DL_LAB1_Report

310512073 電控所 曾致崴

1. Introduction

   在這次的 LAB1 中主要是針對 Backpropagation 以及網路如何更新權重，並了解其中基本原理如何更接近目標，在一開始先對整體網路訓練的流程了解後，先架設 Feedforward network，並利用得到的 output 利用 Backpropagation 計算出各項參數的 gradient 值最後 update 權重，完成一次的訓練，而在訓練的參數上對於結果來說也是很重要的，例如 learning rate 的調控會影響整體模型在訓練時震盪的程度，以及好不好收斂，hidden layer 的 unit 要設多少，也會影響網路的收斂速度，若是太大或太小也會對網路造成不同的影響。

   整題程式執行方式

   執行程式環境 : Windows 11
   Source code 底下總共有三個 py 檔 分別為 main.py、nets.py、utiles.py，其中 main 才是主要執行的程式，執行程式會出現 acc 95%的兩個 case

2. Experiment setups
   a. Sigmoid functions

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

圖 1

   b. Neural network

   在本次的 lab 中我主要是使用 class 的方式去建置整體的 model，而其中有一些初始化的參數，如下圖 2 所示

   1. lr (learning rate) : 0.001
   2. epoch : 1000
   3. random_seed : 0 此參數是為了確保每一次 training 的結果相同，在 init_weight 中有定義
   4. hid_layer_size : 是指 hidden layer 中要用多少個 node

```python
class nn_net_func():
    def __init__(self, lr=0.001, epoch=1000, ground_truth=None, random_seed=0, num_classes=1, hidden_node = 5):
        self.lr = lr
        self.epoch = epoch
        self.random_seed = random_seed
        self.hid_layer_size = hidden_node
        self.ground_truth = ground_truth
        self.output_size = num_classes
        pass
```

圖 2

在下圖 3 中定義了 6 個參數w_1、w_2、w_3、b_1、b_2、b_3 的大小

w_1 : 2 x hidden layer node

w_2 : hidden layer node x hidden layer node

w_3 : hidden layer node x number of classes

b_1 : 1 x hidden layer node

b_2 : 1 x hidden layer node

b_3 : 1 x number of classes

```python
def init_weight(self, input):
    np.random.seed(self.random_seed)
    W1 = np.random.randn(self.hid_layer_size, input.shape[1])
    b1 = np.zeros([self.hid_layer_size, 1])
    W2 = np.random.randn(self.hid_layer_size, self.hid_layer_size)
    b2 = np.zeros([self.hid_layer_size, 1])
    W3 = np.random.randn(self.output_size, self.hid_layer_size)
    b3 = np.zeros([self.output_size, 1])
    parameters = {"W1": W1, "b1": b1, "W2": W2, "b2": b2, "W3": W3, "b3" : b3}
    return parameters
```

圖 3

在 Network 的設計上這次是以為架構主要是有 3 個 linear 層以及 3 個 activation function ReLU、Sigmoid、Sigmoid，如下圖 4 所示，圖中左邊為網路架構流程圖，右邊為網路架構示意圖

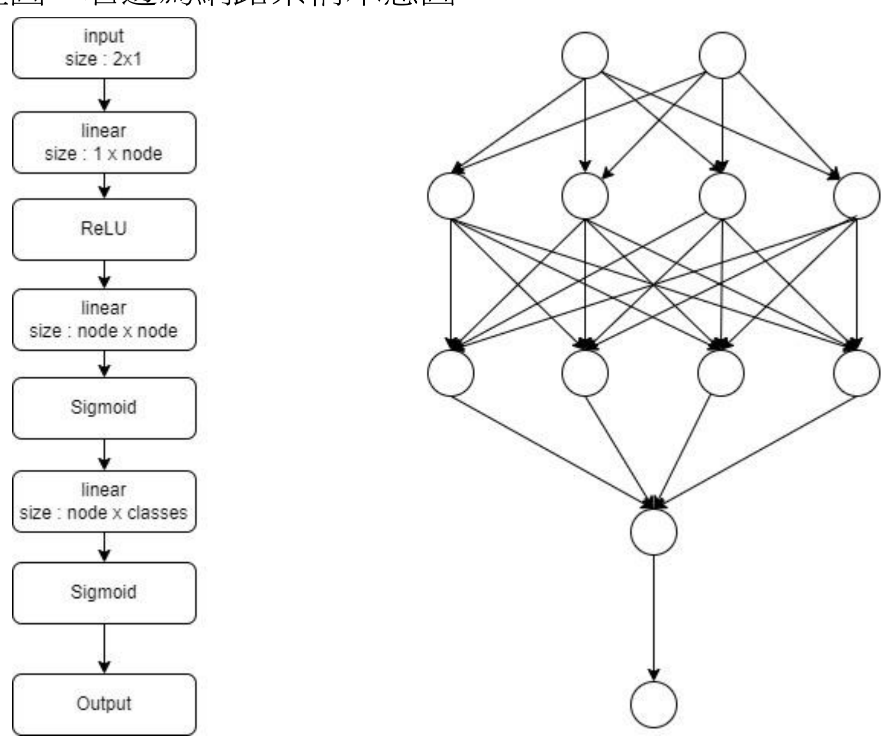


圖 4 網路架構示意圖

```python
def feedforward(self, x, parameters):
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    W3 = parameters["W3"]
    b3 = parameters["b3"]

    z1 = self.linear(x.reshape((2,1)), W1, b1)
    a1 = self.nn_ReLU(z1)
    z2 = self.linear(a1, W2, b2)
    a2 = self.nn_sigmoid(z2)
    z3 = self.linear(a2, W3, b3)
    a3 = self.nn_sigmoid(z3)

    tmp_parameters = {"z1": z1, "z2": z2, "z3": z3, "a1": a2, "a2": a2, "a3": a3}
    return a3, tmp_parameters
```

圖 5 feedforward network 程式圖

```python
def linear(self, x, w, b):
    return np.dot(w, x)+b


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


def nn_ReLU(self, x):
    return np.maximum(0, x)
```

圖 6 forward network 用到的 function

c. Backpropagation

Backpropagation 主要可以分為兩個部分，分別是 propagation、weight update

1. Propagation

因為 $z_i = w_i a_{i-1} + b_i$ 以及 $a_i = activation\_function(z_i)$，利用 chain rule 反著推，從最後一層是 sigmoid 開始，對a_3偏微分如下式

$$\frac{\partial L}{\partial a_3} = -\left(\frac{y}{a_3} - \frac{1-y}{1-a_3}\right)$$

接著對z_3偏微分(linear 層)

$$\frac{\partial L}{\partial z_3} = \frac{\partial L}{\partial a_3}\frac{\partial a_3}{\partial z_3} = \frac{\partial L}{\partial a_3}\left(\frac{1}{1+e^{-z_3}}\right)\left(1-\frac{1}{1+e^{-z_3}}\right)$$

$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial z_3}{a_2}^T/2$$

$$\frac{\partial L}{\partial b_3} = \frac{\partial L}{\partial z_3}/2$$

接著對a_2偏微分(Sigmoid 層)

$$\frac{\partial L}{\partial a_2} = \frac{\partial L}{\partial z_3}\frac{\partial z_3}{\partial a_2} = {w_3}^T\frac{\partial L}{\partial z_3}$$

接著對z_2偏微分(linear 層)

$$\frac{\partial L}{\partial z_2} = \frac{\partial L}{\partial a_2}\frac{\partial a_2}{\partial z_2} = \frac{\partial L}{\partial a_2}\left(\frac{1}{1+e^{-z_1}}\right)\left(1-\frac{1}{1+e^{-z_1}}\right)$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial z_2}{a_1}^T/2$$

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial z_2}/2$$

接著對a_1偏微分(ReLU 層)=>對小於 0 的直接給值=0 其餘一樣

$$\frac{\partial L}{\partial a_1} = {w_2}^T\frac{\partial L}{\partial z_2}$$

$$\frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial a_1}, and\ 數值小於\ 0\ 直接給\ 0$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial z_1}x^T/2$$

$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial z_1}/2$$

依照這個寫成程式如下圖 7 所示

```python
def backpropagation(self, parameters, tmp_parameters, x, y):

    W1 = parameters["W1"]
    W2 = parameters["W2"]
    W3 = parameters["W3"]
    A1 = tmp_parameters["a1"]
    A2 = tmp_parameters["a2"]
    A3 = tmp_parameters["a3"]
    Z1 = tmp_parameters["z1"]
    Z2 = tmp_parameters["z2"]
    Z3 = tmp_parameters["z3"]


    dA3 = - (np.divide(y, A3) - np.divide(1 - y, 1 - A3))

    temp_s = self.nn_sigmoid(Z3)
    dZ3 = dA3 * temp_s * (1-temp_s)        # Sigmoid (back propagation)

    dW3 = 1/2 * np.dot(dZ3, A2.T)
    db3 = 1/2 * np.sum(dZ3, axis=1, keepdims=True)
    dA2 = np.dot(W3.T,dZ3)

    # dA2 = - (np.divide(y, A2) - np.divide(1 - y, 1 - A2))

    temp_s = self.nn_sigmoid(dA2)
    dZ2 = dA2 * temp_s * (1-temp_s)        # Sigmoid (back propagation)

    dW2 = 1/2 * np.dot(dZ2, A1.T)
    db2 = 1/2 * np.sum(dZ2, axis=1, keepdims=True)
    dA1 = np.dot(W2.T,dZ2)

    # ReLU (back propagation)
    dZ1 = np.array(dA1, copy=True) # just converting dz to a correct object.
    dZ1[Z1 <= 0] = 0    # When z <= 0, you should set dz to 0 as well.
    # dZ1 = self.nn_ReLU(dA1)

    dW1 = 1/2 * np.dot(dZ1, x.reshape(1,2))
    db1 = 1/2 * np.sum(dZ1, axis=1, keepdims=True)
    grads = {"dW1": dW1, "db1": db1, "dW2": dW2, "db2": db2, "dW3": dW3, "db3": db3}
    return grads
```

圖 7 Backpropagation 程式圖

2. Weight update

   在 weight update 就是，原本的值 ”減” backpropagation 所得到的 gradient 各項參數值乘上 learning rate 對 weight 做更新，如下圖 8 所示

```python
def update_net(self, parameters, gradient_data):
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    W3 = parameters["W3"]
    b3 = parameters["b3"]

    dW1 = gradient_data["dW1"]
    db1 = gradient_data["db1"]
    dW2 = gradient_data["dW2"]
    db2 = gradient_data["db2"]
    dW3 = gradient_data["dW3"]
    db3 = gradient_data["db3"]

    W1 = W1 - self.lr*dW1
    b1 = b1 - self.lr*db1
    W2 = W2 - self.lr*dW2
    b2 = b2 - self.lr*db2
    W3 = W3 - self.lr*dW3
    b3 = b3 - self.lr*db3

    parameters = {"W1": W1, "b1": b1, "W2": W2, "b2": b2, "W3" : W3, "b3" : b3}

    return parameters
```

圖 8 weight update 程式圖

3. Results of your testing
    a. Screenshot and comparison figure

    Task 1 predict linear label (lr = 0.01, epoch=500, random_seed=0, hidden unit = 10)

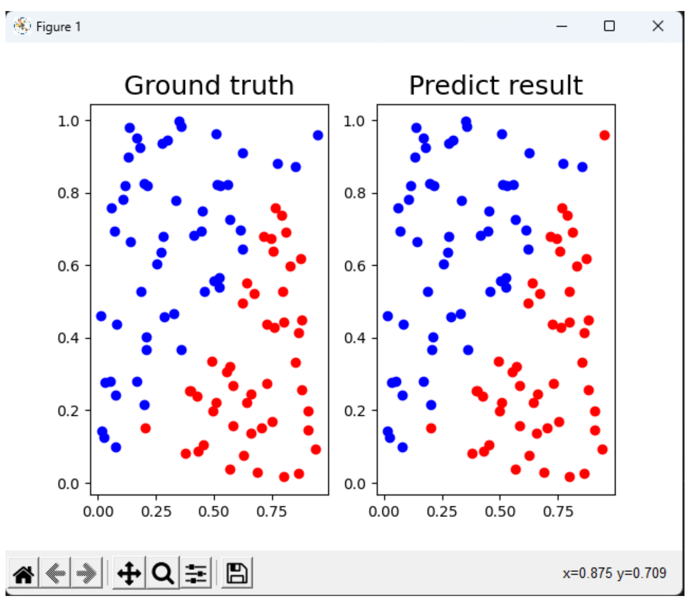


圖 9

Task 2 predict XOR label (lr = 0.01, epoch=300, random_seed=0, hidden unit =100)

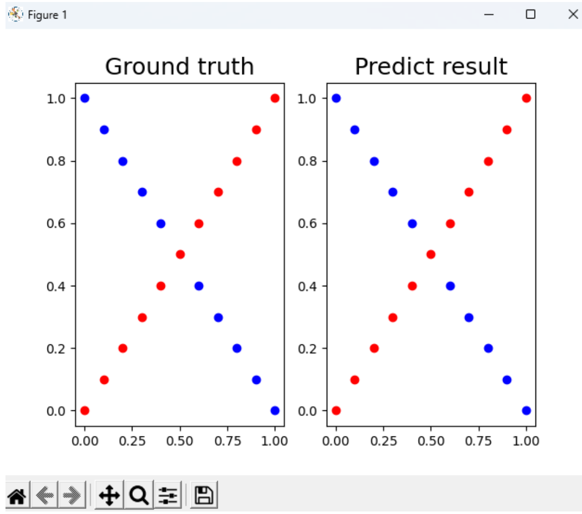


圖 10

b. Accuracy of prediction

Task 1 predict linear label

```
epoch = 359 ,loss =  0.0011549600386722742
epoch = 360 ,loss =  0.0011441216564357965
epoch = 361 ,loss =  0.0011333877259094972
epoch = 362 ,loss =  0.0011227563486674265
epoch = 363 ,loss =  0.001112172979636004
epoch = 364 ,loss =  0.0011017146763083617
epoch = 365 ,loss =  0.001091373301184203
epoch = 366 ,loss =  0.0010811348496876797
epoch = 367 ,loss =  0.0010709939394650133
epoch = 368 ,loss =  0.0010609479671185816
epoch = 369 ,loss =  0.0010509952337103467
epoch = 370 ,loss =  0.0010410881057247045
epoch = 371 ,loss =  0.0010312937403283064
epoch = 372 ,loss =  0.0010216050826767835
epoch = 373 ,loss =  0.001012009832958538
epoch = 374 ,loss =  0.0010025032486648006
epoch = 375 ,loss =  0.0009930830200855921
acc =  99.0 %
```

圖 11

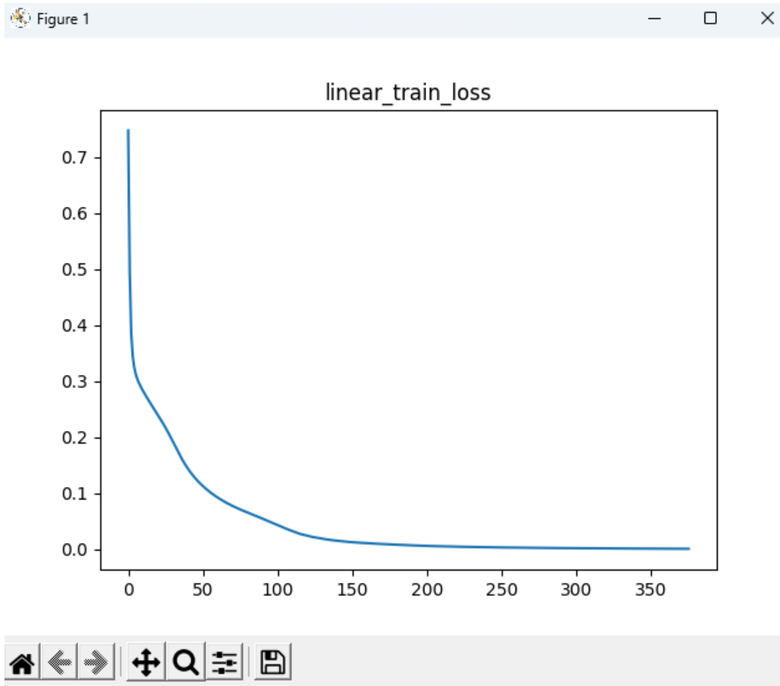
Task 2 predict XOR label

```
epoch = 244 ,loss =  0.0013326222929338067
epoch = 245 ,loss =  0.0013014641108270956
epoch = 246 ,loss =  0.0012771217197207003
epoch = 247 ,loss =  0.0012454464405235257
epoch = 248 ,loss =  0.001221298269387325
epoch = 249 ,loss =  0.0012001369887528683
epoch = 250 ,loss =  0.0011747658189669693
epoch = 251 ,loss =  0.001159592127467996
epoch = 252 ,loss =  0.0011365670389775389
epoch = 253 ,loss =  0.0011120665323200388
epoch = 254 ,loss =  0.0010971162585065944
epoch = 255 ,loss =  0.001074176245228789
epoch = 256 ,loss =  0.0010592013950156895
epoch = 257 ,loss =  0.0010420225647818336
epoch = 258 ,loss =  0.0010246184171822375
epoch = 259 ,loss =  0.0010064778947058617
epoch = 260 ,loss =  0.000989913869392099
acc =  100.0 %
```

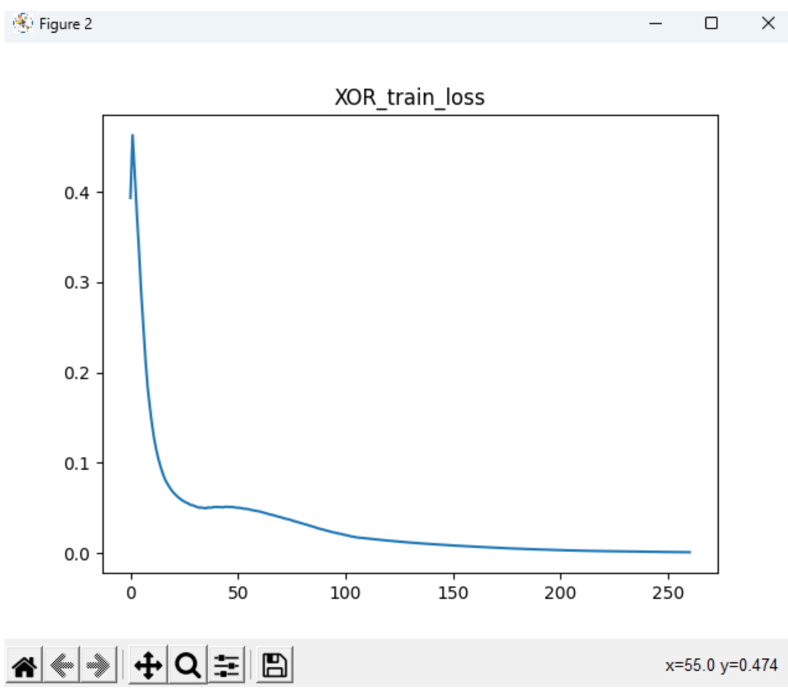
圖 12

c. Learning curve

Task 1 predict linear label



Task 2 predict XOR label



d. Others

```python
def train(self, input):
    x_data, y_data = input
    parameters = self.init_weight(input=x_data)
    stop_flag = 0
    loss_list = []
    for i in range(self.epoch):
        if stop_flag:
            break
        loss = 0
        for idx, x in enumerate(x_data):
            y = y_data[idx]
            a3, tmp_parameters = self.feedforward(x=x, parameters=parameters)
            loss = self.cost_func(a3, y)
            grads = self.backpropagation(parameters=parameters, tmp_parameters=tmp_parameters, x=x, y=y)
            parameters = self.update_net(parameters=parameters, gradient_data=grads)
            del y, a3, tmp_parameters, grads
        # if i%50==0:
        print("epoch = "+str(i)+" ,loss = ", str(loss))
        loss_list.append(loss)
        if loss<0.001:
            stop_flag = 1
            break
```

圖 13 train function

在程式的底下有加上一個限制條件當 loss 小於 0.001 時就會跳掉 training，因為當 loss 小於一定值時去計算 backpropagation 時會使分母太小造成結果變成 NaN 的情況，所以為了避免這樣的狀況發生，在此有加上保護機制

4. Discussion

a. Different learning rate

目前固定 hidden layer unit = 10，epoch = 500，learning rate 以 10 倍做調整

Task 1 predict linear label

hidden unit	epochs	Learning rate	Acc	End epoch
10	500	0.0001	59%	500
10	500	0.001	91%	500
10	500	0.01	99%	300
10	500	0.1	99%	25

在這個 task 底下可以看到 learning rate 變大整體收斂的速度也變快，但超過包護機制所設定 threshold，就會開始發散

Task 2 predict XOR label

hidden unit	epochs	Learning rate	Acc	End epoch
10	500	0.0001	47.62%	500
10	500	0.001	80.96%	500
10	500	0.01	85.71%	500
10	500	0.1	57.14%	500

在這個 task 底下，可以明顯的看到 learning rate 越大，有先變好的趨勢但最後是整個發散掉的情況，在 train loss 上可以明顯的看到 loss 到大概 100 epochs 時有收斂，但後續就發散掉

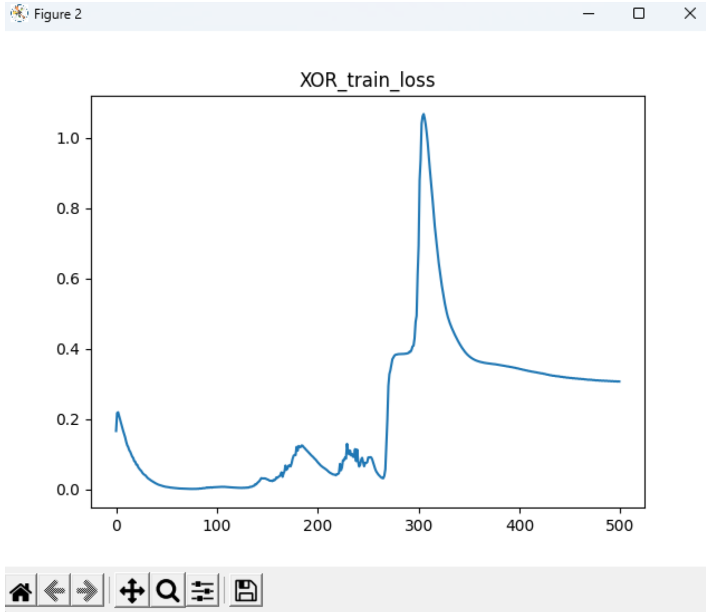


圖 14 loss 沒收斂圖

b. Different numbers of hidden units

Task 1 predict linear label

hidden unit	epochs	Learning rate	Acc	End epoch
1	500	0.01	63%	500
10	500	0.01	100%	330
50	500	0.01	99%	37
100	500	0.01	98%	57

在 task 1 上的表現，與前一個章節調整 lr 得到相同的結論，當 hidden unit 越多，網路會越快收斂，但同時當今天超過保護機制的狀況會造成發散 nan 的情況

Task 2 predict XOR label

hidden unit	epochs	Learning rate	Acc	End epoch
1	500	0.01	47%	500
10	500	0.01	85.71%	500
50	500	0.01	90.48%	500
100	500	0.01	100%	260

在 hidden layer unit = 1 的情況下，整體的模型因為不構 complex 所以在一開始就發散掉，如下圖所示，當 hidden unit 開始變多後模型才開始會收斂的比較好

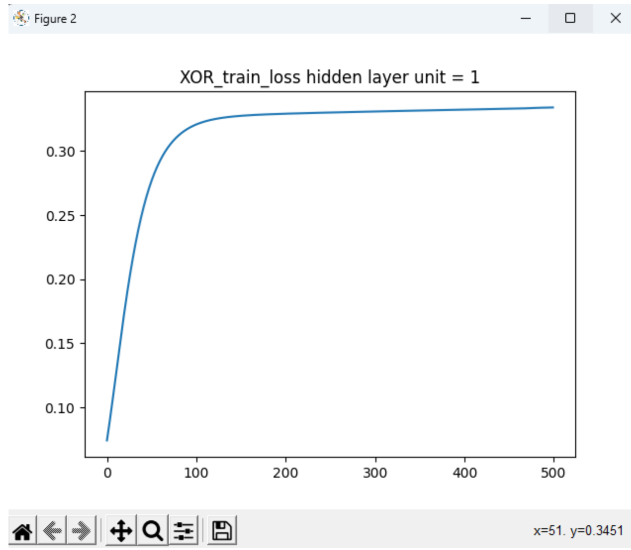


圖 15 hidden unit = 1 loss 圖

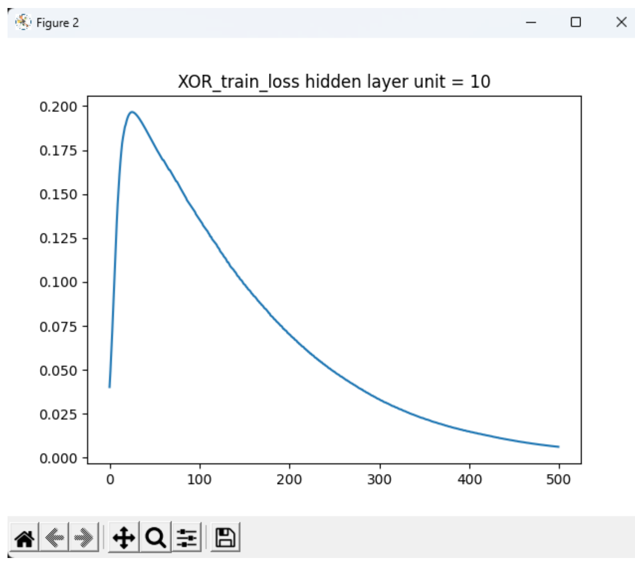


圖 16 hidden unit = 10 loss 圖

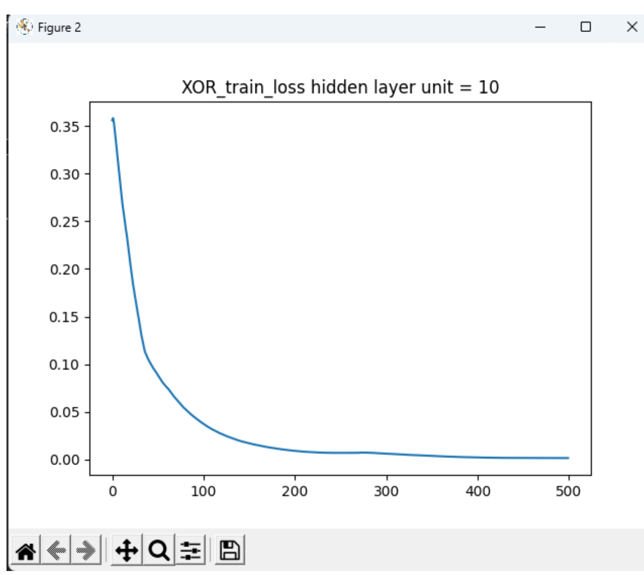


圖 17 hidden unit = 50 loss 圖

c. Without activation functions

5. Extra
   a. Implement different optimizers
      在 cost function 上我使用的是 cross entropy

```
def cost_func(self, output, y):
    return -(1/2)*( np.sum( (y*np.log(output).T) + ( (1-y)*(np.log(1-output).T) ) ) )
```

   b. Implement different activation functions
      ReLU

```
def nn_ReLU(self, x):
    return np.maximum(0, x)
```