650417
0.9837259338991894
0.061992302655516696
1.0475364076127869
0.0407547253497329
0.033807947109119496
-0.009284955030353537
0.04415983268134349
0.024820503604944477
0.01360645784080905
1.0551384212455193
-0.028761920952579345

0.009349463295915239
0.9938158077020224
0.0336288223792689
0.055301624743229194
-0.022869698696691643
1.0395673411207524
-0.03254734486783972
0.8652382887552736
-0.08119991207943045
1.0264686893050055
0.9859247181550632
-0.014970561313703001
0.01424648354843061
1.0013195528550278
0.9749903998211285
1.0462095735302583
1.004690736045594
-0.07736521801040364
0.04903472146457766
0.03405084506733003
0.030442376119727665
0.99490767880758
1.0966743104134427
0.04241636429799556
1.0213615896461594
0.9823497265836558
0.023364043699070525
0.9896996600642465
0.9875493824535424
1.0220077531821024
0.9811476392073544
1.0595097281307173
-0.0823637058744131
0.579109436687258
1.0236765996847463
-0.03902095688828955
-0.0057045225586560555
0.9993629611386745
-0.05084223552364886
-0.062316786769353705
1.0039044366818626
0.9958102984915915
0.9994887232885825
0.09297988624773135
0.06477687148946387
1.0498511236247072
0.3372997116508678
0.00022525477253587667
-0.021915019138396552
-0.08054046199135678
-0.08542986415481946
0.07881907101892205
0.04120599788331347
1.002009999406094
1.012851637884913
1.0522736288870418
-0.014890645754009668
-0.057396369068444475

-0.06285211077948549
1.0139625731401138
0.06468824593253375
-0.006143722199024371
0.10361331210683744
1.0068437586169408
0.0411224683536231
-0.06602176885492872
0.047900559636969486
1.1103210042943656
1.049755415924407
1.0329653495736326
0.8249855053260946
0.03955565256950866
0.7089006151821751
0.05235956546773313
0.9849591372476765

- generate_linear 資料當中有機率在 中間線 的部分估計錯誤



- 關於初始化
  - 使用Xavier Glorot提出的方法

```
def generate_init_weight_matrix(m,n):
    x = np.random.randn(n, m) / np.sqrt(n /

    Q = np.array(x).reshape(m,n)
    return Q
```

Discussion

在課堂中常常會直接跳過各個element的偏微分，直接就計算整個weight matrix的偏微分
在寫程式的時候容易搞不清楚到底是element或者是matrix

直到看了一篇有實際假設變數值的教學才比較明白

另外有關於sigmoid derivative 推導