DLP
LAB 1

謝侑哲

112550069

Github:

https://github.com/youzhe0305/NYCU-DLP

1. INTRODUCTION

本次 lab 為，在不使用深度學習套件的情況下，使用 NumPy 跟基本的 library 時做出一個有 2 層 hidden layer 的神經網路，示意圖如下。

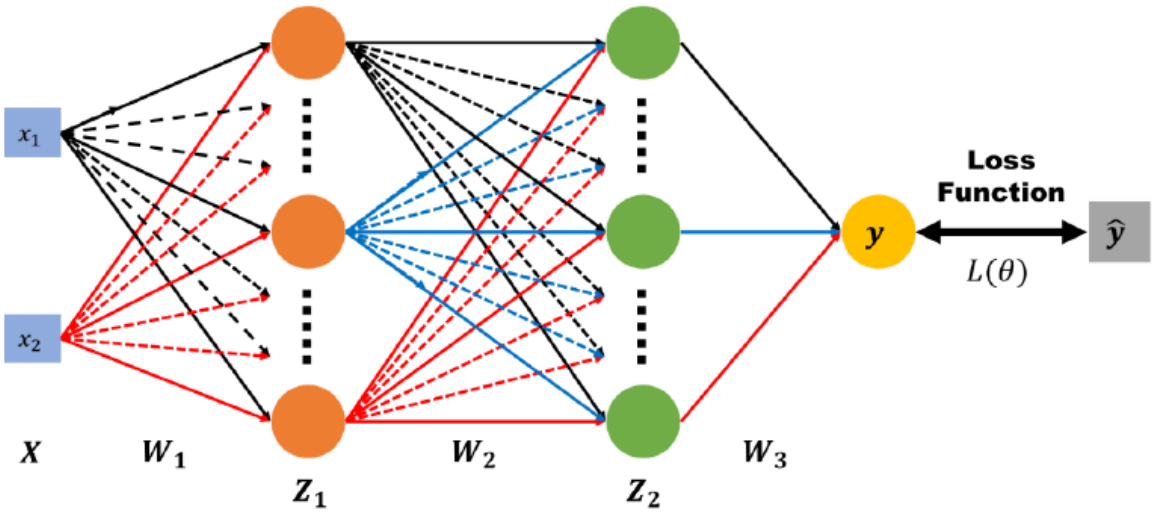
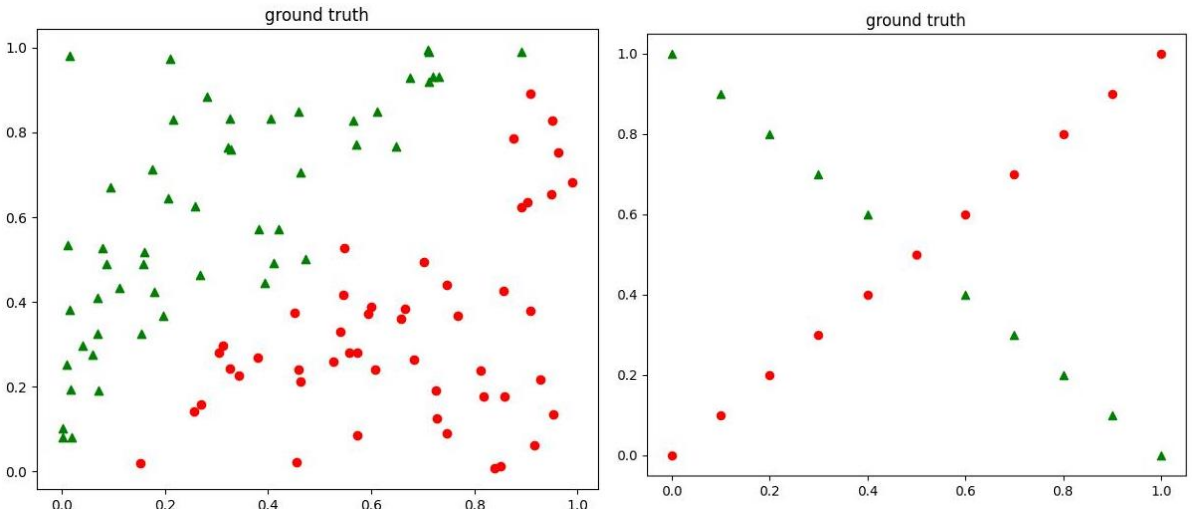


Figure 2. Forward pass

Lab 中我們實現了神經網路的 architecture, forward, backpropagation 以及其他需要使用到的函數的導數計算等等。

這個 lab 中，我們用來進行訓練與測試的數據為，在二維平面上被分成 2 個類別的點，也就是說我們的神經網路實現的是二元分類問題，採用 Logistic Regression 來實現。訓練的資料示意圖如下:



2. Experiment setups

A. Sigmoid functions

Lab 中，我們首先實現了 sigmoid function，這是一個可以把輸出限制在 0~1 之間的 activation function，算式為$sigmoid(x) = 1/(1 + e^{-x})$，其中 sigmoid 給出的輸出，也可以做為二元分類的機率，為 y=1 的 class 的機率，相對的也可以算出 y=0 的 class 的機率。用線性層搭配上 sigmoid 就可以變成該 lab 的實作對象 Logistic Regression。

以下為我的 sigmoid 實現，輸入 x 會是 numpy 的 array，並且是個二維的陣列，儲存了一個 batch 中的每個 sample 的資料。

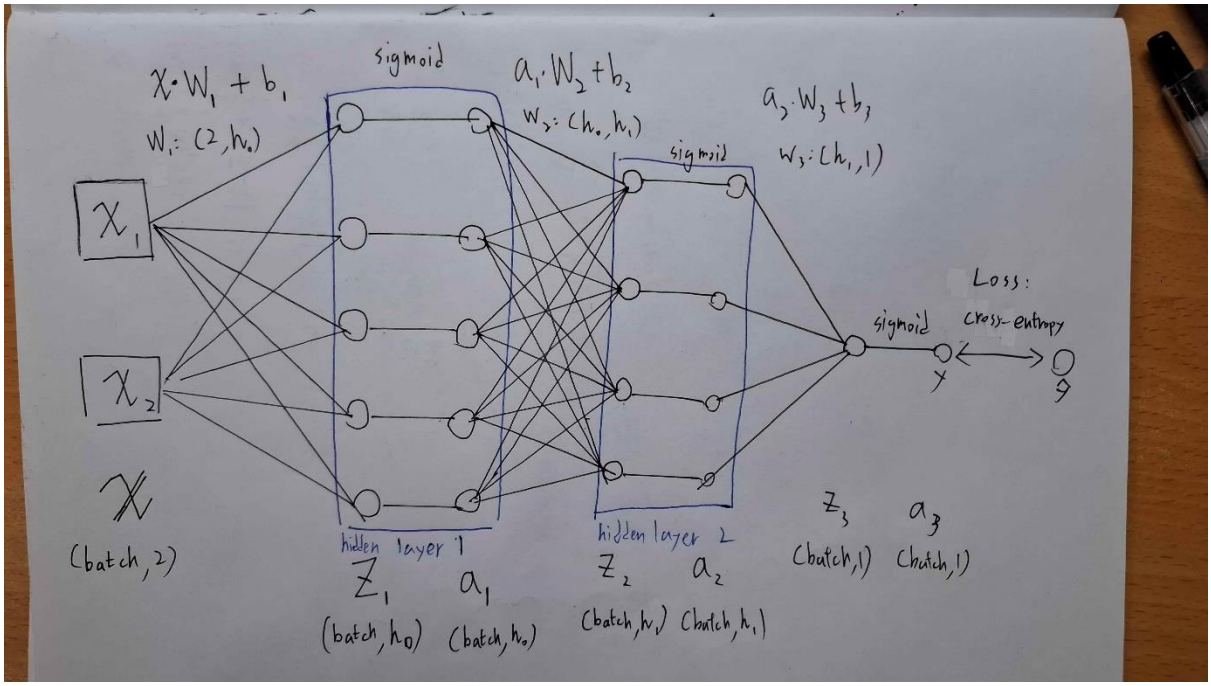
```
def sigmoid(x):
    '''
    Sigmoid function, the basic activation funciton behind each layer
    Make liner regression to logistic regression
    x.shape: batch_size * n
    '''
    return 1 / (1 + np.exp(-x))
```

另外，我也實現了 sigmoid 的導數，為$\frac{\partial Loss}{\partial x} = \frac{\partial Loss}{\partial y}\frac{\partial y}{\partial x}$，如下:

```
def sigmoid_derivative(x, grad_Y):
    '''
    Derivative of d Loss / d x
    formula: sigmoid(x)
    x.shape: batch_size * n
    ret.shape: batch_size * n
    '''
    return grad_Y * sigmoid(x) * (1 - sigmoid(x))
```

B. Neural network

如同在 Introduction 提到的，我實作了 2 層 hidden layer 的神經網路，並給出了示意圖，這裡我將描繪更加詳細的圖片，把架構用圖的方式描寫得更詳細。



圖中可以看到，層數的輸入依序是 X->Z1->a1->Z2->a2->Z3->a3，Z1,Z1 為 hidden layer 的部分 Z3 為 output layer，其中每個 layer 後面都會加上 activation function 讓模型能有更 non-linear 的能力。

另外，我設計的神經網路架構，可以一次吃好幾筆資料(一個 batch)，並把整個 batch 透過矩陣運算的方式一起計算。

實作上整體的架構如下:

```
class SimpleNN():
    def __init__(self, batch_size, hidden_layer_size, learning_rate=0.1):···

    def forward(self, X, batch_size = None):···

    def criterion(self, y_hat, pred_y):···

    def backproapgation(self, X, y_hat, pred_y):···

    def para(self):···

    def train(self, X_train, Y_train, n_epoch, sample_size, batch_size, train_name):···

    def test(self, X_test, Y_test, sample_size, test_name):···
```

Layer 的建構，我是採用直接在 class 用 NumPy array 設各個 layer 的 matrix，用 Z1, a1, W1 等等方式做命名區分，其中 W 權重的初始化為隨機在 0~1 之間的數值。

```
def __init__(self, batch_size, hidden_layer_size, learning_rate=0.1):
    '''
    form layers
    batch_size mean for each step, use how many samples
    hidden_layer_size.shape: 1 * 2, to decide the neurons in ith hidden_layer
    '''
    self.batch_size = batch_size
    self.lr = learning_rate

    self.W1 = np.random.uniform(0,1,(2,hidden_layer_size[0]))
    self.b1 = np.zeros(hidden_layer_size[0])
    self.Z1 = np.zeros((batch_size, hidden_layer_size[0]))
    self.a1 = np.zeros((batch_size, hidden_layer_size[0]))
```

Forward 的部分，因為神經元之間的連線、計算，可以整合成矩陣的運算，因此實作上我直接使用 NumPy 的矩陣乘法跟上面建構好的 Sigmoid function 來做。

```
def forward(self, X, batch_size = None):

    if batch_size == None:
        batch_size = self.batch_size
    # Layer 1
    self.Z1 = X@self.W1 + np.tile(self.b1, (batch_size, 1))
    self.a1 = sigmoid(self.Z1)
```

Loss Function 因為輸出是二元分類的機率，所以使用 Binary Cross Entropy 來做評估，另外因為一次吃了整個 batch，因此 loss 我是採用 batch 裡面每一筆資料的平均(總和值用矩陣乘法計算)。

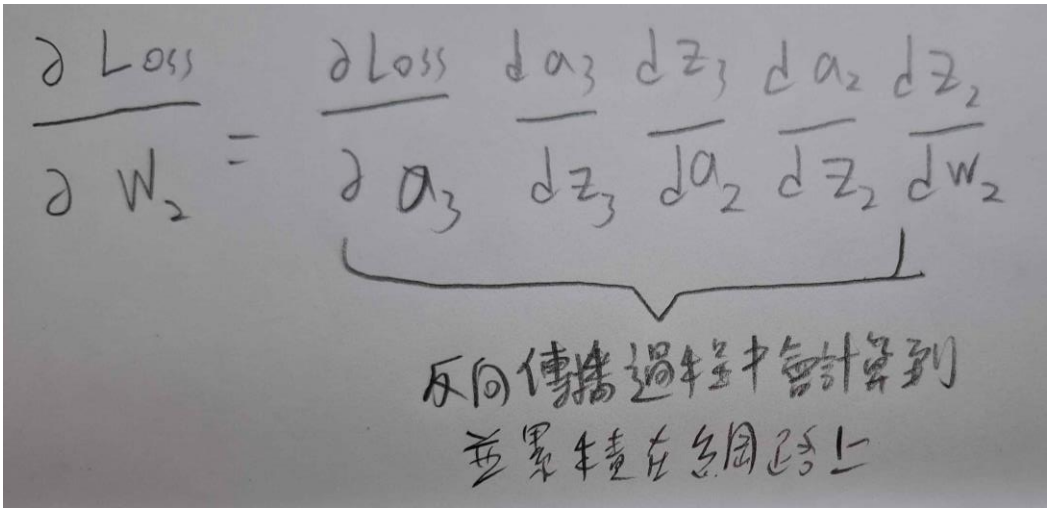
```
def criterion(self, y_hat, pred_y):
    '''
        y, pred_y shape: batch * 1
        With sigmoid, it's 2 classification problem
        Use cross-entropy as loss function
        for y = sigma(y * ln(y_pred))
    '''
    loss =  - (y_hat.T @ np.log(pred_y) + (1 - y_hat).T @ np.log(1 - pred_y)) / self.batch_size
    return loss[0][0]
```

訓練的部分，最基本的 Model 我是使用 mini-batch Gradient Descent 實作。

```
def train(self, X_train, Y_train, n_epoch, sample_size, batch_size, train_name):
    loss_check_point_x = []
    loss_check_point_y = []
    for epoch in range(n_epoch):
        for batch in range( np.ceil(sample_size / batch_size).astype(int) ):
            inputs = X_train[batch*batch_size : min(sample_size, (batch+1)*batch_size), :]
            labels = Y_train[batch*batch_size : (batch+1)*batch_size, :]
            prediction = self.forward(inputs)
            loss = self.criterion(labels, prediction)
            self.backproapgation(inputs, labels, prediction)
```

C. Backpropagation

Backpropagation 的核心是透過微分的連鎖律，可以透過上一項的導數乘上下一項的導數，做到計算隔好幾層 layer 的導數傳遞。可以進而做到梯度下降讓 loss 值變得更小。Backpropagation 的原理如下圖:



因此為了做 Backpropagation，我首先實現了神經網路中會使用到的計算的導數，利用了上課提到的 Matrix 微分。

```python
def matrix_right_mul_W_derivative(X, grad_Y):
    '''
    Derivative of d Loss / d W
    formula Y = XW + b
    W.shape: n1 * n2
    X.shape: batch * n1
    Y.shape: batch * n2
    grad_Y.shape = batch * n2
    ret.shape: n1 * n2
    '''
    return  X.T @ grad_Y
```

```python
def matrix_right_mul_X_derivative(W, grad_Y):
    '''
    Derivative of d Loss / d W
    formula Y = XW + b
    W.shape: n1 * n2
    X.shape: batch * n1
    Y.shape: batch * n2
    grad_Y.shape = batch * n2
    ret.shape: n1 * n2
    '''
    return  grad_Y @ W.T
```

```python
def matrix_plus_derivative(grad_Y, batch_size):
    '''
    Derivative of d Loss / d b
    formula: Y = XW + b
    XW.shape: batch * n2
    Y.shape: batch * n2
    grad_Y.shape: batch * n2
    b.shape: batch * n2 (we should stack same b
    ret.shape: n2
    because ret is matrix, we should gain the me
    '''
    return np.sum(grad_Y, axis=0) / batch_size
```

(以上 Y=XW+b 中，Loss 對於 W, X, b 的梯度計算，並把後一層的梯度作為參數)

得到各算式的導數之後，就可以開始做 backpropagation 的流程，流程如下:

從 Cross-Entropy 的導數開始，一步一步往下傳，套入上面的梯度計算的函數，再

把 W, b 等參數減去梯度*學習率，進行更新。第二個 hidden layer 到 output 的程式碼如下圖，其他層同理。

```
grad_a3_yhat = - (y_hat / pred_y - (1 - y_hat) / (1 - pred_y)) # Loss: - (y_hat * np.log(
grad_Z3_a3 = sigmoid_derivative(self.Z3, grad_a3_yhat) # sigmoid(Z3)
grad_a2_Z3_W3 = matrix_right_mul_W_derivative(self.a2, grad_Z3_a3) # XW + b, cacualte W
grad_a2_Z3_b3 = matrix_plus_derivative(grad_Z3_a3, self.batch_size) # XW + b, cacualte b
grad_a2_Z3 = matrix_right_mul_X_derivative(self.W3, grad_Z3_a3) # XW + b, cacualte X

self.W3 -= grad_a2_Z3_W3 * self.lr
self.b3 -= grad_a2_Z3_b3 * self.lr
```

3. Results of testing

A. Screenshot and comparison figure

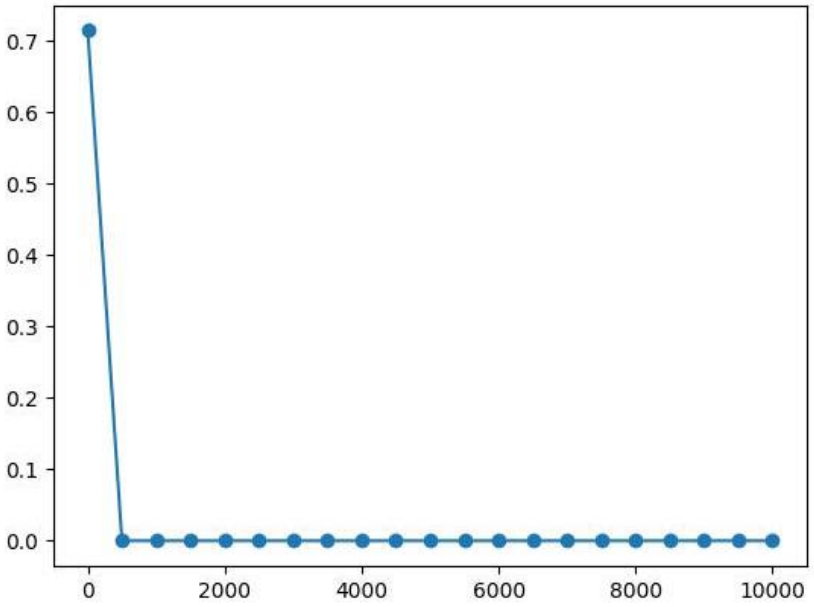
在訓練上，我皆採用 10000 個 epoch 來做訓練，0.1 的 learning rate，(10,10) 的 hidden units，並且每 500 個 epoch 做一次輸出，供我們查看 loss 值的變化。底下分別呈現 2 個 data function 產生的資料。

(1) def generate_liner(n=100)

loss 狀況:

```
epoch: 1/10000, loss: 0.714846645809776
epoch: 500/10000, loss: 3.825495883207572e-06
epoch: 1000/10000, loss: 2.7216791002645657e-07
epoch: 1500/10000, loss: 7.70741996555338e-08
epoch: 2000/10000, loss: 3.40661013917578e-08
epoch: 2500/10000, loss: 1.8746979670867292e-08
epoch: 3000/10000, loss: 1.1736948885148154e-08
epoch: 3500/10000, loss: 7.996295092536302e-09
epoch: 4000/10000, loss: 5.781747760066709e-09
epoch: 4500/10000, loss: 4.368763568571869e-09
epoch: 5000/10000, loss: 3.4148126968121884e-09
epoch: 5500/10000, loss: 2.741714670013236e-09
epoch: 6000/10000, loss: 2.2496694291388627e-09
epoch: 6500/10000, loss: 1.8793861787705623e-09
epoch: 7000/10000, loss: 1.5939053719011654e-09
epoch: 7500/10000, loss: 1.3692475232095879e-09
epoch: 8000/10000, loss: 1.189319755518404e-09
epoch: 8500/10000, loss: 1.0430049059946144e-09
epoch: 9000/10000, loss: 9.22428079404173e-10
epoch: 9500/10000, loss: 8.218876143476991e-10
epoch: 10000/10000, loss: 7.371745165386614e-10
test, sample_size: 100, accuracy: 100.0%
```

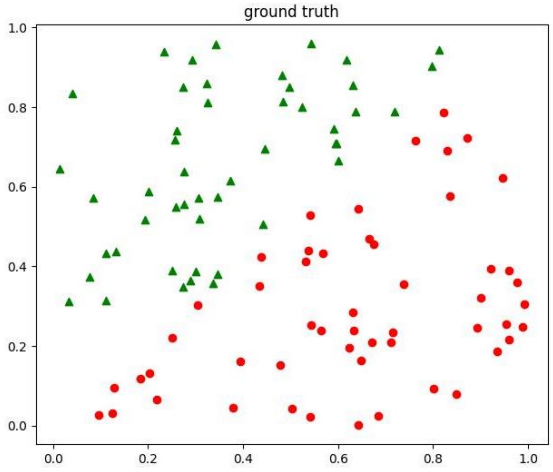
(training process)

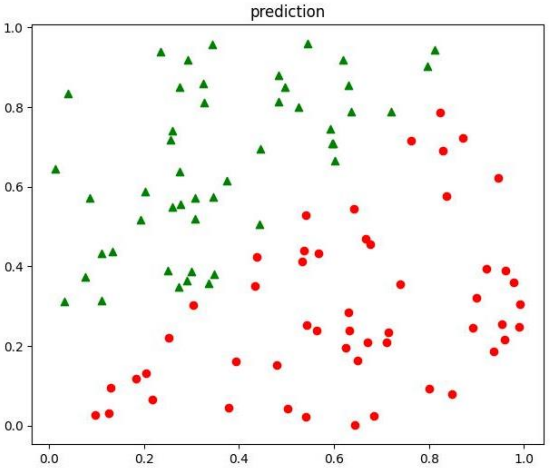


(epoch-loss curve)

可以看到在經過 10000 個 epoch 之後，loss 達到了幾近於 0，並且準確率達到了 100%，也就是說 model 能給出準確率高且信心也很高的預測。

分類結果比對:



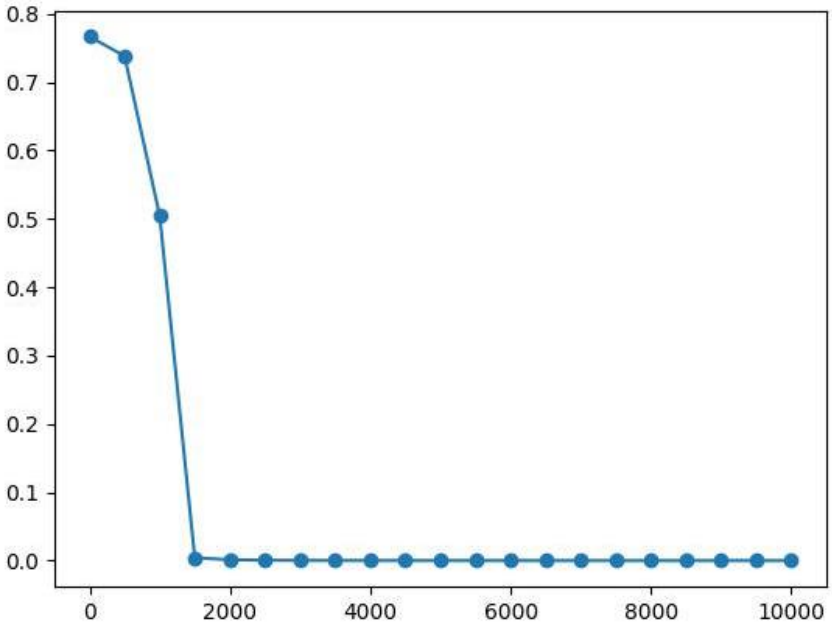


(2) def generate_XOR_easy(n=10)

loss 狀況:

```
epoch: 1/10000, loss: 0.7665386102174129
epoch: 500/10000, loss: 0.7379892552604955
epoch: 1000/10000, loss: 0.5045914821911811
epoch: 1500/10000, loss: 0.004195354904172128
epoch: 2000/10000, loss: 0.0012294847439982177
epoch: 2500/10000, loss: 0.0005503493104513755
epoch: 3000/10000, loss: 0.0003254067232777743
epoch: 3500/10000, loss: 0.00022199259221608147
epoch: 4000/10000, loss: 0.00016473865865767635
epoch: 4500/10000, loss: 0.00012913248665916426
epoch: 5000/10000, loss: 0.00010516979040325594
epoch: 5500/10000, loss: 8.809860037443561e-05
epoch: 6000/10000, loss: 7.540409005360181e-05
epoch: 6500/10000, loss: 6.564329422040106e-05
epoch: 7000/10000, loss: 5.793466021574287e-05
epoch: 7500/10000, loss: 5.1711960204382334e-05
epoch: 8000/10000, loss: 4.6596317322672166e-05
epoch: 8500/10000, loss: 4.2325439825058405e-05
epoch: 9000/10000, loss: 3.8712449340491056e-05
epoch: 9500/10000, loss: 3.562088025338406e-05
epoch: 10000/10000, loss: 3.294893424838955e-05
test, sample_size: 21, accuracy: 100.0%
```

(training process)

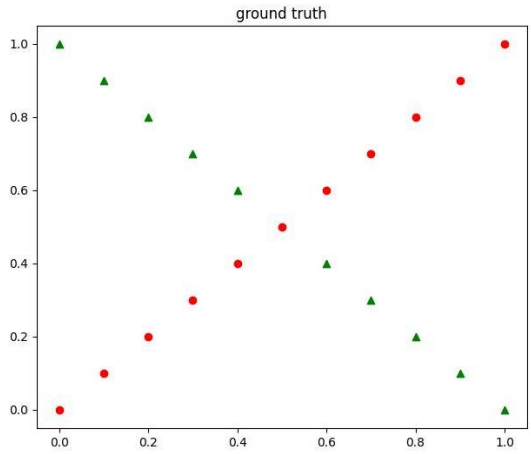


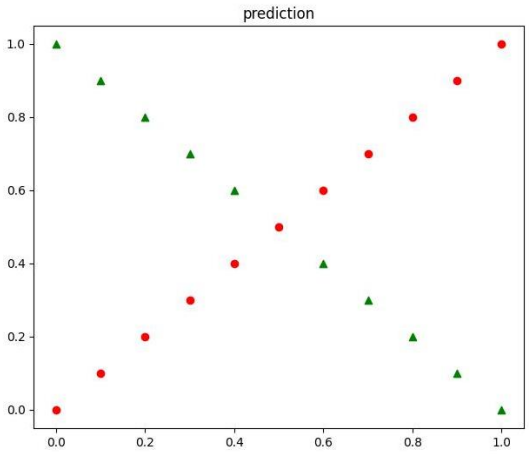
(epoch-loss curve)

可以看到在經過 10000 個 epoch 之後，loss 達到了幾近於 0，並且準確率達到了 100%，也就是說 model 能給出準確率高且信心也很高的預測。跟 liner 的資

料主要的差別是一開始 loss 下降的比較慢，很可能是因為 XOR 的資料相對比較複雜，更偏離線性可分，所以到了 1500 個 epoch 才進入收斂。

分類結果比對:



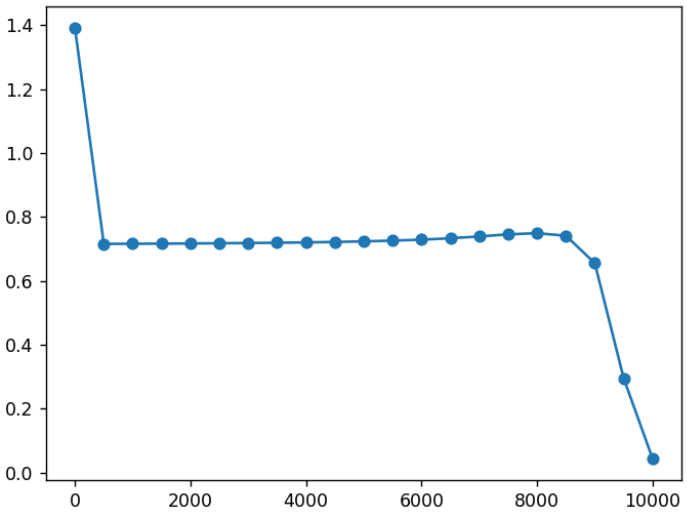


4. Discussion

A. Try different learning rates

因為 XOR data 的數據跑起來更明顯，所以我這裡呈現用不同的 learning rate 做出來的結果。同樣是每 500 epoch 為 1 個點，共訓練 100000 次。

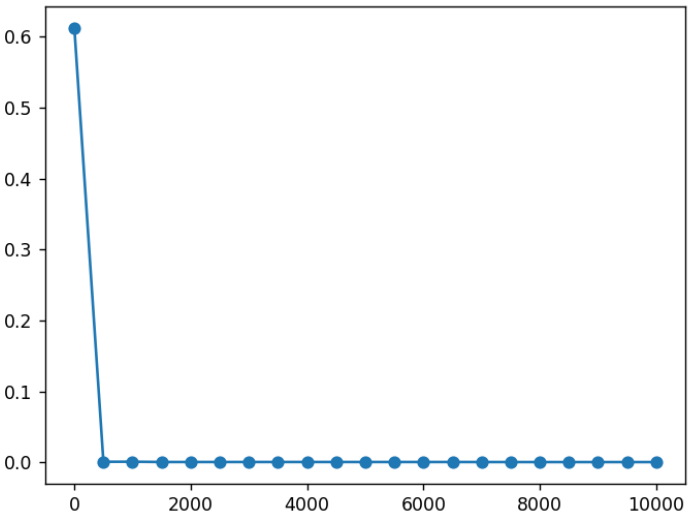
Learning rate: 0.01 (1/10 倍)



可以明顯看見 loss 的下降速度變慢了，並且有很長一段的時間卡在差不多的數值上，很可能是在那段接近水平的線期間，model 進入了 local minimum。類似於掉入了一個小低谷，到了後面才成功爬出來。

這也展現了 learning rate 太小的問題，model 很可能因為某個局部的極小值就停止更新。步長不夠讓他跨出那個局部極小值的低谷。

Learning rate: 1 (10 倍)



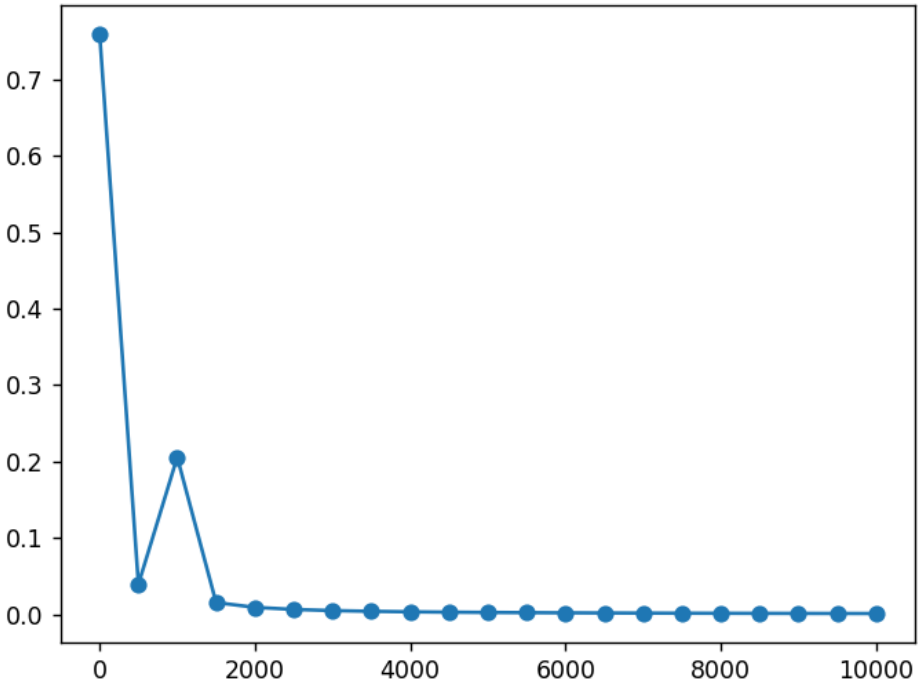
可以看到 loss 下降的速度比 0.1 時更快，但是他收斂的地方在 2.8e-4，相較於學習率 0.1 時的收斂值 3.3e-5 是稍大一些的。反映出如果學習率的步長設的太長的話，如果 global minimum 的低谷比較淺，那 model 就會跨不進去裡面，或是一跨進去裡面就會直接跳出來，所以學習率設太高，雖然訓練得比較快，但是有可能達不到 global minimum。

B. Try different numbers of hidden units

這裡使用 linear data 來做呈現，因為這樣的結果更明顯。我採用的 training data 與 test data 是同一批，所以探討 hidden unit 數量過多造成的 overfitting 是不實際的，因此這裡只探討 hidden unit 不足時的結果。

\

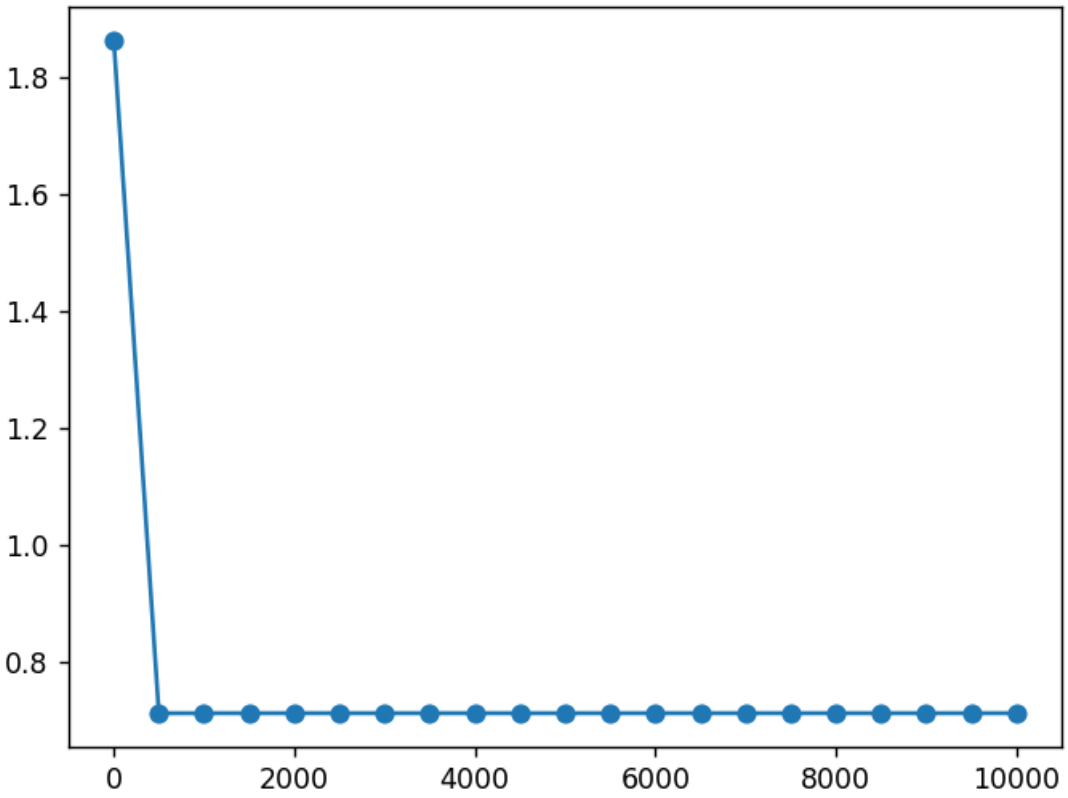
(1) hidden units: (1,1)



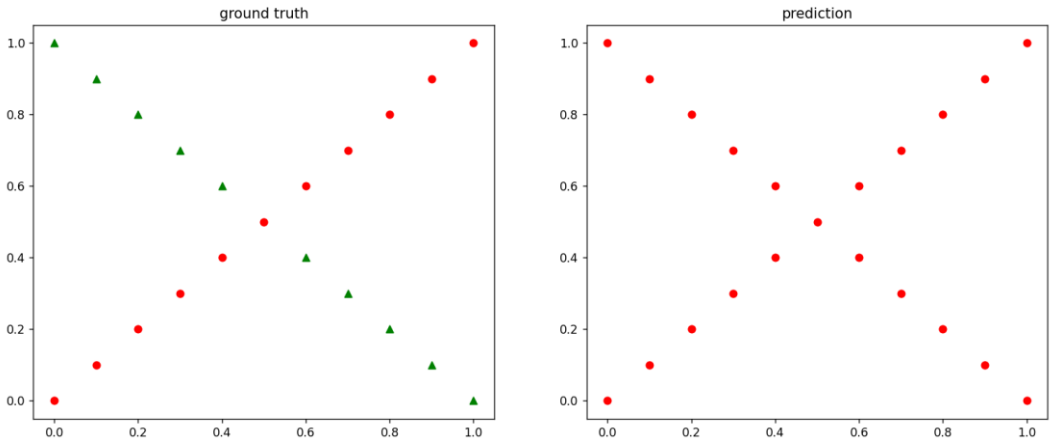
可以明顯看見訓練的速度變慢了，甚至中間有一段 loss 上升，可見當參數不足時，是很難在少量的更新得出比較好的結果的，而後面收斂的那段，收斂的 loss 值為 1e-3，明顯少於我設定的一般情況(10,10)的 7.3e-10，差了 10^7 倍。所以當 hidden unit 不足時，他是很難給出非常好的結果的(雖然在這個 case 中準確率仍是 100%)。

C. Try without activation functions

這裡把 2 個 hidden layer 的 activation 刪除並測試結果，使用的是 XOR data，因為 XOR 更加的 non-linear，而去掉 activation 就是讓 model 的 non-linear 能力下降。。



可以看見一下子收斂，但是收斂的 loss 值在 0.71，相較於標準的 3e-5 是差了好幾個量級。準度也只剩下 52%，從下述分布圖中可以看見，她幾乎就是在用亂猜的



從這個例子可以明確看出 activation 造成的 non-linear 效果有多大的影響，XOR 的這種極端線性不可分的資料，就會在沒有 activation 的狀況下直接炸掉。

5. Extra

A. Implement different optimizers.

原本使用的 optimizer 是 mini-batch Gradient Descent，而在此之上，我把它改成了 Stochastic Gradient Descent (SGD)，做法是在每個 epoch 之前，都先把整個 dataset 的順序打亂，再套上 mini-batch Gradient Descent。程式碼如下，多了把資料的 index 做 shuffle 的部分。

```
for epoch in range(n_epoch):
    indexs = np.arange(X_train.shape[0])
    indexs = np.random.shuffle(indexs)
    X_train = X_train[indexs][0]
    Y_train = Y_train[indexs][0]
    for batch in range( np.ceil(sample_size / batch_size).astype(int) ):
        inputs = X_train[batch*batch_size : min(sample_size, (batch+1)*batch_size), :]
        labels = Y_train[batch*batch_size : (batch+1)*batch_size, :]
        prediction = self.forward(inputs)
        loss = self.criterion(labels, prediction)
        self.backproapgation(inputs, labels, prediction)
```

Result:

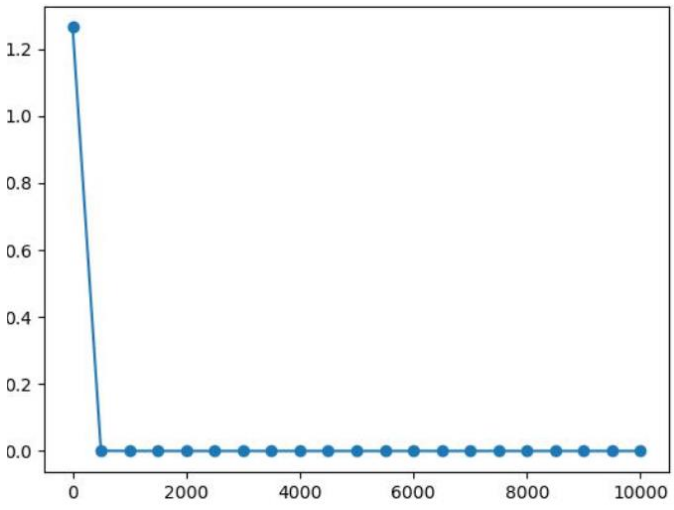
Linear data:

```
epoch: 10000/10000, loss: 6.790006897456736e-10
test, sample_size: 100, accuracy: 100.0%
```

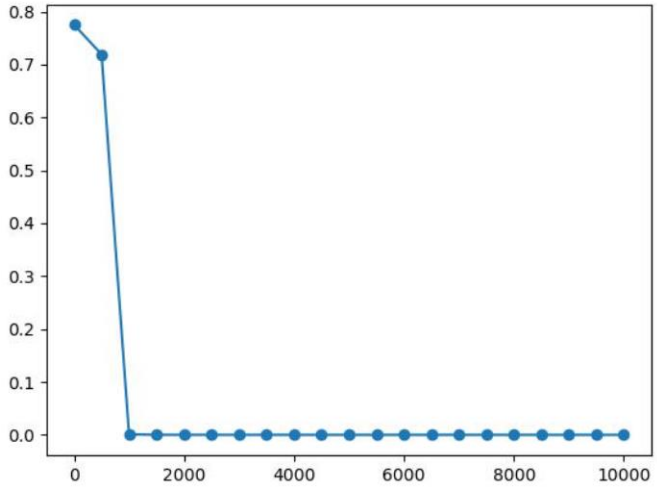
XOR data:

```
epoch: 10000/10000, loss: 1.8354394249620078e-06
test, sample_size: 21, accuracy: 100.0%
```

Linear data epoch-loss curve:



XOR data epoch-loss curve



從數據及各圖中可以看出，SGD 的效果在這個 case 會跟一般的 mini-batch GD 差不多，在 XOR 的線似乎有訓練的更快的跡象，但不是很明顯。不過如果換在更複雜的 case，SGD 應該會比 mini-batch GD 的效果更好，因為 batch 內數據的隨機排列組合讓 generalize 的能力更強。

B. Implement activation functions.

在此 lab 的基本神經網路架構中，採用 sigmoid 作為 hidden layer 的 activation function，不過在大部分的深度學習中，大家還是比較傾向於採用 ReLU 系列的 activation function，因為他的導數會是單純的 0, 1，可以避免梯度消失或爆炸的問題，也可以讓負數的數值直接被砍掉，相當於讓 bias 成為了神經元被激發的門檻，會讓整體的 non-linear 有足夠的強度。

這次實做中，我分別採用了 ReLU 及其延伸 Leaky ReLU:

(1) ReLU:

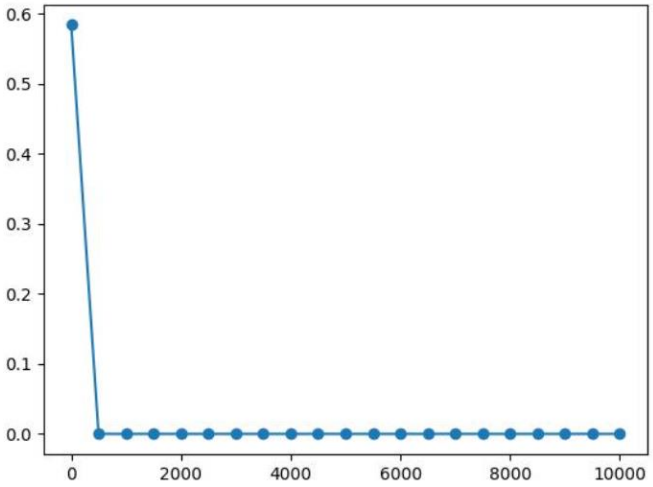
實現:

基本上就是對於整個 matrix 套 max(0,x)，即對於每個神經元分別套 ReLU

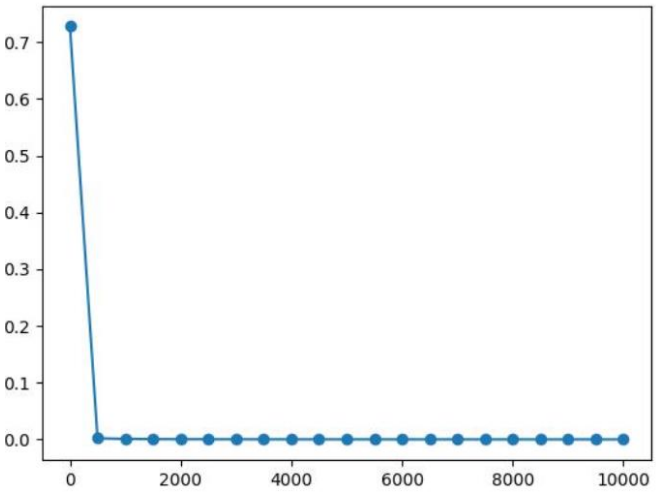
```python
def ReLU(x):
    '''
    ReLU function, additional activation funciton
    x.shape: batch_size * n
    '''
    return np.maximum(0,x)

def ReLU_derivative(x, grad_Y):
    '''
    Derivative of d Loss / d x
    formula: ReLu(x)
    x.shape: batch_size * n
    grad_Y.shape: batch_size * n
    ret.shape: batch_size * n
    '''
    return (x >= 0).astype(int) * grad_Y
```

Result:

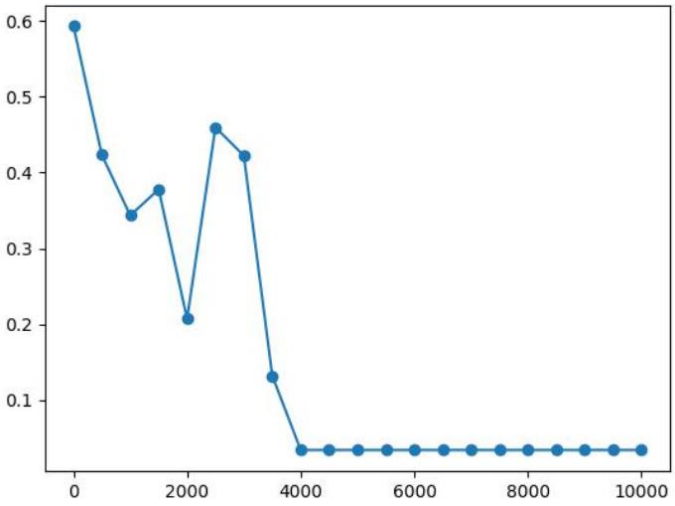
Linear Data:



XOR Data:



基本上得到的成效跟套 sigmoid 是差不多的，可能要在更深的神經網路上，才能比較看出 sigmoid 跟 ReLU 的差別，另外在實驗的過程中，我也有觀察到使用 ReLU 後壞掉的狀況，如圖:



中間有 loss 逆勢上升的狀況，讓優化變得很不穩定，我推測可能是 ReLU 的一個特性，因為出現負數後，梯度為 0，導致神經元被關閉，可能會比較難再次開啟他，就會導致 model 的複雜度下降，讓她沒辦法給出很好的結果，甚至變得更糟。因為觀察到了 ReLU 可能出現的不穩定性，所以我後面嘗試了更 soften 的 Leaky ReLU。

(1) Leaky ReLU:

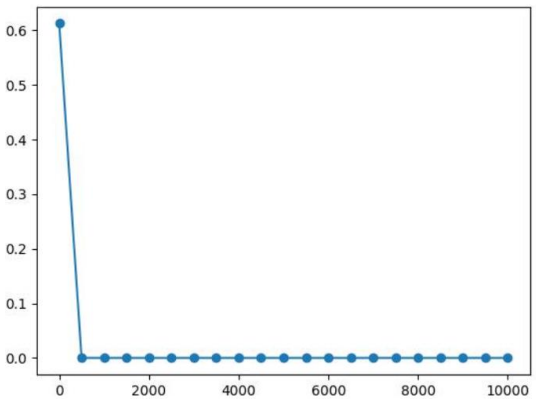
實現:

就是 ReLU 在負數時的輸出由 0 改成 alpha*x，alpha 設為 0.01

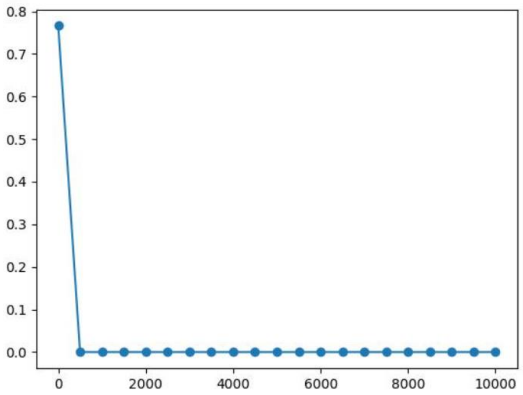
```python
def Leaky_ReLU(x, alpha = 0.01):
    '''
    ReLU function, additional activation funciton
    x.shape: batch_size * n
    '''
    return np.where(x>0, x, alpha * x)

def Leaky_ReLU_derivative(x, grad_Y, alpha=0.01):
    '''
    Derivative of d Loss / d x
    formula: Leakly_ReLu(x)
    x.shape: batch_size * n
    grad_Y.shape: batch_size * n
    ret.shape: batch_size * n
    '''
    return np.where(x>0, 1, alpha) * grad_Y
```

Result:

Linear Data



XOR Data:



得到的效果在這個 case 裡面仍是差不多，不過在我的多次實驗下，都沒有出現像是 ReLU 那樣不穩定的問題，Leaky ReLU 讓神經網路的訓練能夠保持穩定的梯度傳遞，就不會出現花好久時間訓練，結果就因為一點不穩定性爆炸的狀況。