DLP HW1 Report

0756138 黃伯凱

March 25, 2019

1 Introduction

Neural Network 是由許多層 Neuron 以及 activation function 組合而成，兩層 layer 之間可以是 fully connected 也可以不是。每條 edge 上都有 weight，這些 weight 就是我們希望 neural network 可以學會的值。每一層 neuron 的 value 都是由前一層 neuron 與 weight 線性組合而成，而 output 則是將此線性組合的結果經過 activation function 得到。

從第一層不斷的線性組合、經過 activation function ...直到算出 output layer 的過程稱為 **Forward**。而更新 weight 的方式則是由後往前，稱為 **Backpropagation**。

由於更新 weight 是在計算 gradient，所以會有很多 chain rule 在裡面。從第一層 layer 開始更新的話會遇到 chain rule 的問題，導致 gradient 很難計算。所以才需要從後面往前更新 weight，故稱為 Backpropagation。

這一份報告會詳細介紹 neural network 的原理以及 backpropagation 的計算方式，最後呈現 linear, XOR classifier 的實驗結果以及 loss。

2 Experiment Setups

2.1 Sigmoid Functions

Sigmoid function 公式如下所示：

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

而 sigmoid 曲線為下圖

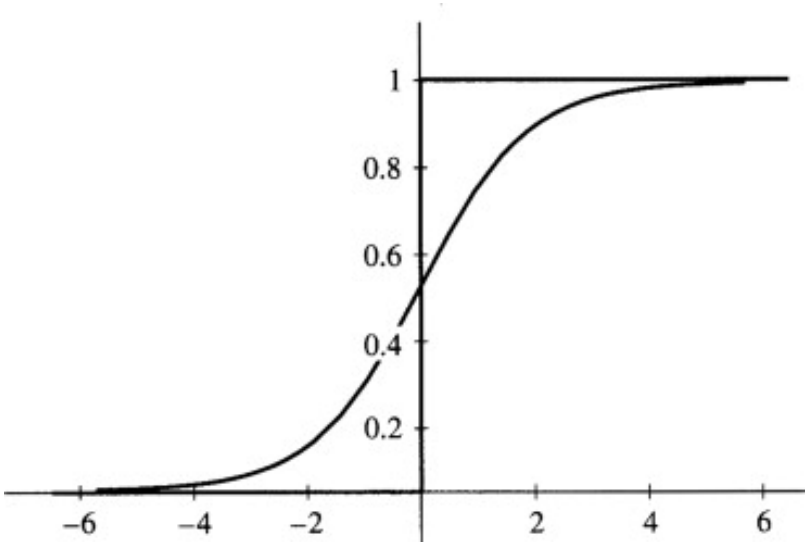


Figure 1: Sigmoid Function

我的 sigmoid function 如下：

```
def sigmoid(self, x):
    return 1/(1+np.exp(-x))
```

Figure 2: Sigmoid Function Code

另外還需要注意的是 sigmoid function 的微分，因為在做 backpropagation 的時候會用到。公式如下：

$$\frac{d}{dx}\frac{1}{1+e^{-x}} = \frac{d}{dx}(1+e^{-x})^{-1} = -(1+e^{-x})^{-2}\frac{d}{dx}(1+e^{-x})$$

$$= -(1+e^{-x})^{-2}(-e^{-x}) = \frac{e^{-x}}{(1+e^{-x})^2}$$

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

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

$$= \sigma(x)(1-\sigma(x))$$

Figure 3: Derivative of Sigmoid

Sigmoid 微分如下，因為我的 input 值已經是 sigmoid function，所以此 function code 裡面不用再做一次 sigmoid：

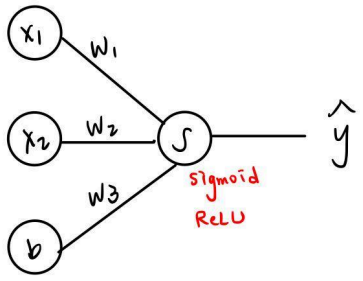
```
def sigmoid_prime(self, x):
    return x * (1 – x)
```

Figure 4: Derivative of Sigmoid Code

最後，sigmoid function 還扮演著一個很重要的角色。由於我們的 decision boundary 並非每一次都會是線性的，大多數都會是非線性的。這個時候非線性函數 sigmoid 就發揮它的作用了，線性的 input 通過 sigmoid function 之後會變成非線性函數，這個時候就可以求得我們想要的非線性 decision boundary。

2.2 Neural Network

神經網路是由許多 nueron 組成的架構，Figure 5(a) 為簡易的架構，Figure 5(b) 為估計值的算法：



(a) Neural Network

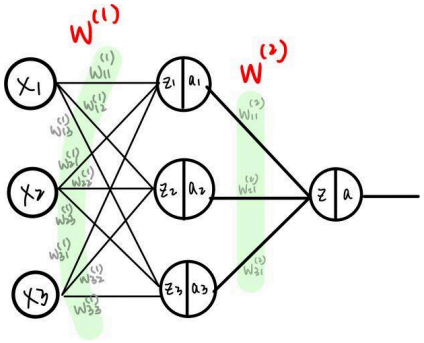
$$\hat{y} = \sigma(w_1 x_1 + w_2 x_2 + w_3 b)$$
$$= \sigma(WX + b)$$

(b) Neural Network Linear Combination

Figure 5: Simple Neural Network Architecture

上述公式裡的 w 權重，即為我們希望神經網路可以學起來的值。

下圖則為 3 個 input、一層 hidden layer、一個 output 的 neural network (z 為該 neuron 的輸入、a 為其輸出)：



(a) Neural Network

$$z_1 = w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2 + w_{31}^{(1)} x_3$$
$$z_2 = w_{12}^{(1)} x_1 + w_{22}^{(1)} x_2 + w_{32}^{(1)} x_3$$
$$z_3 = w_{13}^{(1)} x_1 + w_{23}^{(1)} x_2 + w_{33}^{(1)} x_3$$
$$z = w_{11}^{(2)} a_1 + w_{21}^{(2)} a_2 + w_{31}^{(2)} a_3$$
$$a = \sigma(z)$$

(b) Neural Network Linear Combination

Figure 6: Neural Network Architecture

上述步驟就是 neural network 的 forward 步驟，我的程式碼如下：

L_in : neuron input
L_out : neuron output

```
def forward(self, y):
    for idx in range(1, len(self.L_out)):
        # Each col represents value of each node (input_num * hidden_size)
        self.L_in[idx] = np.matmul(self.L_out[idx-1], self.w[idx-1])
        self.L_out[idx] = self.sigmoid(self.L_in[idx])
```

Figure 7: Forward Code

做完 forward 算出 predict y 之後，再去計算與 ground truth 的差異。利用這些差異可以更新 weight，這個步驟就是在學真正的 weight 應該為何。而這個步驟就是 backpropagation，這個部分會在下個段落提到。

2.3 Backpropagation

Backpropagation 的目的是在更新 weight 的值，其實就是在做 gradient descent。一個 neural network 有非常多個 weight 要更新，所以要把 Error 對所有的 weight 做微分找最佳的 weight 組合。每一次做完 forward 之後，Error 就好像下圖中的最高點，weight 分別是圖中的每一個維度，而我們要找到的有最低 Error 的 weight 組合 (也就是圖中的 Global Min)。所以 w_{jk} 的更新值即為 $\frac{\partial E}{\partial w_{jk}}$。

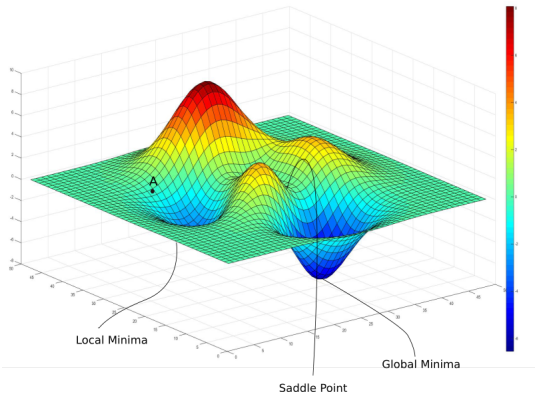


Figure 8: Gradient Descent

這邊舉幾個例子來看要怎麼將 Error 對 weight 微分，並以上一個段落的 neural network 為例：

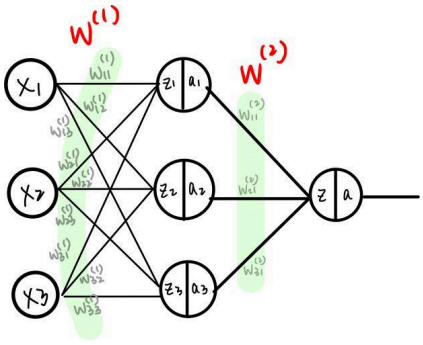


Figure 9: Neural Network

先看對 W^2 的微分 (E 為 loss)：

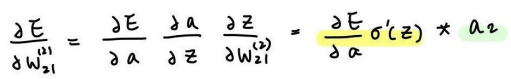


Figure 10: Partial Derivative of E with respect to W^2

再看對 W^1 的微分 (E 為 loss)：

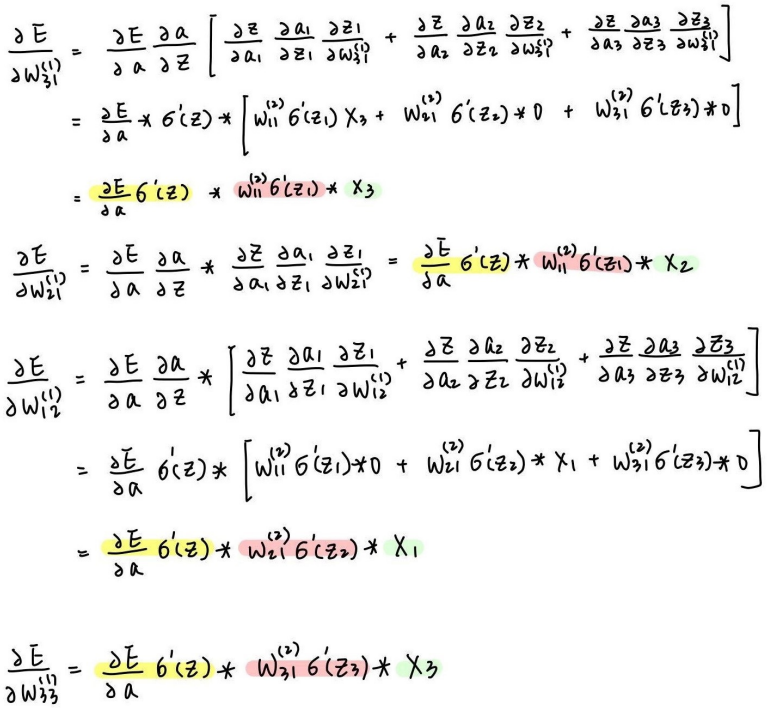


Figure 11: Partial Derivative of E with respect to W^1

從上面的式子可以發現，圖相同色塊的部分其實都是固定的形式，所以是可以寫成通式的。為了方便推導 backpropagation 的通式，這裡要先定義各個參數的意義：

z_j^l：第 l 層第 j 個 neuron 的輸入

a_j^l：第 l 層第 j 個 neuron 的輸出，即為經過 activation function 的 z_j^l

w_{jk}^l：第 (l-1) 層第 j 個 neuron 與第 l 層第 k 個 neuron 之間的 weight

假設 output Layer 為第 L 層，先算出最後第 L 組 weight 的偏微分，可以得到藍色框框的值，將他記為 δ_j^L，接下來的偏微分還會用到這個值。

$$\frac{\partial E}{\partial w_{jk}^L} = \frac{\partial E}{\partial a_k^L}\frac{\partial a_k^L}{\partial z_k^L}\frac{\partial z_k^L}{\partial w_{jk}^L} = \frac{\partial E}{\partial a_k^L}\sigma'(z_k^L) * a_j^{L-1}$$

Figure 12: Partial Derivative with respect to W^L

接下來算出第 (L-1) 組 weight 的偏微分，此時就會用到剛才算出來的 δ_j^L。接下來，一樣紀錄 δ_j^{L-1}，因為下一次篇微分會用到。

$$\frac{\partial E}{\partial w_{jk}^{L-1}} = \frac{\partial E}{\partial a_k^{L-1}}\frac{\partial a_k^{L-1}}{\partial z_k^{L-1}}\frac{\partial z_k^{L-1}}{\partial w_{jk}^{L-1}}$$

$$= \sum_i \left[\frac{\partial E}{\partial a_i^L}\frac{\partial a_i^L}{\partial z_i^L}\frac{\partial z_i^L}{\partial a_k^{L-1}}\right] * \sigma'(z_k^{L-1}) * a_j^{L-2}$$

$$= \sum_i \left[\frac{\partial E}{\partial a_i^L}\sigma'(z_i^L) * w_{ki}^L\right] * \sigma'(z_k^{L-1}) * a_j^{L-2}$$

($\frac{\partial E}{\partial a_i^L}\sigma'(z_i^L) = \delta_i^L$)

Figure 13: Partial Derivative with respect to W^{L-1}

以此類推，算第三組 weight 也做重複的動作。

$$\frac{\partial E}{\partial w_{jk}^{L-2}} = \frac{\partial E}{\partial a_k^{L-2}} \frac{\partial a_k^{L-2}}{\partial z_k^{L-2}} \frac{\partial z_k^{L-2}}{\partial w_{jk}^{L-2}}$$

$$= \sum_i \left[\frac{\partial E}{\partial a_i^{L-1}} \frac{\partial a_i^{L-1}}{\partial z_i^{L-1}} \frac{\partial z_i^{L-1}}{\partial a_k^{L-2}} \right] * \sigma'(z_k^{L-2}) * a_j^{L-3}$$

$$= \sum_i \left[\frac{\partial E}{\partial a_i^{L-1}} \sigma'(z_i^{L-1}) * w_{ki}^{L-1} \right] * \sigma'(z_k^{L-2}) * a_j^{L-3}$$

Figure 14: Partial Derivative with respect to W^{L-2}

由上述數學式可以發現，只要記錄每一次算出來的 δ 就可以在下一次使用。而且每一次的偏微分其實都是固定的形式，因此我們可以把偏微分以及 δ 都寫成通式如下：

$$\begin{cases} \delta_j^L = \frac{\partial E}{\partial a_j^L} \sigma'(z_j^L) \\ \delta_j^l = \sigma'(z_j^l) \sum_i w_{ji}^{l+1} \delta_i^{l+1} \\ \frac{\partial E}{\partial w_{jk}^l} = a_j^{l-1} \delta_j^l \end{cases}$$

Figure 15: Formula of Backpropagation

有了這個通式之後就可以用程式碼呈現 backward，並且更新 weight：

```
def backward(self, y):
    # delta : applying derivative of sigmoid to error (inputNum * LayerSize)
    self.delta[-1] = (y-self.L_out[-1]) * self.sigmoid_prime(self.L_out[-1])
    for idx in reversed(range( 1, len(self.L_out)-1 )):
        self.delta[idx] = np.matmul(self.delta[idx+1], self.w[idx].T)
        self.delta[idx] *= self.sigmoid_prime(self.L_out[idx])

    # adjusting weights (input_size * h1_size)
    for idx in range(len(self.w)):
        self.w[idx] += self.lr * np.matmul( self.L_out[idx].T, self.delta[idx+1] )
```

Figure 16: Backpropagation Code

總之，neural network 即為 input neuron 的線性組合再透過非線性的 sigmoid、relu 將其轉換成非線性函數。經過數次重複的動作後算出 predict y。利用 predict y 與 ground truth y 的差異並利用 backpropagation 更新 weight。重複好幾次 forward 和 backpropagation 之後就可以將 weight 更新成我們想要 neural network 學會的值。

3 Results

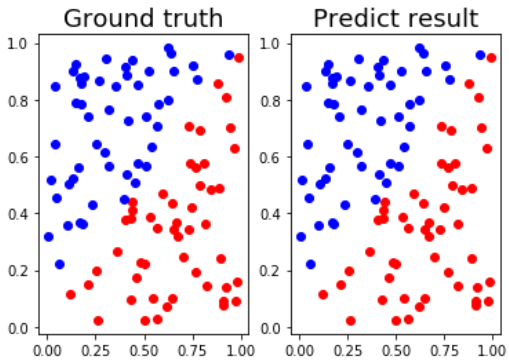
3.1 Screenshot and Comparison figure

3.1.1 Linear

Linear random data 的 loss (每 5000 epoch 印一次)，以及 classify 結果：

```
epoch 5000 loss : 2.4809000529235078e-06
epoch 10000 loss : 1.2848406259874037e-06
epoch 15000 loss : 8.716957265789108e-07
epoch 20000 loss : 6.606307643662063e-07
epoch 25000 loss : 5.321677272233936e-07
epoch 30000 loss : 4.456439937884803e-07
epoch 35000 loss : 3.83362328656941e-07
epoch 40000 loss : 3.3636823239466545e-07
epoch 45000 loss : 2.9963947865156094e-07
epoch 50000 loss : 2.701391594794243e-07
epoch 55000 loss : 2.5746325634845065e-07
epoch 60000 loss : 2.5156026217645894e-07
epoch 65000 loss : 2.487095078355357e-07
epoch 70000 loss : 2.473080099882682e-07
epoch 75000 loss : 2.466131581226936e-07
epoch 80000 loss : 2.4626719198176155e-07
epoch 85000 loss : 2.4609457206923067e-07
epoch 90000 loss : 2.460083526798647e-07
epoch 95000 loss : 2.4596526559915405e-07
epoch 100000 loss : 2.4594372770883233e-07
```

(a) Epoch Loss



(b) Classify Result

Figure 17: Linear Result

把所有 loss 印出來看看 learning rate 是否有問題：

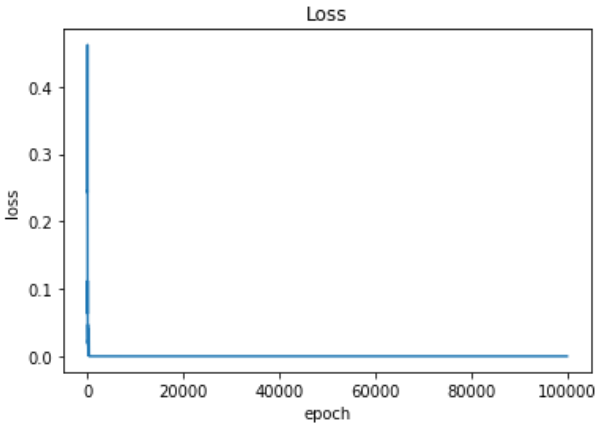


Figure 18: Linear Loss

Prediction 的結果如下：

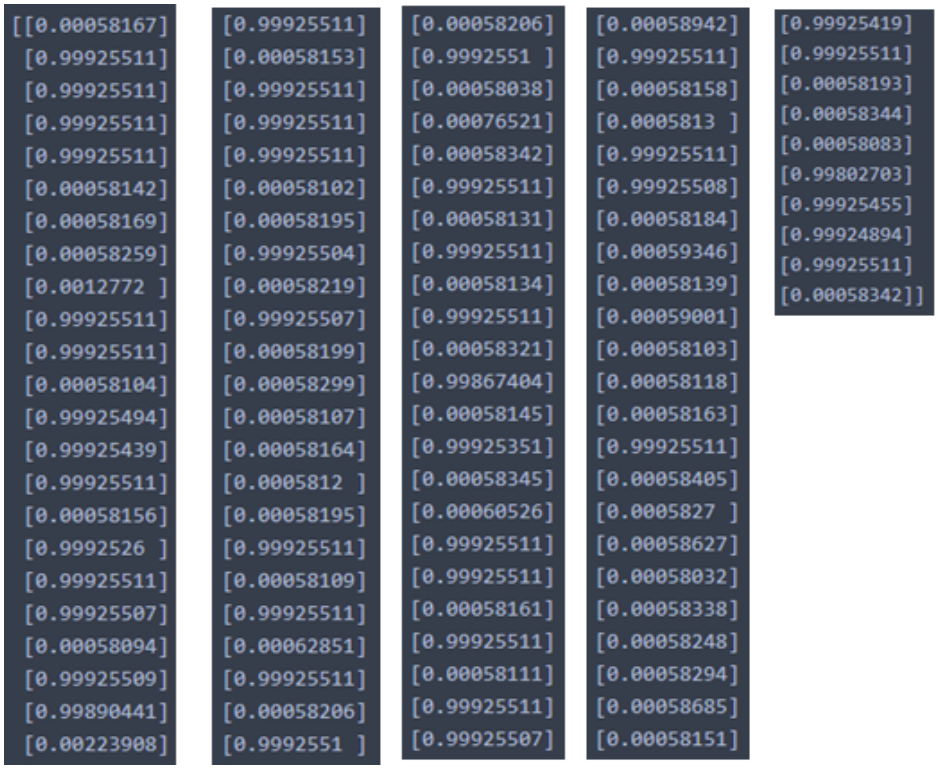


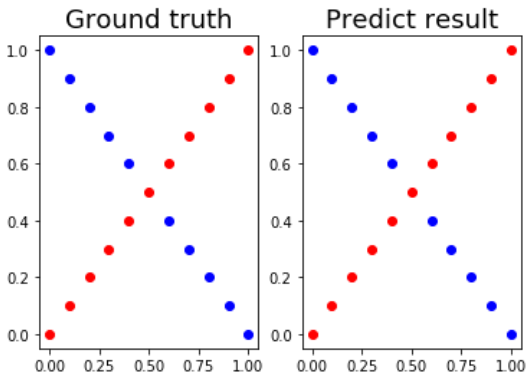
Figure 19: Linear Prediction Result

3.1.2 XOR

XOR random data 的結果 (每 5000 epoch 印一次)，以及 classify 結果：

```
epoch 5000 loss : 6.909776319400981e-05
epoch 10000 loss : 3.199231209262838e-05
epoch 15000 loss : 2.0663833061535e-05
epoch 20000 loss : 1.5216862697498355e-05
epoch 25000 loss : 1.202477185291968e-05
epoch 30000 loss : 9.930926763787658e-06
epoch 35000 loss : 8.453225671449708e-06
epoch 40000 loss : 7.355328660124542e-06
epoch 45000 loss : 6.507889822174798 5e-06
epoch 50000 loss : 5.8342206625412665e-06
epoch 55000 loss : 5.285993957110996e-06
epoch 60000 loss : 5.048494862295648e-06
epoch 65000 loss : 4.937508392216238e-06
epoch 70000 loss : 4.883810041968098e-06
epoch 75000 loss : 4.857392810049435e-06
epoch 80000 loss : 4.84429016624814e-06
epoch 85000 loss : 4.837765090691901e-06
epoch 90000 loss : 4.834509083970003e-06
epoch 95000 loss : 4.8328827095758475e-06
epoch 100000 loss : 4.832069929146924e-06
```

(a) Epoch Loss



(b) Classify Result

Figure 20: Linear Result

把所有 loss 印出來看看 learning rate 是否有問題：

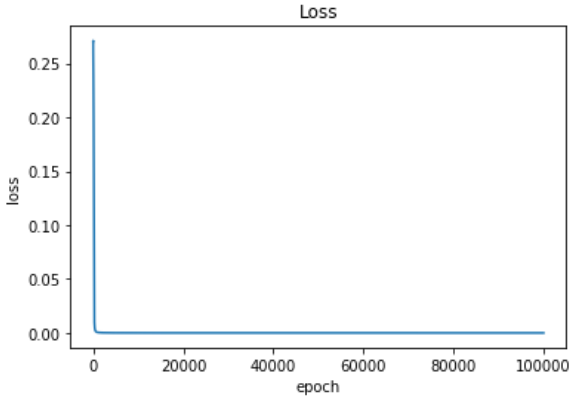


Figure 21: Loss

Prediction 的結果如下：

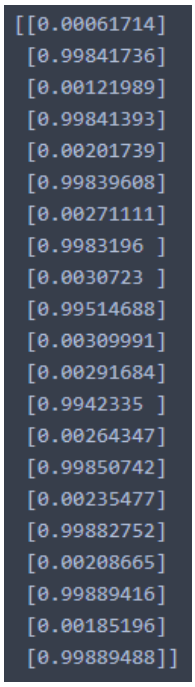


Figure 22: Prediction Result

3.2 Others to present

我的實作比較彈性，有幾個特色，以下分成三個部分討論。

3.2.1 Matrix Form

若使用 elementwise 的方式寫 neural network，每一層 Layer 的 forward 計算都要 1 ~ 2 行。因此如果有 100 層 Layer 就要 100 ~ 200 行，非常的不彈性。所以我把每一層的 input, output, weight 都寫成 Matrix form，以節儉 code 且避免因為冗長而造成的錯誤。

再看一次我的 forward, backward 的 code，都是用 matrix form 來完成：

```
def forward(self, y):
    for idx in range(1, len(self.L_out)):
        # Each col represents value of each node (input_num * hidden_size)
        self.L_in[idx] = np.matmul(self.L_out[idx-1], self.w[idx-1])
        self.L_out[idx] = self.sigmoid(self.L_in[idx])
```

Figure 23: Forward Code

```
def backward(self, y):
    # delta : applying derivative of sigmoid to error (inputNum * LayerSize)
    self.delta[-1] = (y-self.L_out[-1]) * self.sigmoid_prime(self.L_out[-1])
    for idx in reversed(range( 1, len(self.L_out)-1 )):
        self.delta[idx] = np.matmul(self.delta[idx+1], self.w[idx].T)
        self.delta[idx] *= self.sigmoid_prime(self.L_out[idx])

    # adjusting weights (input_size * h1_size)
    for idx in range(len(self.w)):
        self.w[idx] += self.lr * np.matmul( self.L_out[idx].T, self.delta[idx+1] )
```

Figure 24: Backpropagation Code

3.2.2 Flexible Hidden Size

因為並不是所有的 neural network 的 hidden layer 都是同一個 size，所以我的實作也讓 hidden layer, weight 都可以自訂。

自訂的方法如下圖紅框，每一個數字代表每一層的 hidden size：

```
epoch = 100000
X, y = generate_linear(100)
Xpredict = X

hiddenSize = (10,5,3,5)
NN = Neural_Network(X, y, hiddenSize)
for epoch in tqdm(range(epoch)):
    NN.train(X, y, epoch)
show_result(X, y, NN.L_out[-1])
show_loss(epoch, NN.loss)
```

Figure 25: HiddenSize Code

由於 hidden size 是彈性的，所以每一層 layer input 和 output 的 initial 方式也必須根據每一層 size 而調整。而我的 initial 方式是讓還沒計算到的值都預設為 None。

```
# Initial i/o, delta of each layer
self.L_in = []
self.L_out = []
self.delta = []
self.L_in.append(X)  # Set L_in[0] as original input
self.L_out.append(X/np.amax(X, axis=0))  # L_out[0] normalize the value of orignial input
self.delta.append(np.array([None]))  # The first layer has no delta
for idx in range(len(self.hiddenSize)):
    # delta : applying derivative of sigmoid to error (inputNum * hiddenSize)
    self.delta.append( np.array( [[None]*self.hiddenSize[idx]]*self.inputNum ))
    self.L_in.append ( np.array( [[None]*self.hiddenSize[idx]]*self.inputNum ))
    self.L_out.append(np.array( [[None]*self.hiddenSize[idx]]*self.inputNum ))
# delta : applying derivative of sigmoid to error (inputNum * outputSize)
self.L_in.append ( np.array( [[None]*self.outputSize]*self.inputNum ))
self.L_out.append(np.array( [[None]*self.outputSize]*self.inputNum ))
self.delta.append( np.array( [[None]*self.outputSize]*self.inputNum ))
```

Figure 26: Initial Hidden Layer Code

weight 也一樣必須根據 hidden size 來調整，而我的 initial 方式則是讓每一層 weight 初始值都是隨機值。

```
# Initial weight
self.w = []
self.w.append(np.random.randn(self.inputSize, self.hiddenSize[0]))  # input -> hidden #1
for idx in range(len(self.hiddenSize)-1):  # hidden #idx -> hidden #idx+1
    self.w.append(np.random.randn(self.hiddenSize[idx], self.hiddenSize[idx+1]))
self.w.append(np.random.randn(self.hiddenSize[-1], self.outputSize))  # last hidden -> output
```

Figure 27: Initial Weight Code

3.2.3 Dynamic Learning Rate

我的初始 learning rate 設為 0.5，但是如果一直用這個 learning rate 的話，可能到後面會找不到最小值。所以我每 5000 個 epoch 會比較一次，如果 $0.9 < \frac{loss_{new}}{loss_{old}} < 1.1$，則會將 learning rate 除以 2。確保 gradient descent 不會一直在最小值兩側跳來跳去導致降不下去。

```
if (epoch+1) % 5000 == 0:
    print('epoch %s loss : %s' %(str(epoch+1), str(self.loss[-1])))
    if 0.9 < self.loss[-1]/self.loss[-4999] < 1.1:
        self.lr /= 2
```

Figure 28: Learning Rate Code

4 Discussion

這份作業比較困難的是 backpropagation 的部分，因為想要把他寫的比較有彈性、又想用 matrix form 表示，所以必須推導 backpropagation 的數學公式，因此在數學上花了非常多的時間。另外，在調整 learning rate 也試了非常多次，不管 lr 太大或太小都會讓 loss 卡住降不下去，所幸我設定的參數都還能讓 loss 降到 10^{-6} 上下。