深度學習Lab1 Report
309553012 黃建洲

I. Introduction

這一次的作業讓我們自行手刻一個簡單的neural network作為分類器。基本上是input/output layer加上兩層hidden layer的架構，並實作神經網路資料forward passing和back propagation，以及根據前兩者所得到的forward gradient和backward gradient合併計算出gradient後進行網路權重的更新。
測試資料為助教所提供的線性資料分布以及XOR資料分布。
使用語言為python，不使用pytorch等現有深度學習package，並使用CPU進行訓練。

II. Experiment Setup

A. Sigmoid functions:

由於輸入為matrix形式，故使用numpy進行運算

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

# y = sigmoid(x)
def derivative_sigmoid(y):
    return np.multiply(y, 1.0 - y)
```

B. Neural Network

神經網路架設的部分我學習常見的模板分為兩部分建立。

第一部分首先定義一個layer的元素: (給定該layer的input unit數量以及output unit數量)

1. 初始化: 根據layer的input unit數量以及output unit數量初始化一個極小的權重值。例如若input unit = 2, output unit = 4，則權重值矩陣大小為(2+1) * 4，所加上的1代表bias。
2. Forward: 將內層的input陣列都補上一個1後(為了對齊weight加上的bias項)，將input矩陣與weight矩陣做矩陣乘法再用activation function計算得到該layer的output項
3. backward(back propagation): 根據wiki所提供的back propagation算法如下

$$\boxed{\boxed{\frac{dC}{da^L}\cdot (f^L)'}\cdot W^L}\cdot (f^{L-1})'\cdot W^{L-1}\cdots (f^1)'\cdot W^1.$$

其中紅色框框的部分代表第一層的backward gradient，左項為derivative loss，右項則為derivative activation function。而綠色框框我則解釋為要feed給上一層的derivative loss項，是將backward gradient與weight的轉置矩陣(為了使矩陣乘法合法)相乘。
以題目給予的基礎layer sizes[2,4,4,1]為例，在output layer開始進行backward時，derivative loss matrix size為N x 1，derivative action function matrix size為N x 1，作element wise乘法後為N x 1。而weight

matrix size為2 x 1(因微分去除bias項), 轉置後為1x2得feed backward之derivative loss matrix為 N x 2。依此類推後皆合法。

4. update: 給予指定learning rate, 並利用在forward function中得到的forward gradient與backward function中得到的backward gradient, 相乘後算出total gradient, 並以learning rate為幅度對權重進行調整。

第二部分定義整個網路的流程:

1. 初始化: 根據指定的layer sizes來進行每一層的初始化。比如題目所給定的[2,4,4,1], 我們需要以layer(input size, output size)的格式產生layer(2,4), layer(4,4), layer(4,1)。
2. forward: 給定初始輸入值後，送入第一層layer計算，並依序將每一層的output輸入給下一層的input, 直到最後一層算出整個網路的output為止。
3. backward: 給定derivative loss值後，從最後一層layer開始進行backward, 並將每一次算出的derivative loss值送入上一層進行計算，直到第一層計算完成為止。
4. update: 單純在算出gradient(每一個layer自行暫存)後讓每一層layer進行一次update
5. Train: 將整個訓練的流程進行整理, 給定Max epoch數後以及loss threshold後, 每個epoch依序讓網路進行forward, backward, 以及update的流程, 並計算該epoch的loss value。當loss value小於loss threshold或是epoch數大於max epoch數後訓練結束。

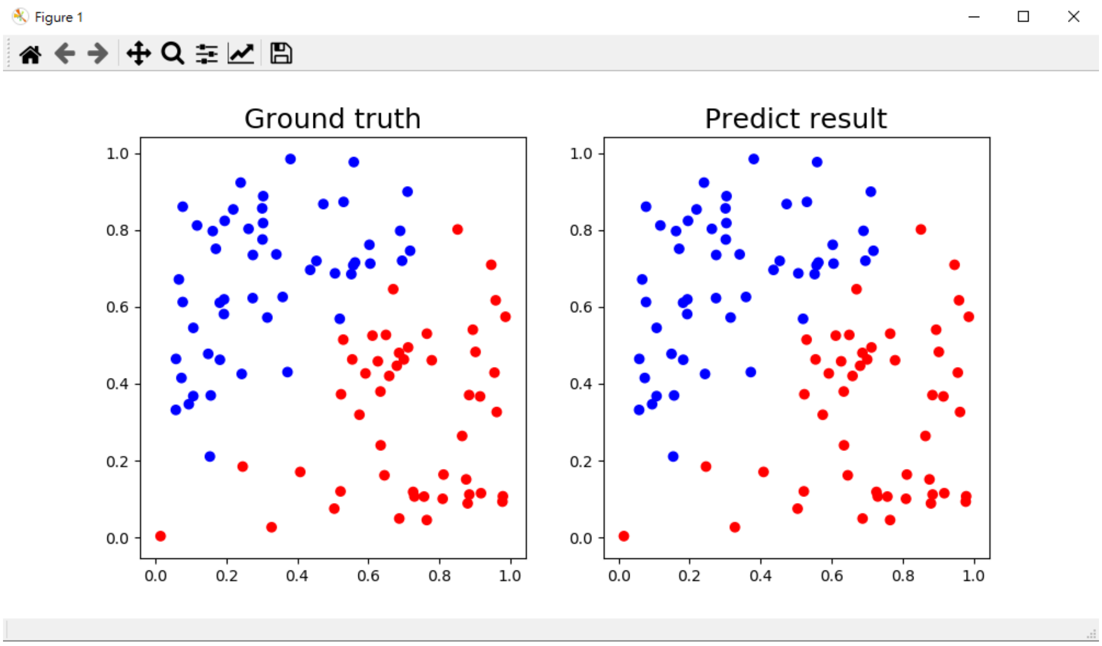
III. Result of your testing

這邊的epoch都以1000為單位

(1) 線性分布資料:
(2) 訓練epoch數: 115000
(3) activation function: sigmoid
(4) Accuracy: 100%
(5) optimizer: MSE
(6) loss threshold: 0.01
(7) learning rate = 0.001

- ## 線性分布資料集

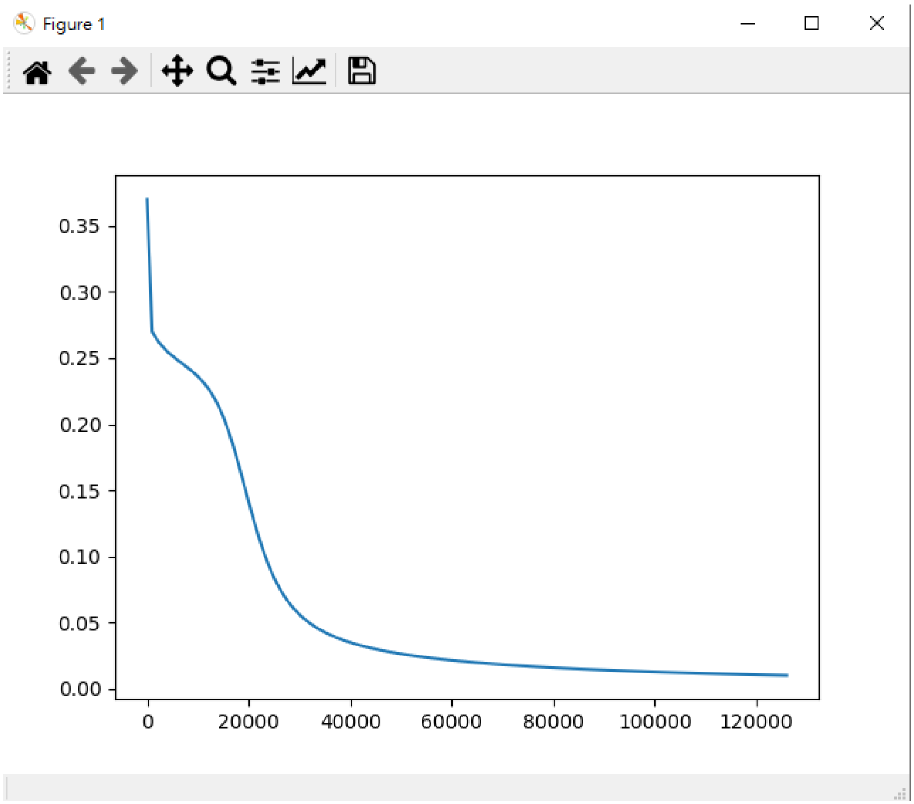
1. 比較圖



2. epoch與loss的對應資料

```
C:\Windows\system32\cmd.exe - python  DL_lab1.py
epoch: 88000   loss: 0.012444081447673528
epoch: 89000   loss: 0.012332335155110214
epoch: 90000   loss: 0.012222733033558402
epoch: 91000   loss: 0.01211519721787995
epoch: 92000   loss: 0.012009653910244502
epoch: 93000   loss: 0.011906033111999172
epoch: 94000   loss: 0.01180426837675916
epoch: 95000   loss: 0.01170429658275958
epoch: 96000   loss: 0.01160605772271695
epoch: 97000   loss: 0.011509494709630228
epoch: 98000   loss: 0.011414553197113297
epoch: 99000   loss: 0.01132118141299918
epoch: 100000   loss: 0.011229330005024685
epoch: 101000   loss: 0.011138951897721253
epoch: 102000   loss: 0.011050002159324888
epoch: 103000   loss: 0.01096243787812416
epoch: 104000   loss: 0.010876218047328181
epoch: 105000   loss: 0.010791303457817774
epoch: 106000   loss: 0.010707656598144488
epoch: 107000   loss: 0.010625241561207302
epoch: 108000   loss: 0.010544023957088047
epoch: 109000   loss: 0.010463970831572623
epoch: 110000   loss: 0.01038505058992567
epoch: 111000   loss: 0.010307232925524877
epoch: 112000   loss: 0.010230488752993499
epoch: 113000   loss: 0.010154790145501254
epoch: 114000   loss: 0.010080110275930613
epoch: 115000   loss: 0.010006423361631258
```

3. epoch與loss的曲線圖



4. prediction result

```
[[0.00663376]
 [0.00506767]
 [0.0026551 ]
 [0.01800357]
 [0.99081819]
 [0.99352481]
 [0.00753079]
 [0.38513491]
 [0.98284819]
 [0.99637966]
 [0.00254602]
 [0.00272408]
 [0.00235793]
 [0.99688905]
 [0.00477249]
 [0.99580153]
 [0.98439428]
 [0.95459407]
 [0.00265384]
 [0.68578283]
 [0.99460084]
 [0.01451278]
 [0.99585349]
 [0.00279724]
 [0.99663519]
 [0.73271091]
 [0.99195977]
 [0.98435304]
 [0.96795217]
 [0.002139  ]
 [0.9939785 ]
 [0.99431064]
 [0.01550305]
 [0.0020171 ]
 [0.26531624]
 [0.0023194 ]
 [0.03301409]
 [0.99240127]
 [0.00736737]
 [0.99436671]
 [0.98632919]
 [0.99529996]
 [0.00335398]
 [0.37702337]
 [0.02105504]
 [0.03360579]
 [0.29705883]
 [0.9963015 ]
 [0.01278765]
 [0.99507411]
 [0.00258661]
 [0.99328216]
 [0.00219989]
 [0.96074925]
 [0.98667652]
 [0.00417628]
 [0.99380916]
 [0.97202288]
 [0.99641823]
 [0.99660829]
 [0.00795012]
 [0.98111947]
 [0.0035778 ]
 [0.13566186]
 [0.93790804]
 [0.99038233]
 [0.00342514]
 [0.12410685]
 [0.00491595]
 [0.00199774]
 [0.01513649]
 [0.66279223]
 [0.01881173]
 [0.95949494]
```

C:\Windows\system32\cmd.exe

```
 [0.99663519]
 [0.73271091]
 [0.99195977]
 [0.98435304]
 [0.96795217]
 [0.002139  ]
 [0.9939785 ]
 [0.99431064]
 [0.01550305]
 [0.0020171 ]
 [0.26531624]
 [0.0023194 ]
 [0.03301409]
 [0.99240127]
 [0.00736737]
 [0.99436671]
 [0.98632919]
 [0.99529996]
 [0.00335398]
 [0.37702337]
 [0.02105504]
 [0.03360579]
 [0.29705883]
 [0.9963015 ]
 [0.01278765]
 [0.99507411]
 [0.00258661]
 [0.99328216]
 [0.00219989]
 [0.96074925]
 [0.98667652]
 [0.00417628]
 [0.99380916]
 [0.97202288]
 [0.99641823]
 [0.99660829]
 [0.00795012]
 [0.98111947]
 [0.0035778 ]
 [0.13566186]
 [0.93790804]
 [0.99038233]
 [0.00342514]
 [0.12410685]
 [0.00491595]
 [0.00199774]
 [0.01513649]
 [0.66279223]
 [0.01881173]
 [0.95949494]
 [0.03930079]
 [0.01177177]
 [0.99617042]
 [0.99443888]
 [0.98584451]
 [0.99686588]
 [0.98612347]
 [0.00315208]
 [0.00256404]
 [0.90956309]
 [0.01617071]
 [0.95291382]
 [0.10704857]
 [0.99651885]
 [0.99664428]
 [0.00216037]
 [0.97491221]
 [0.73414702]
 [0.99602613]
 [0.00465585]
 [0.06865236]
 [0.0122853 ]
 [0.91771554]
 [0.00232574]
 [0.67108737]
 [0.99716828]]
```

- XOR 資料集:

  圖片:

  (1) 訓練epoch數: 269000

  (2) activation function: sigmoid

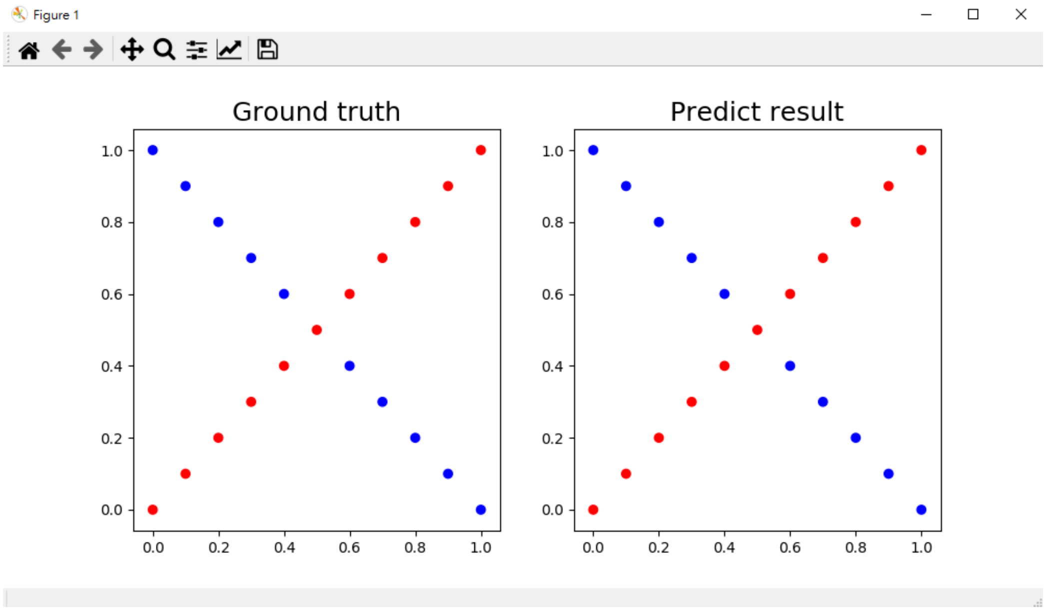
  (3) Accuracy: 100%

  (4) optimizer: MSE

  (5) loss threshold: 0.01

  (6) learning rate = 0.001

1. 比較圖

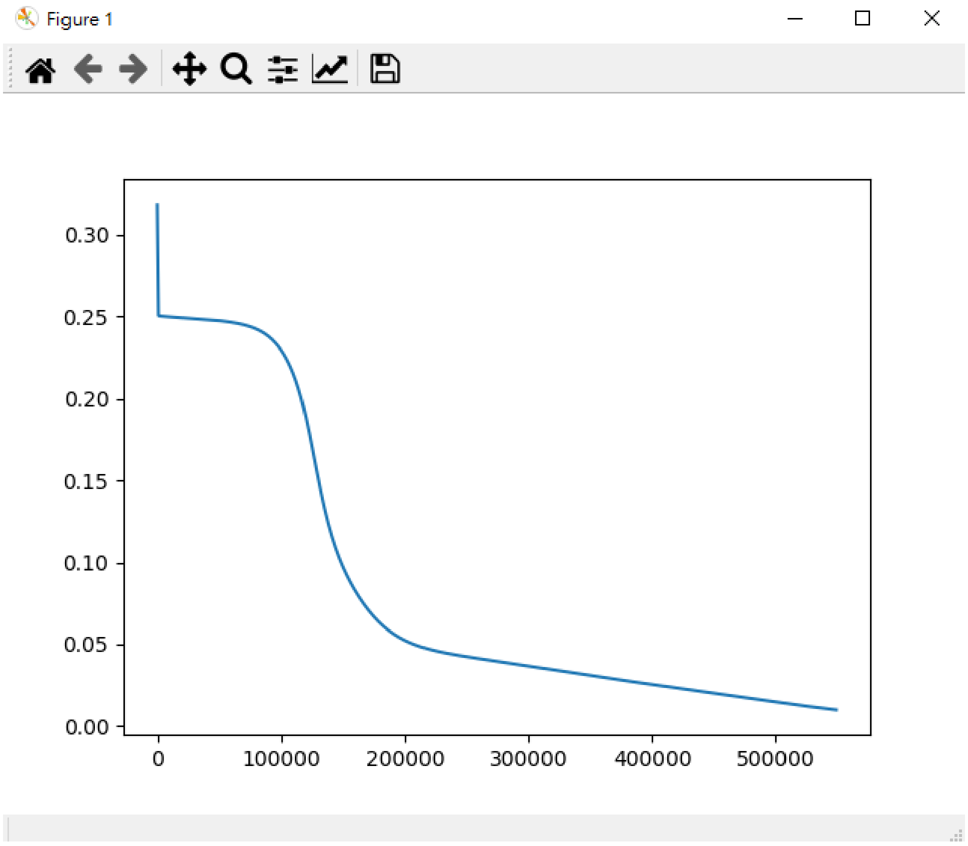


2. epoch與loss的對應資料

C:\Windows\system32\cmd.exe

```
epoch: 163000  loss: 0.024962279938859928
epoch: 164000  loss: 0.02410245287974303
epoch: 165000  loss: 0.023275548248137757
epoch: 166000  loss: 0.02248066632235214
epoch: 167000  loss: 0.021716886217066127
epoch: 168000  loss: 0.020983269727865544
epoch: 169000  loss: 0.020278865566176384
epoch: 170000  loss: 0.019602713785302917
epoch: 171000  loss: 0.018953850233322012
epoch: 172000  loss: 0.018331310901340058
epoch: 173000  loss: 0.01773413606546838
epoch: 174000  loss: 0.0171613741474445
epoch: 175000  loss: 0.016612085241938218
epoch: 176000  loss: 0.01608534427823925
epoch: 177000  loss: 0.015580243800335937
epoch: 178000  loss: 0.015095896362600772
epoch: 179000  loss: 0.01463143654867093
epoch: 180000  loss: 0.014186022628980652
epoch: 181000  loss: 0.013758837878098536
epoch: 182000  loss: 0.013349091576882714
epoch: 183000  loss: 0.01295601972679944
epoch: 184000  loss: 0.012578885504856445
epoch: 185000  loss: 0.012216979487727228
epoch: 186000  loss: 0.011869619673032976
epoch: 187000  loss: 0.011536151324576829
epoch: 188000  loss: 0.01121594666677882
epoch: 189000  loss: 0.010908404451752018
epoch: 190000  loss: 0.01061294942051459
epoch: 191000  loss: 0.010329031677826731
epoch: 192000  loss: 0.010056125998139598
```

3. epoch與loss的曲線圖



4. Prediction result

```
[[0.06087425]
 [0.98639577]
 [0.07151044]
 [0.98545828]
 [0.08248942]
 [0.98277603]
 [0.09209411]
 [0.96984117]
 [0.09903851]
 [0.76426974]
 [0.10292629]
 [0.1041142 ]
 [0.76536205]
 [0.10330222]
 [0.96243705]
 [0.10119906]
 [0.97642153]
 [0.09836667]
 [0.97949558]
 [0.09519358]
 [0.98048948]]
```

IV. Discussion

1. 嘗試不同的learning rate:

(a) lr = 0.05 →

	線性分布	XOR
(i) epoch數量:	21000	55000
(ii) Accuracy:	100%	100%

(b) lr = 0.5 →

	線性分布	XOR
(i) epoch數量:	1000	9000
(ii) Accuracy:	99%	100%

(c) lr = 5 →

	線性分布	XOR
(i) epoch數量:	0	0
(ii) Accuracy:	100%	100%

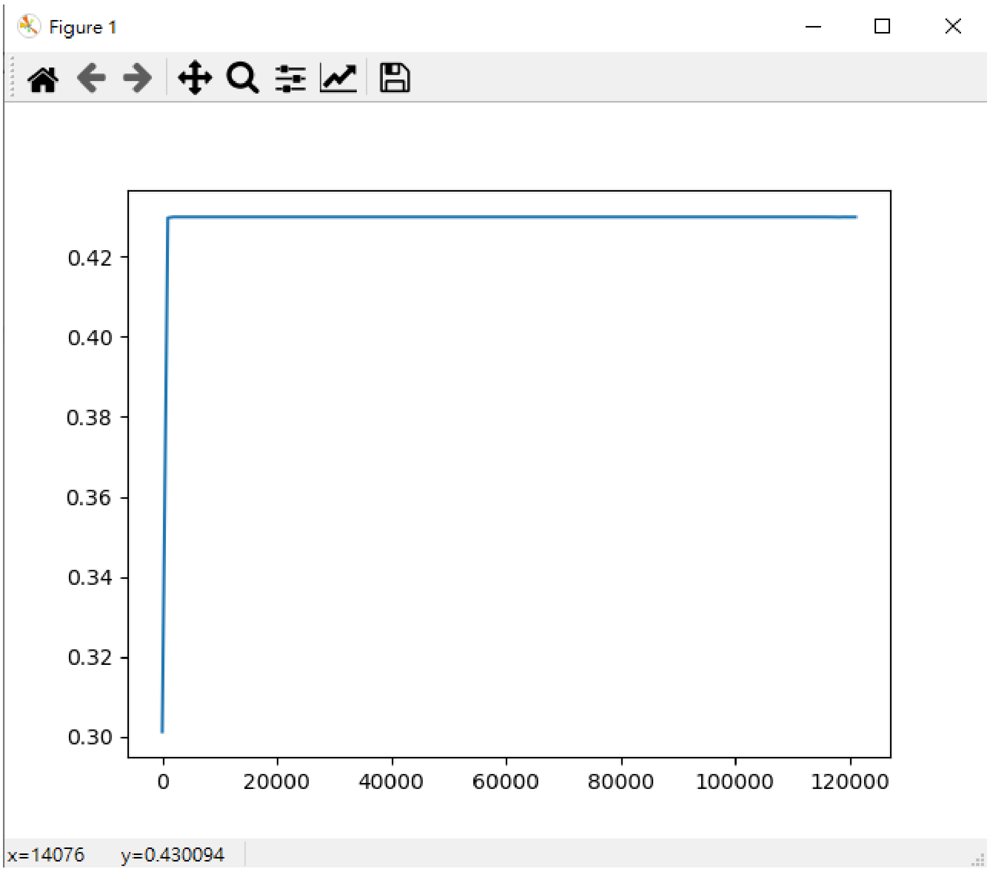
(d) lr = 50

	線性分布	XOR
(i) epoch數量:	121000	1000000
(ii) Accuracy:	98%	47.6%

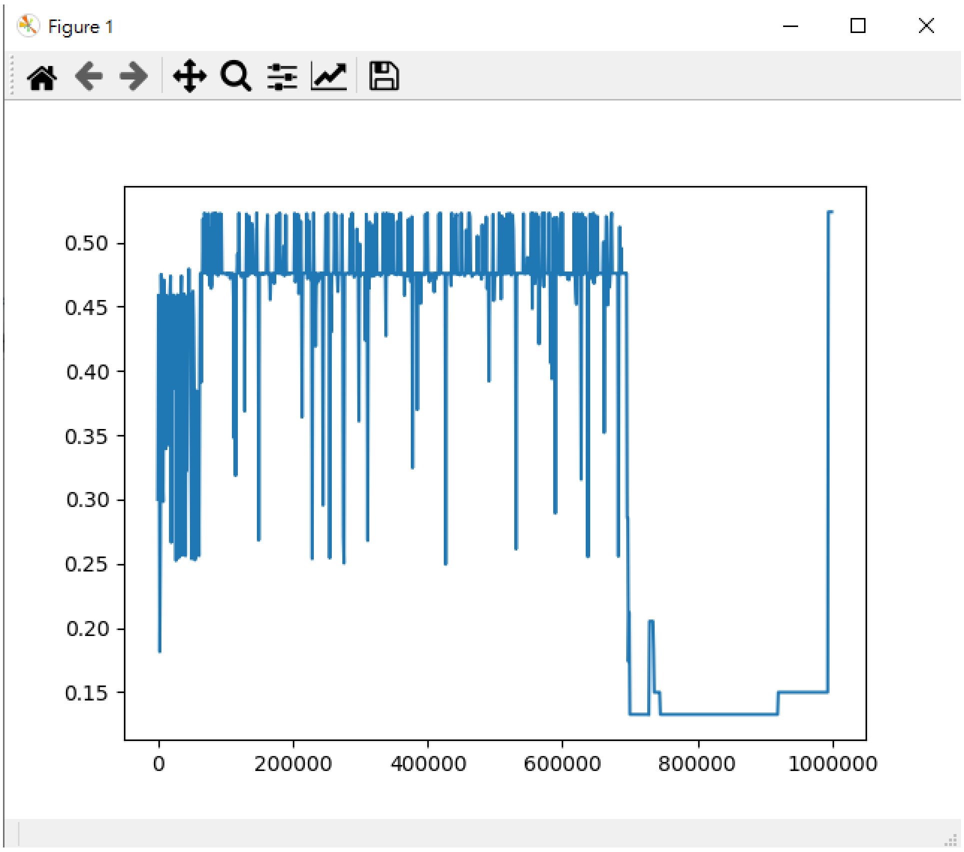
在lr = 50時記錄到震盪的現象

線性分布的epoch-loss圖(機率發生)

(在120000的時候loss急速下降到threshold以下)



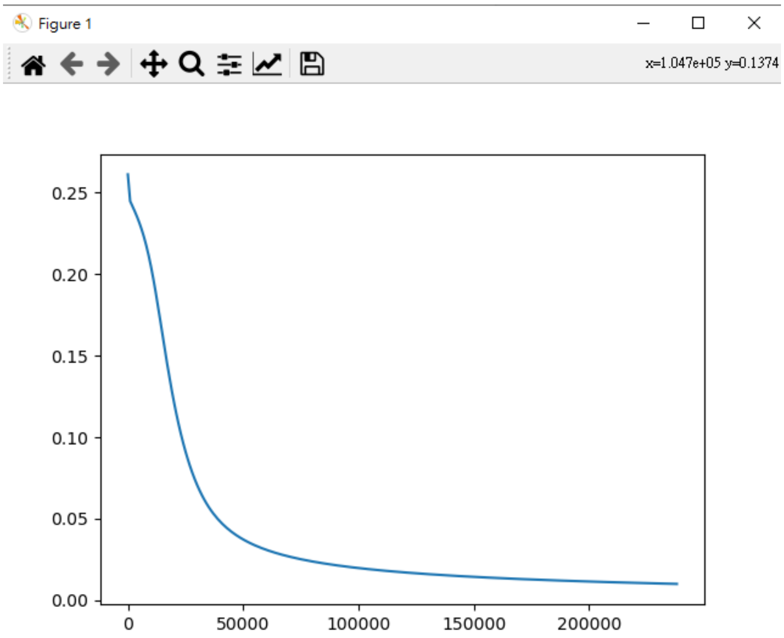
XOR的epoch-loss圖



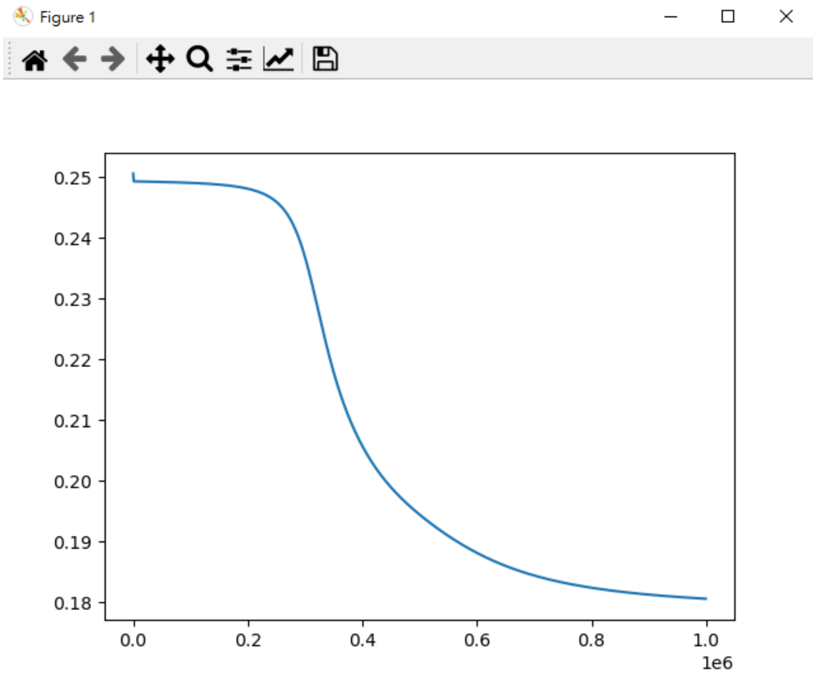
由於training是在方程式中尋找loss的最低點，因此learning rate越小，在方程式中開始震盪的點loss值會越低。這次的作業中資料集都不算太難，甚至到learning rate = 5，loss都還可以正常的降到0.01以下。直到learning rate = 50，在XOR的資料中才可以比較顯著的看到loss震盪的現象。

2. Try different number of hidden units

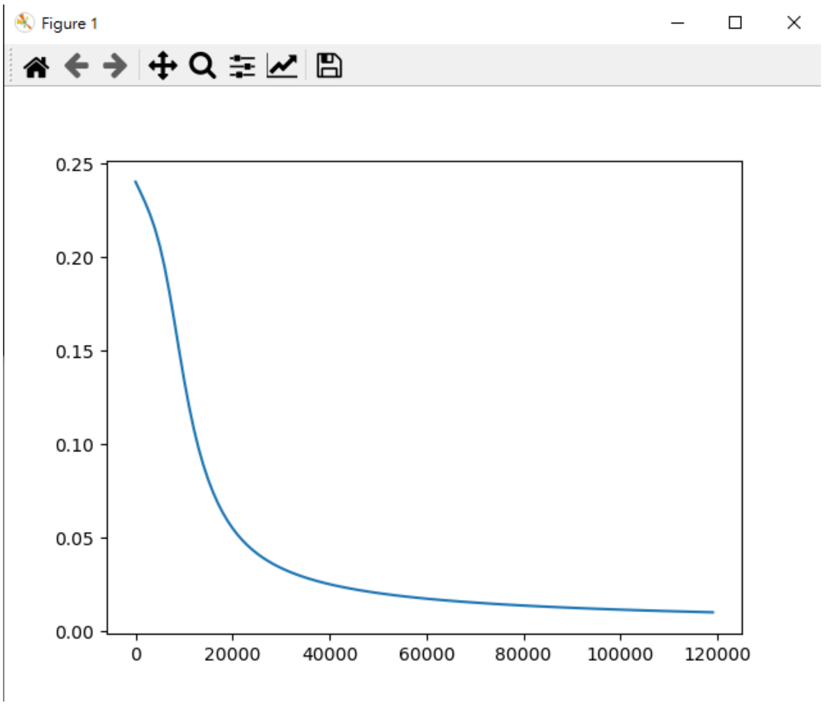
(a) layers size = [2,2,2,1], learning rate = 0.01, loss_threshold = 0.01
線性分布資料 → accuracy = 100%, epoch = 238000
loss-epoch 曲線圖



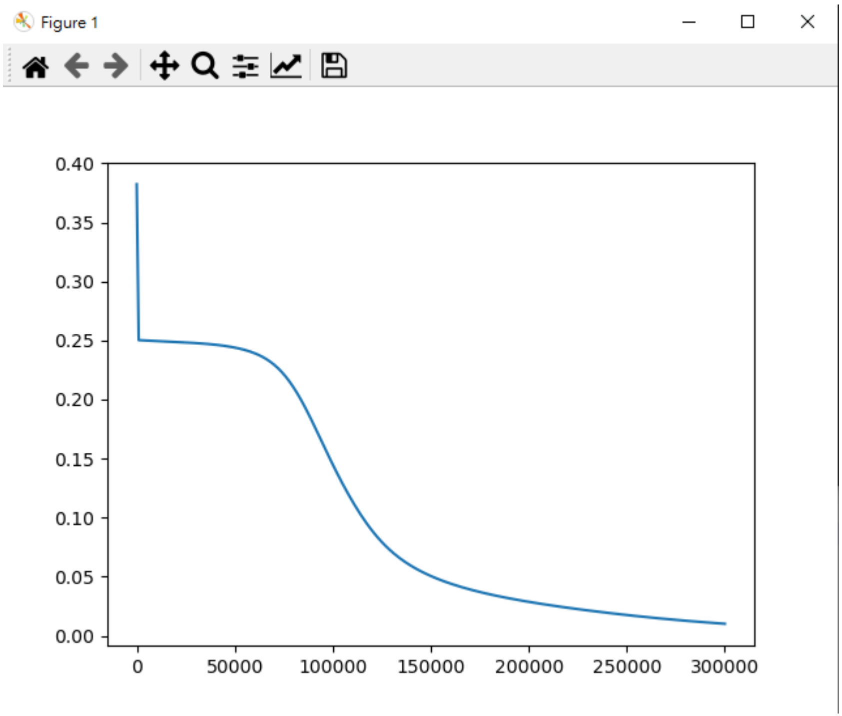
XOR資料 → accuracy = 71.4%, epoch = 1000000(max_epoch)
loss-epoch 曲線圖



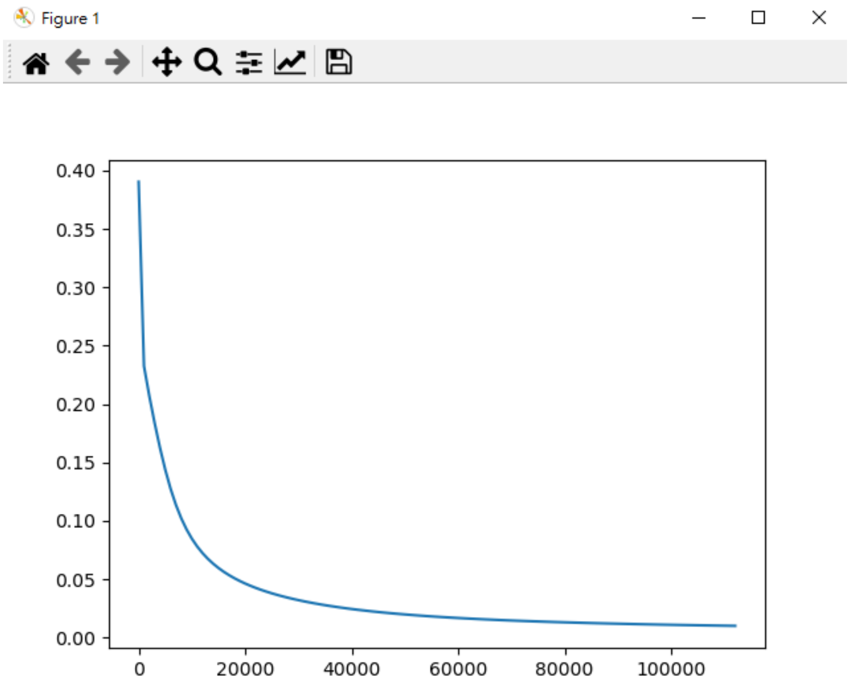
(b) layers size = [2,8,8,1], learning rate = 0.01, loss_threshold = 0.01
線性分布資料 → accuracy: 100%, epoch → 119000
epoch-loss 曲線圖



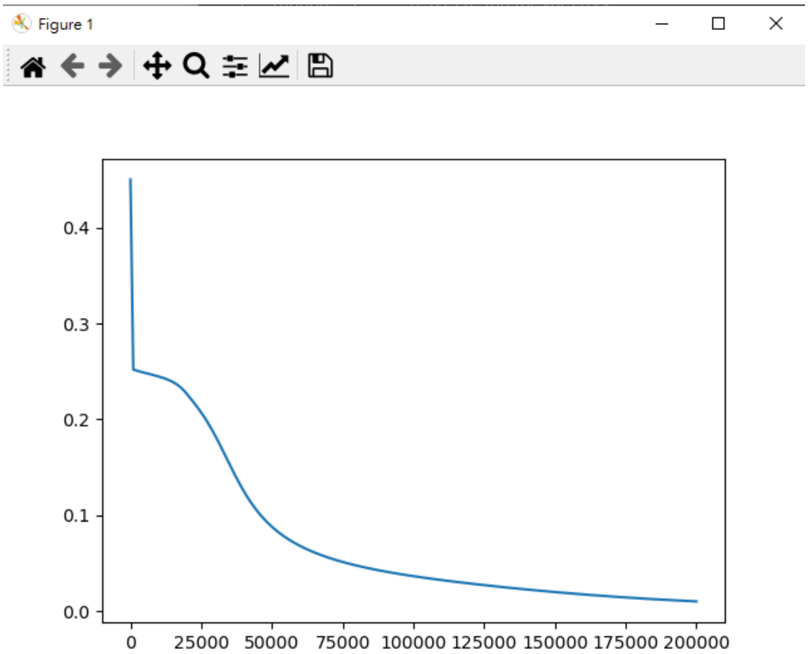
XOR資料 → accuracy: 100%, epoch → 300000
epoch-loss 曲線圖



(c) layers size = [2,8,8,1], learning rate = 0.01, loss_threshold = 0.01
線性資料分布 → accuracy: 99%, epoch: 112000
loss-epoch曲線圖



XOR資料分布 → accuracy: 100%, epoch: 200000
loss-epoch曲線圖



在這項實驗中我嘗試了三個不同的hidden layer的size，分別是2x2、8x8、16x16，而從XOR的實驗結果可以發現被hidden layer size影響最大的是開始收斂所需要的epoch數。size越大，開始收斂所需要的epoch數量越少。但相對的，由於網路架構變大，因此每一次epoch所需要花費的時間也較多。

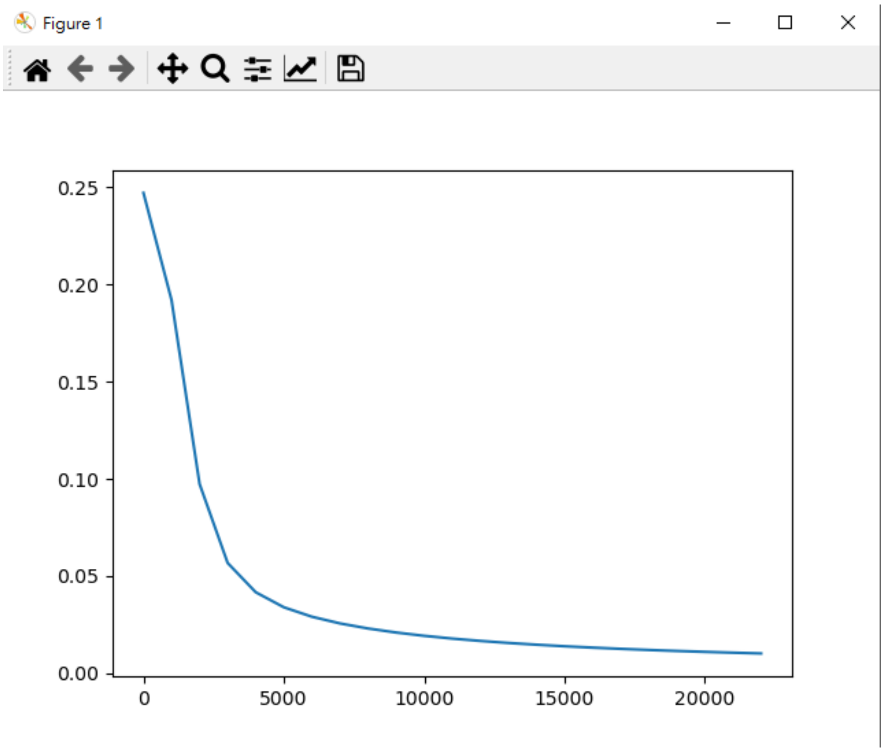
3. Without activation function

將forward的sigmoid部分移除，並將新的activation function視為y = x，故在backward部分derivative activation function = 1。

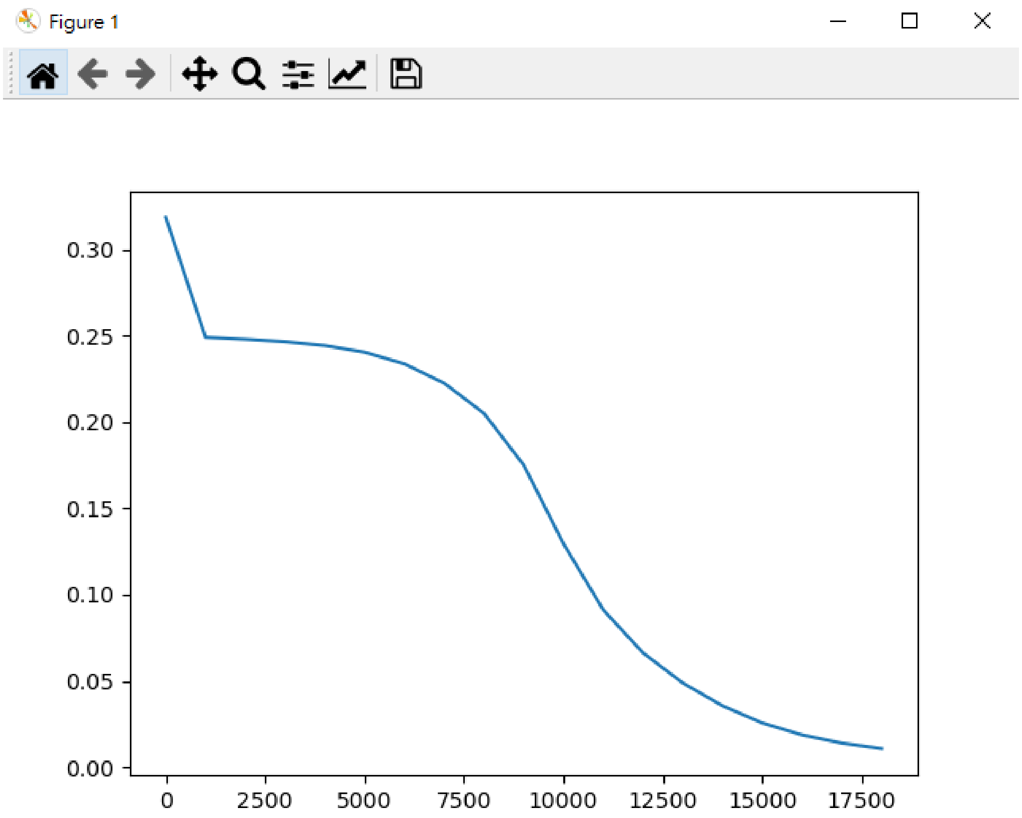
若將output layer的activation也移除會導致loss value為nan，故保留為sigmoid

線性資料分布 → accuracy: 100%, epoch: 22000

epoch-loss曲線圖



XOR資料 → accuracy: 100%, epoch: 18000
epoch-loss曲線圖



在不使用activation的情況下，於hidden layer中的資料會更為分散，導致權重在收斂上的難度也會提高。

V. Extra

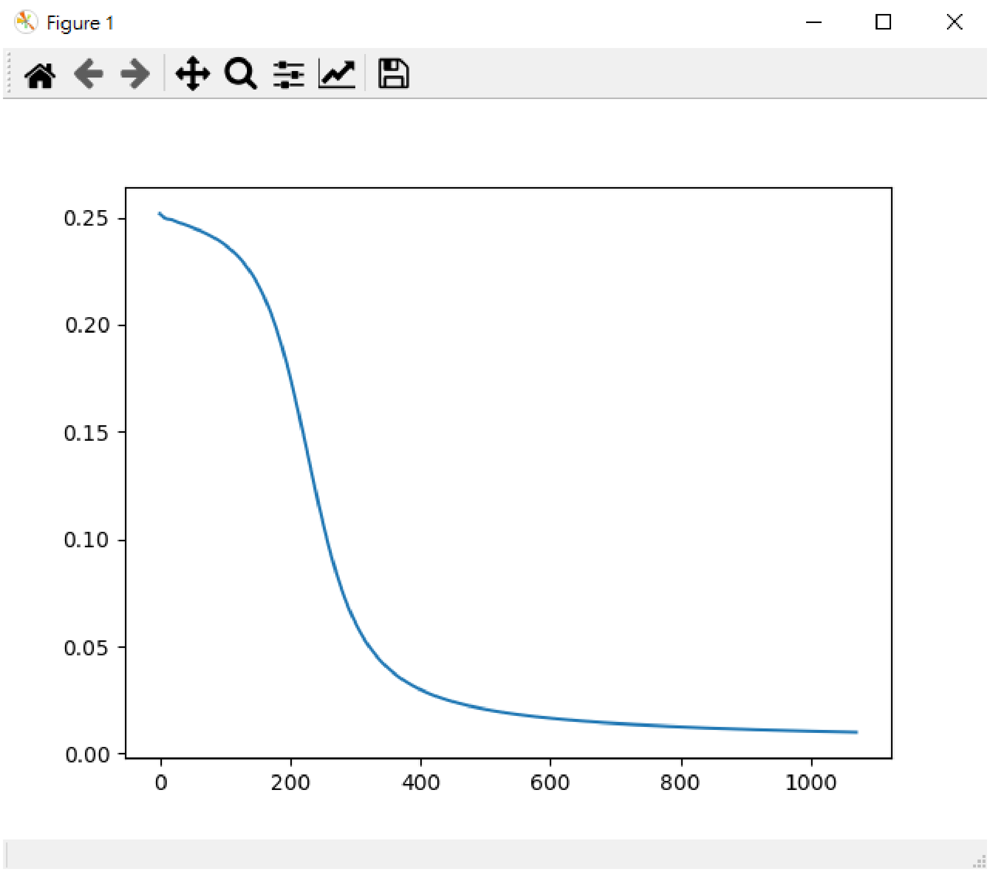
1. Implement different optimizer:

activation function: sigmoid, learning rate: 0.01

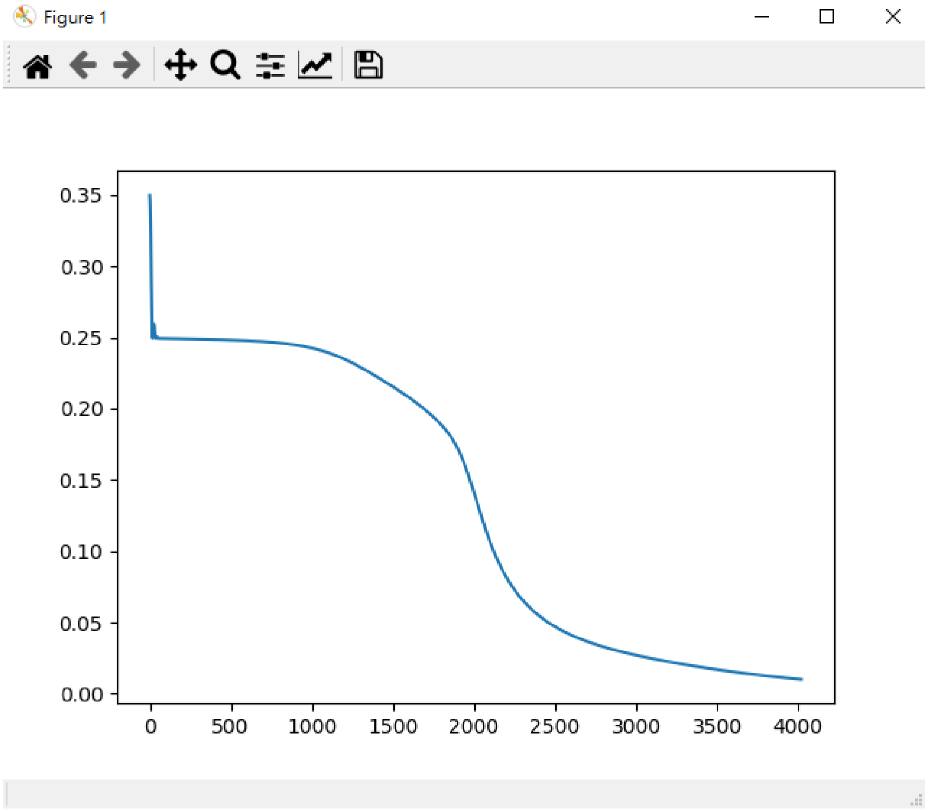
- momemtum

```
#momentum
self.momentum = 0.9 * self.momentum - lr * self.gradient
self.weight = self.weight + self.momentum
```

線性資料分布 →
accuracy:
accuracy: 100%, epoch: 1070



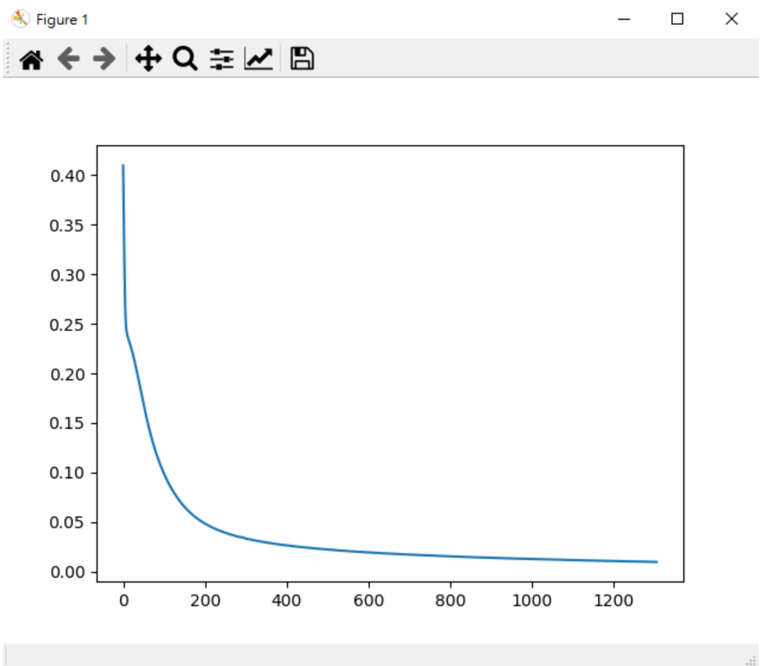
XOR資料 →
accuracy: 100%, epoch:4021



- Adagrad

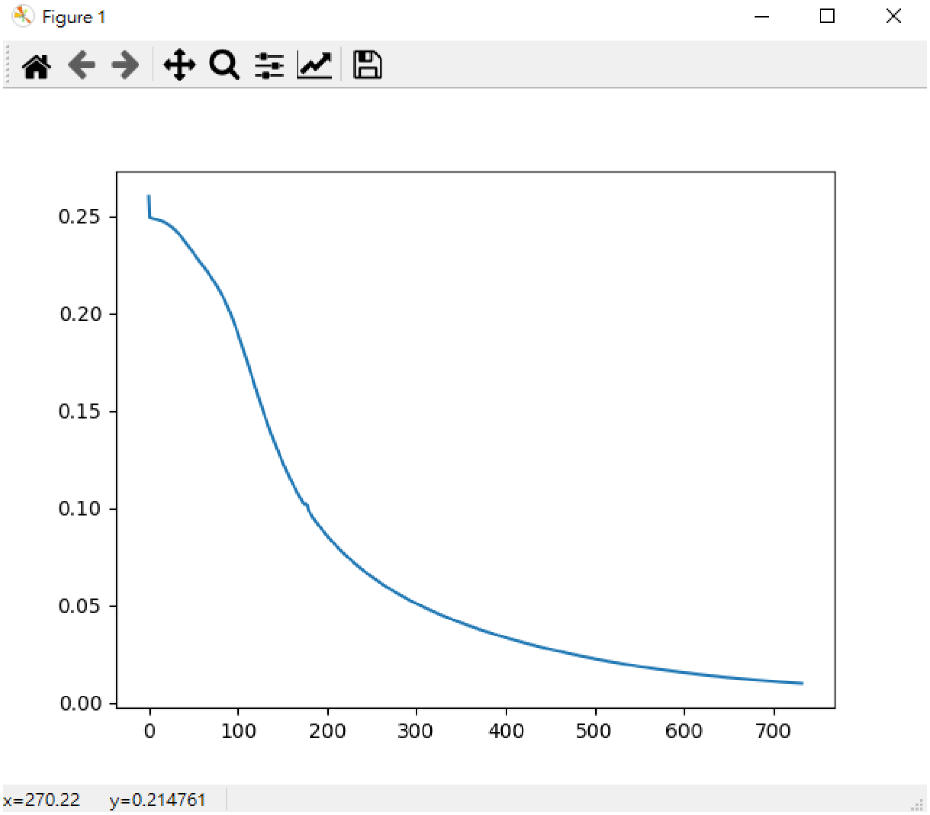
```
#adagrad
self.n += np.square(self.gradient)
n_lr = np.divide(lr, np.sqrt(self.n + 1e-8))
self.weight = self.weight - n_lr * self.gradient
```

線性資料分布 →
accuracy: 100%, epoch: 1304



XOR資料 →

accuracy: 100%, epoch: 732

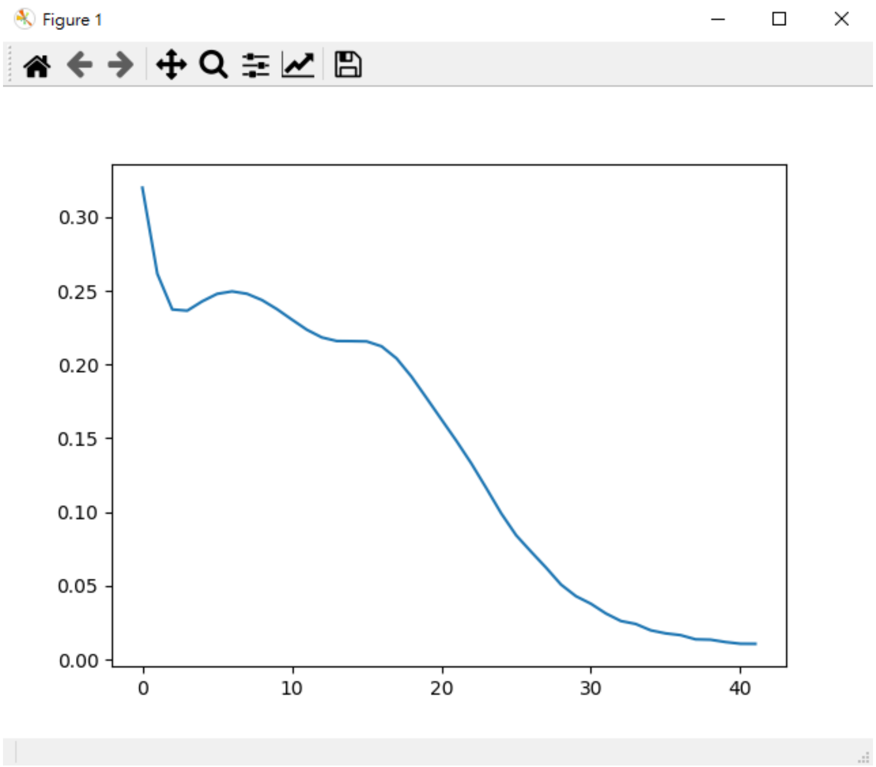


- Adam

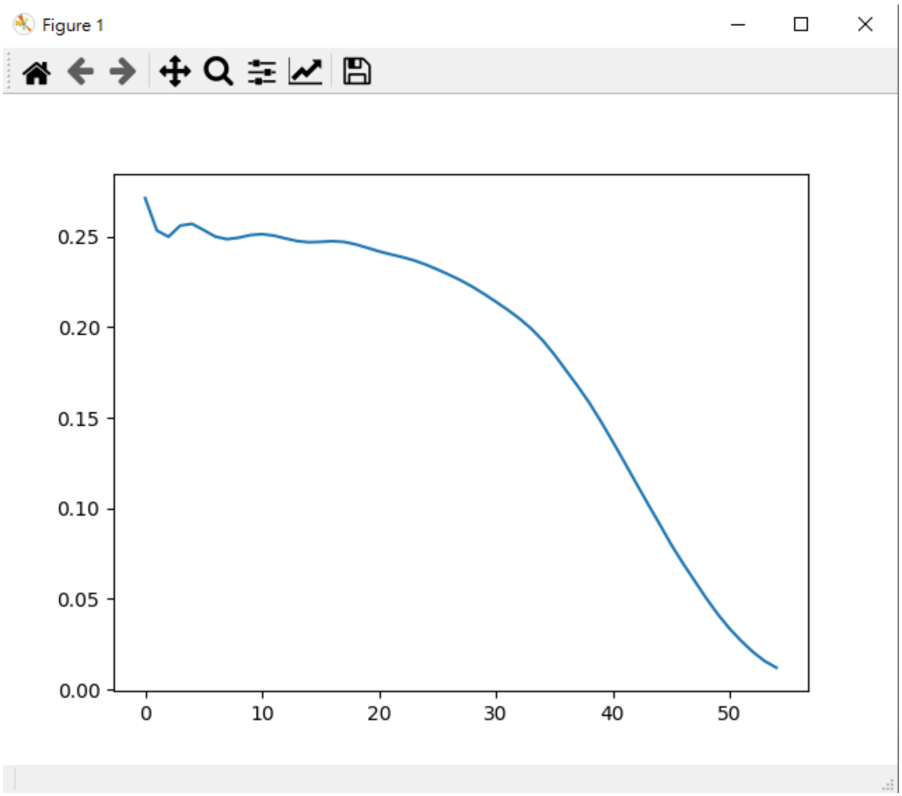
```
#adam
self.mt = self.b1 * self.mt + (1 - self.b1) * self.gradient
self.vt = self.b1 * self.vt + (1 - self.b2) * np.square(self.gradient)
mt_hat = np.divide(self.mt, 1 - self.b1 ** self.t)
vt_hat = np.divide(self.vt, 1 - self.b2 ** self.t)
self.t += 1
self.weight = self.weight - lr * np.divide(mt_hat, np.sqrt(vt_hat) + 1e-8)
```

線性資料分布 →

accuracy: 100%, epoch: 41



XOR資料:
accuracy: 100%, epoch: 54



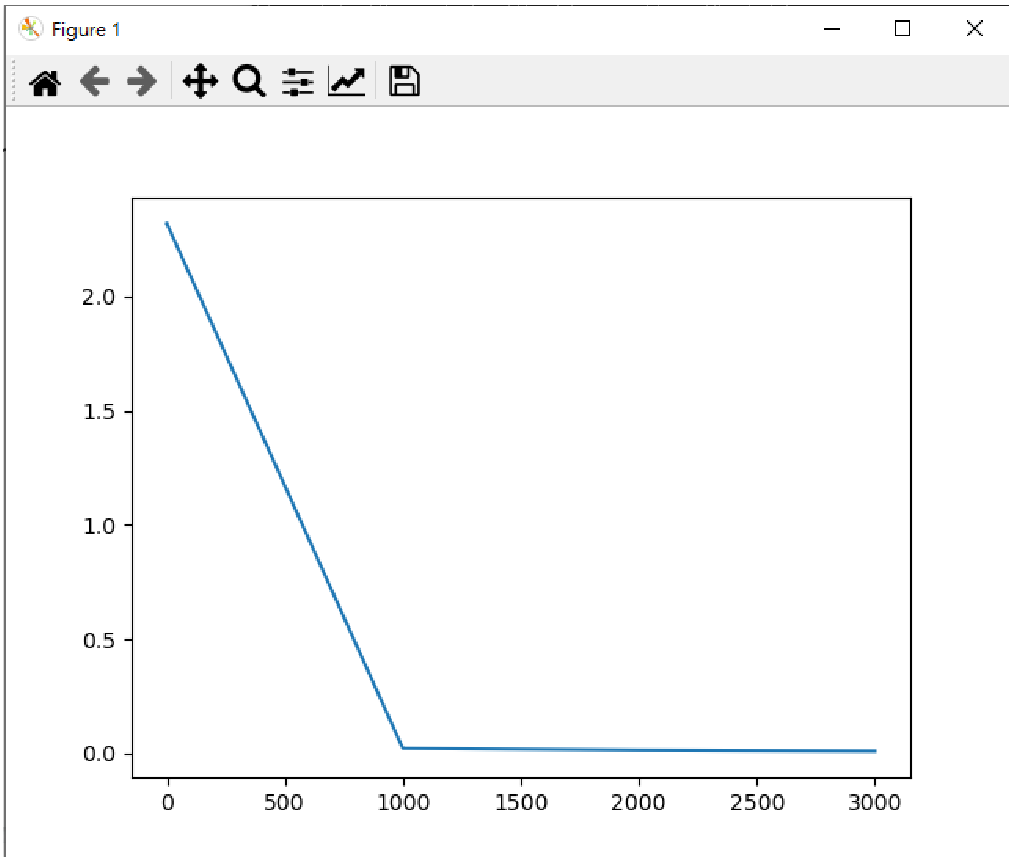
2. Implement different activation functions:

- Tanh

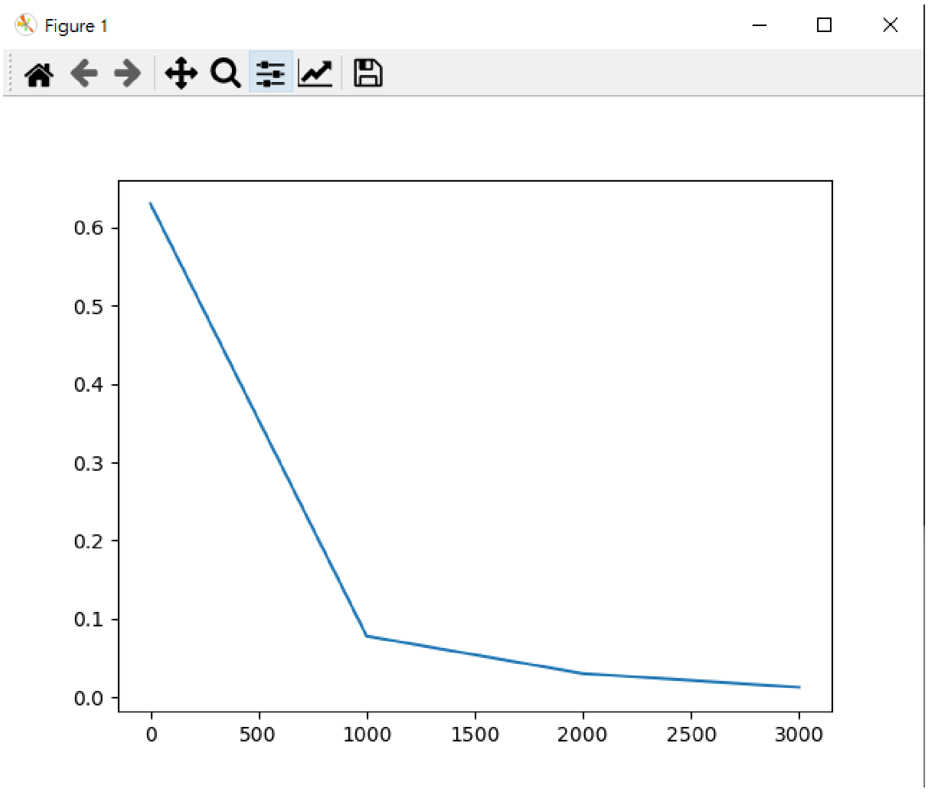
```
def tanh(x):
    return (np.exp(x) - np.exp(-x)) / (np.exp(x) + np.exp(-x))

def derivative_tanh(y):
    return 1 - np.multiply(y,y)
```

線性資料分布 →
accuracy: 100%, epoch: 3000



XOR分布 →
accuracy: 100%, epoch: 3000

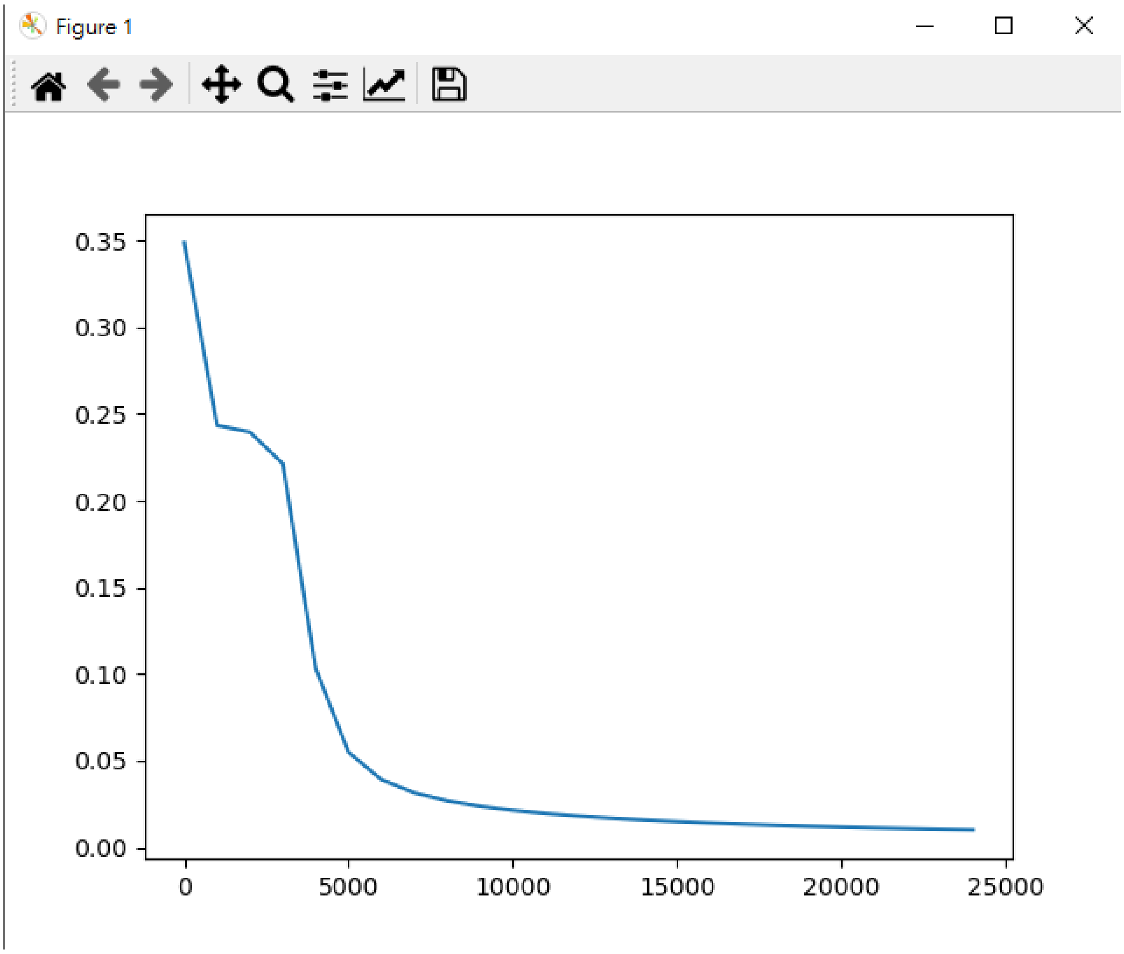


- Relu(output layer使用sigmoid)

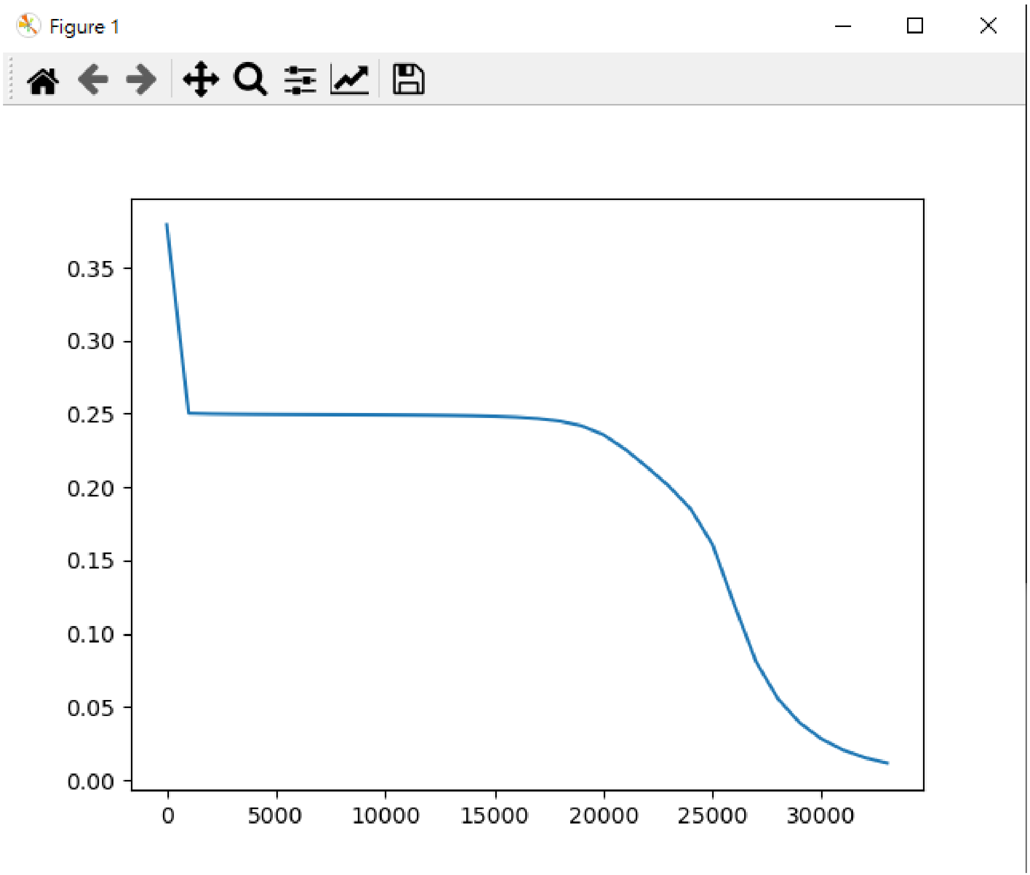
```
def relu(x):
    return np.maximum(0,x)

def derivative_relu(x):
    x[x <= 0] = 0
    x[x >  0] = 1
    return x
```

線性分布資料 →
accuracy: 100%, epoch: 24000



XOR資料 →
accuracy: 100%, epoch → 33000

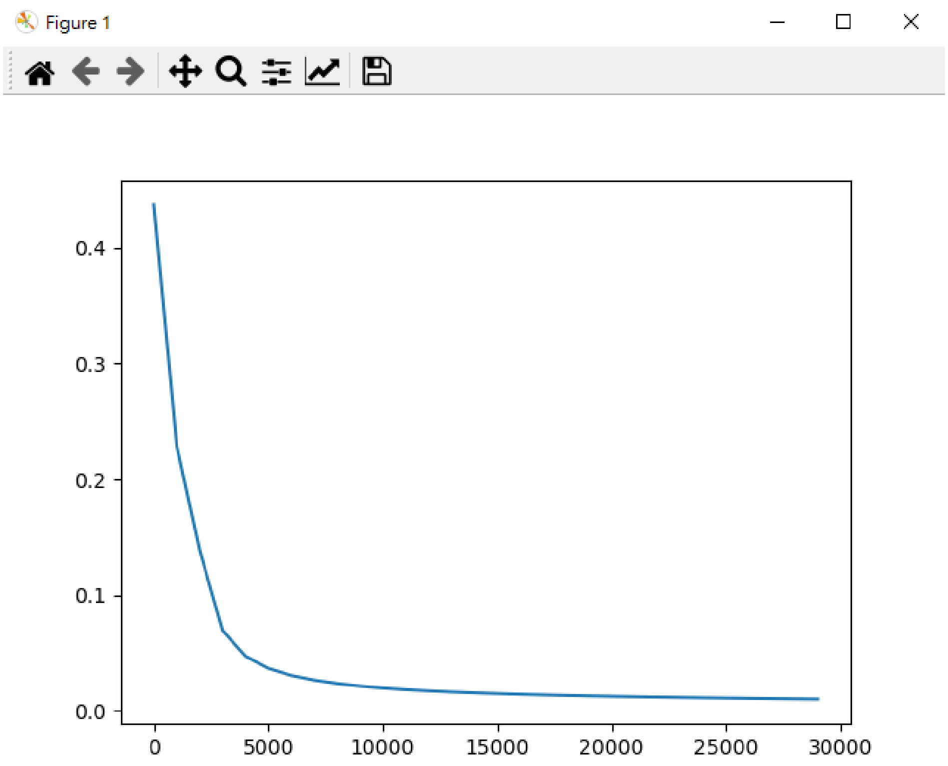


- Leaky-Relu(output layer使用sigmoid)

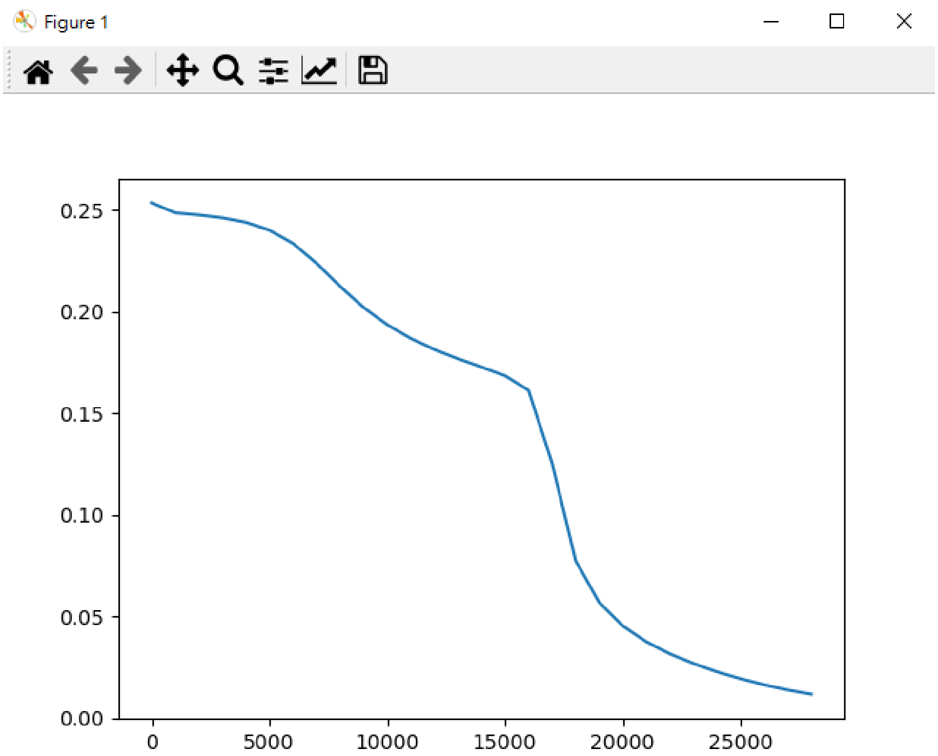
```
def leaky_relu(x):
    return np.maximum(0,x) + np.minimum(0, 0.01 * x)

def derivative_leaky_relu(x):
    x[x <= 0] = 0.01
    x[x >  0] = 1
    return x
```

線性分布資料 →
accuracy: 100%, epoch: 29000



XOR資料 →
accuracy: 100%, epoch: 28000



結論而言，在實驗中經過多次嘗試後，我在這次作業中嘗試出最有效果的activation function是tanh，收斂速度最快並且所需要的epoch數量最少。

Relu和Leaky-Relu的話在這次實驗中相對表現比較不穩定，若不將output layer維持sigmoid或是tanh的話常常會發生dead relu problem，或甚至是根本無法產生loss值。