深度學習

Lab1 : back-propagation

312605003

王語

2023/7/29

---

1. Introduction

本次實驗在僅使用標準函式庫以及 Numpy 的前提下實作神經網路，此神經網路可以使用不同的激勵函數(Active Function)、不同的優化器(Optimizer)以及兩個可調整神經元數目的隱藏層，以上參數可以透過 Command Line Arguments 來設定，具體操作方式詳見 README.md。

實驗中使用正向傳播(Forward propagation)計算預測值，透過 MSE 計算 Loss，使用反向傳播(Back-propagation)計算梯度，透過優化器更新參數，並將結果輸出於 output.txt。

2. Experiment setups

A. Sigmoid functions

用於正向傳播中的 sigmoid function:

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

用於反向傳播中的 derivative sigmoid function:

```python
def derivative_sigmoid(x):
    return np.multiply(x , 1.0-x )
```

B. Neural network

通過 two_layer_network 這個 class 來建立實驗需要的 Neural network，在__init__中會根據訓練參數的設定以及神經網路參數的設定，建構神經網路並且生成初始參數。

```python
class two_layer_network ():
    def __init__(self,neurals_input,neurals_hl_1,neurals_hl_2,neurals_output,learning_rate,optimize_method,weight_initialization) -> None:
        Din_w1 = neurals_input
        Dout_w1 = neurals_hl_1
        Din_w2 = Dout_w1
        Dout_w2 = neurals_hl_2
        Din_w3 = Dout_w2
        Dout_w3 = neurals_output
        np.random.seed(2) # easier to debug
        if (weight_initialization=='normal'):
            self.w1 = np.random.normal(0 , 1 ,(Din_w1,Dout_w1))
            self.w2 = np.random.normal(0 , 1 ,(Din_w2,Dout_w2))
            self.w3 = np.random.normal(0 , 1 ,(Din_w3,Dout_w3))

        elif (weight_initialization=='randn'):
            self.w1 = np.random.randn(Din_w1,Dout_w1) *0.01 # avg = 0 std_deviation = 0.01
            self.w2 = np.random.randn(Din_w2,Dout_w2) *0.01 # avg = 0 std_deviation = 0.01
            self.w3 = np.random.randn(Din_w3,Dout_w3) *0.01 # avg = 0 std_deviation = 0.01
        else :
            raise ArgumentTypeError('weight_initialization : normal or randn')

        self.learning_rate = learning_rate
        if (optimize_method == 'gdm'):
            self.w1_m = np.zeros((Din_w1,Dout_w1))
            self.w2_m = np.zeros((Din_w2,Dout_w2))
            self.w3_m = np.zeros((Din_w3,Dout_w3))
```

此外是 two_layer_network 的方法說明，這個 class 中包含以下五個功能，正向傳播、Loss 計算、反向傳播、梯度計算、參數更新。

```python
def forwardpass(self,input_data,active_funtion):...

def MSE_loss(self,ground_truth):...

def backward_pass(self,ground_truth,active_funtion):...

def gradient(self,input_data):...

def update(self,optimize_method):...
```

C. Backpropagation

即根據激勵函數的設定，將 loss 由 output 向 input 傳遞，計算出 Backward pass。

```python
def backward_pass(self,ground_truth,active_funtion):
    if (active_funtion == 'sigmoid'):
        self.c_y = 2*(self.pred_y - ground_truth) / (self.pred_y.shape[0])
        self.c_z3 = derivative_sigmoid(self.pred_y)*self.c_y
        self.c_z2 = derivative_sigmoid(self.a2)*np.dot(self.c_z3,self.w3.T)
        self.c_z1 = derivative_sigmoid(self.a1)*np.dot(self.c_z2,self.w2.T)

    elif (active_funtion == 'relu'):
        self.c_y = 2*(self.pred_y - ground_truth) / (self.pred_y.shape[0])
        self.c_z3 = derivative_relu(self.pred_y)*self.c_y
        self.c_z2 = derivative_relu(self.a2)*np.dot(self.c_z3,self.w3.T)
        self.c_z1 = derivative_relu(self.a1)*np.dot(self.c_z2,self.w2.T)

    elif (active_funtion == 'tanh'):
        self.c_y = 2*(self.pred_y - ground_truth) / (self.pred_y.shape[0])
        self.c_z3 = derivative_tanh(self.pred_y)*self.c_y
        self.c_z2 = derivative_tanh(self.a2)*np.dot(self.c_z3,self.w3.T)
        self.c_z1 = derivative_tanh(self.a1)*np.dot(self.c_z2,self.w2.T)

    else :
        raise ArgumentTypeError('active function : sigmoid , relu or tanh')
    # ------------shape------------SHOW UP : command+K then commend+U
    # print('c_z1',self.c_z1.shape)               # c_z1 (100, 4)
    # print('c_z2',self.c_z2.shape)               # c_z2 (100, 4)
    # print('c_z3',self.c_z3.shape)               # c_z3 (100, 1)
    # ------------shape------------Annotations : command+K then commend+C
    return None
```

再將 Forward Pass 與 Backward Pass 相乘即可得到梯度。

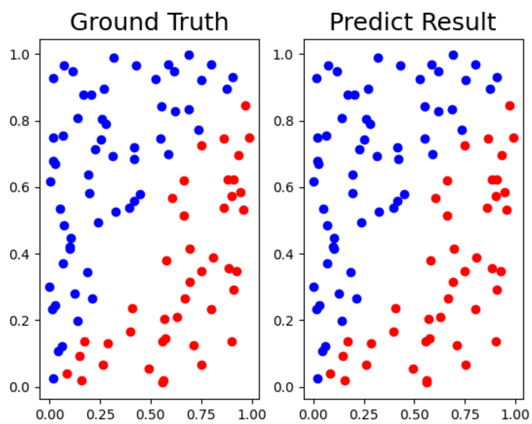
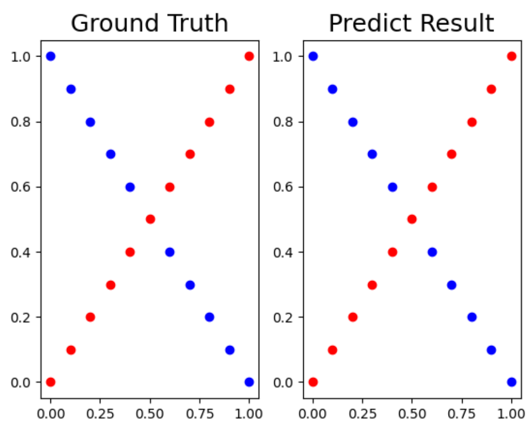
```python
    def gradient(self,input_data):
        # self.c_w1 = np.dot(input_data.T,self.c_z1)
        self.c_w1 = gradient_cal(input_data , self.c_z1 , self.w1.shape)
        # self.c_w2 = np.dot(self.a1.T , self.c_z2 )
        self.c_w2 = gradient_cal(self.a1 , self.c_z2 ,self.w2.shape)
        # self.c_w3 = np.dot(self.a2.T , self.c_z3 )
        self.c_w3 = gradient_cal(self.a2 , self.c_z3 ,self.w3.shape)
        # ------------shape------------SHOW UP : command+K then commend+U
        # print('c_w1',self.c_w1.shape)            # c_w1 (2, 4)
        # print('c_w2',self.c_w2.shape)            # c_w2 (4, 4)
        # print('c_w3',self.c_w3.shape)            # c_w3 (4, 1)
        # ------------shape------------Annotations : command+K then commend+C
        return None

def gradient_cal(forward_gradient , backward_gradient ,refshape):

    if (len(forward_gradient)==len(backward_gradient)):
        result = np.zeros(refshape)
        for i in range (len(forward_gradient)):
            B_array = np.array([backward_gradient[i]])
            F_array = np.array([forward_gradient[i]])
            result = result + np.dot(F_array.T , B_array)
    else :
        raise IndexError('len(forward_gradient)!=len(backward_gradient)')
    result = np.clip(result,-1024. , 1023.)
    return result
```

3. Results of your testing

   A. Screenshot and comparison figure

Linear data	XOR data
	

結果完全吻合，準確率 100%

B. Show the accuracy of your prediction

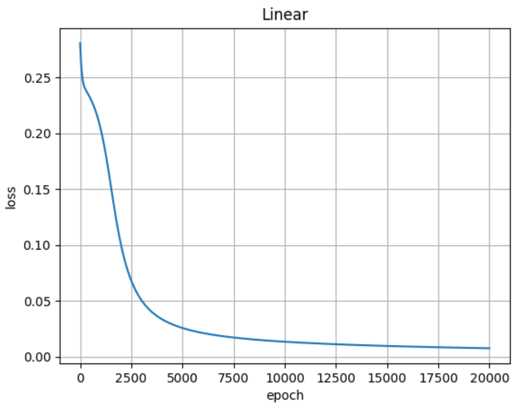
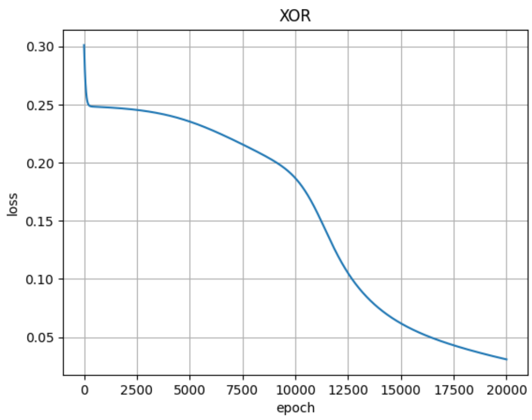
Linear data	Acc = 1.0

```
Iter  80        ground truth = 1        prediction = 0.9992845109069186
Iter  81        ground truth = 1        prediction = 0.9979873702203292
Iter  82        ground truth = 0        prediction = 0.022427403489990092
Iter  83        ground truth = 1        prediction = 0.999811850675937
Iter  84        ground truth = 0        prediction = 0.18899821175535497
Iter  85        ground truth = 1        prediction = 0.9998945135113404
Iter  86        ground truth = 0        prediction = 0.004247232169143875
Iter  87        ground truth = 1        prediction = 0.9975269127497143
Iter  88        ground truth = 0        prediction = 0.0007452805121604041
Iter  89        ground truth = 1        prediction = 0.9996198905472635
Iter  90        ground truth = 0        prediction = 0.000709762424306782
Iter  91        ground truth = 0        prediction = 0.0004212788163292185
Iter  92        ground truth = 1        prediction = 0.9998698841184118
Iter  93        ground truth = 1        prediction = 0.9998276857177403
Iter  94        ground truth = 1        prediction = 0.9997631751001913
Iter  95        ground truth = 0        prediction = 0.005832098138625528
Iter  96        ground truth = 1        prediction = 0.7356151094037776
Iter  97        ground truth = 1        prediction = 0.5497478916590133
Iter  98        ground truth = 1        prediction = 0.9990596866289785
Iter  99        ground truth = 1        prediction = 0.9996012608418873
acc = 1.0
```

XOR data	Acc = 1.0

```
Iter  0         ground truth = 0        prediction = 0.015098164021464201
Iter  1         ground truth = 1        prediction = 0.9872251520167424
Iter  2         ground truth = 0        prediction = 0.047022699156160114
Iter  3         ground truth = 1        prediction = 0.9870965251578503
Iter  4         ground truth = 0        prediction = 0.12461519831586214
Iter  5         ground truth = 1        prediction = 0.9822752416180895
Iter  6         ground truth = 0        prediction = 0.2203891660553177
Iter  7         ground truth = 1        prediction = 0.9434413486514303
Iter  8         ground truth = 0        prediction = 0.26751487498728604
Iter  9         ground truth = 1        prediction = 0.5900754645882952
Iter  10        ground truth = 0        prediction = 0.25129906498577637
Iter  11        ground truth = 0        prediction = 0.19952408610332104
Iter  12        ground truth = 1        prediction = 0.5618544611256031
Iter  13        ground truth = 0        prediction = 0.14319516129586132
Iter  14        ground truth = 1        prediction = 0.9277794568354552
Iter  15        ground truth = 0        prediction = 0.09866259529839719
Iter  16        ground truth = 1        prediction = 0.9808921925855814
Iter  17        ground truth = 0        prediction = 0.06840434717136686
Iter  18        ground truth = 1        prediction = 0.9894667348747002
Iter  19        ground truth = 0        prediction = 0.04909464281597779
Iter  20        ground truth = 1        prediction = 0.9919123077340395
acc = 1.0
```

C. Learning curve (loss, epoch curve)

<table>
<tr><th colspan="2">Learning rate = 0.1</th></tr>
<tr><td>Linear data</td><td>XOR data</td></tr>
<tr><td>
</td><td>
</td></tr>
<tr><td>20000 epoch : loss = 0.0077329034746593615</td><td>20000 epoch : loss = 0.030861509686194014</td></tr>
</table>

D. Anything you want to present

<table>
<tr><th colspan="2">output</th></tr>
<tr><td>Linear data</td><td>XOR data</td></tr>
<tr><td><pre>
| training setting |
    data : Linear
    active finction : sigmoid
    optimize method : gd
    weight initialization : normal
    epoch : 20000

| network parameters |
    h1 units : 4
    h2 units : 4

| result |
    trainning loss = 0.0077329034746593615

    testing loss = 0.009623905122730262
    accuracy = 1.0
</pre></td><td><pre>
| training setting |
    data : XOR
    active finction : sigmoid
    optimize method : gd
    weight initialization : normal
    epoch : 20000

| network parameters |
    h1 units : 4
    h2 units : 4

| result |
    trainning loss = 0.030861509686194014

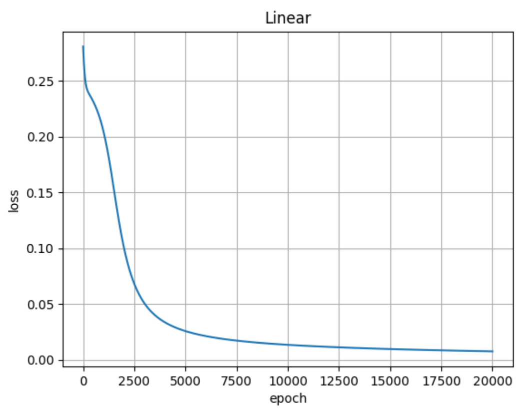
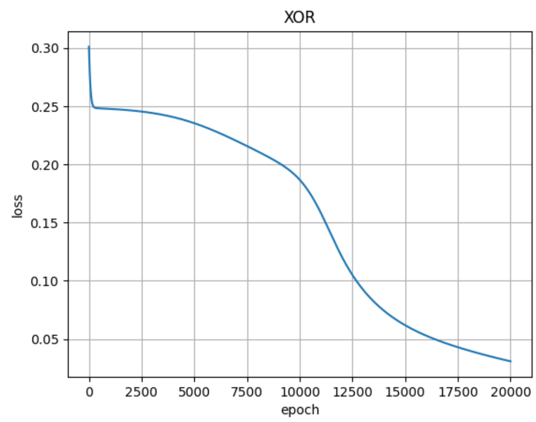
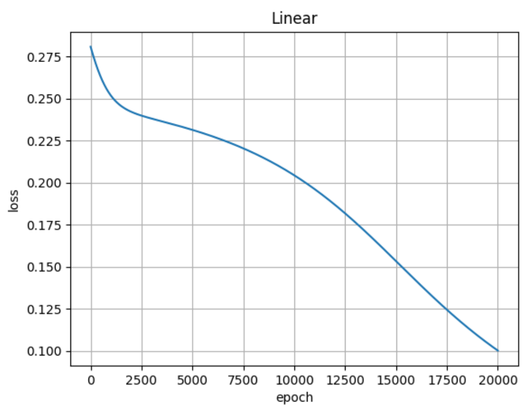
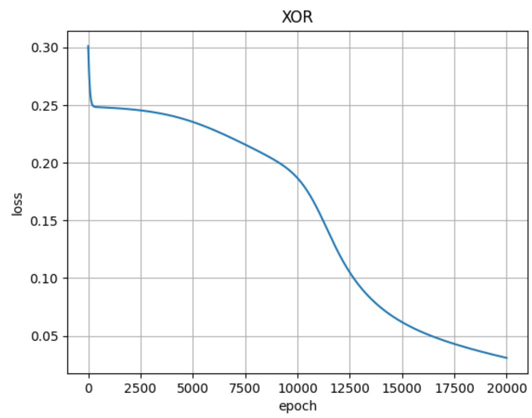
    testing loss = 0.03085728423640736
    accuracy = 1.0
</pre></td></tr>
</table>

記錄該次訓練設定。

此處的參數是程式預設數值，在 Discussion 章節中的對照組皆使用預設數值。

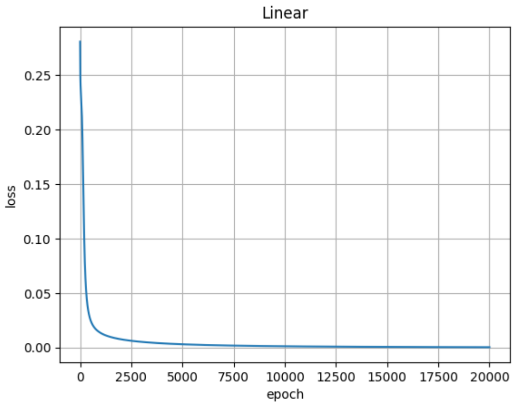
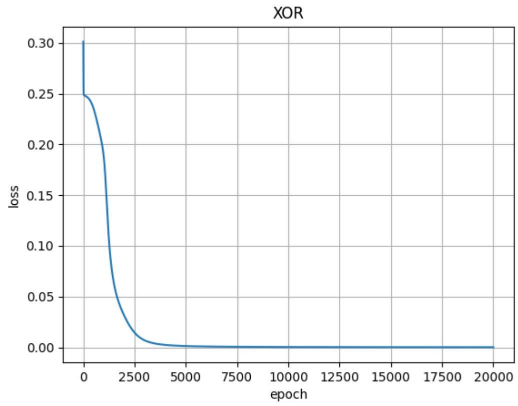
4. Discussion

A. Try different learning rates

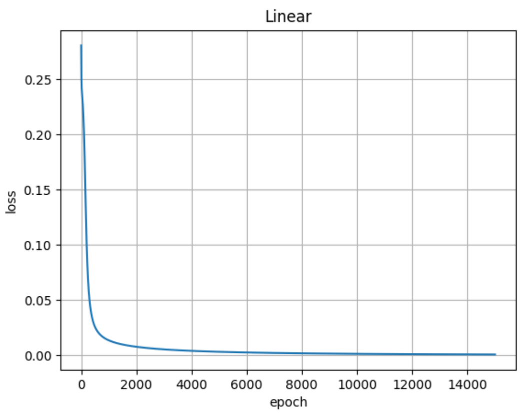
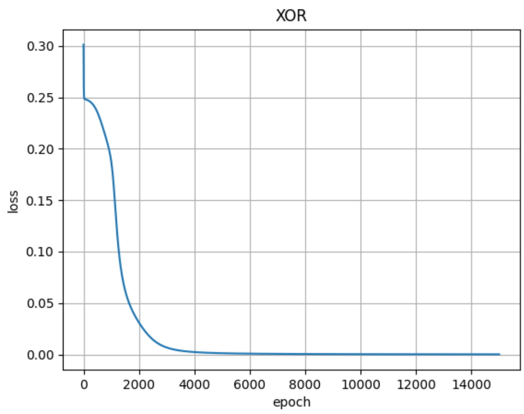
Learning rate = 0.1 (對照組)	
Linear data	XOR data
	
20000 epoch : loss = 0.007732903474659361	20000 epoch : loss = 0.030861509686194014
Acc = 1.0	Acc = 1.0
Learning rate = 0.01 （實驗組）	
Linear data	XOR data
	
20000 epoch : loss = 0.10014453046463924	20000 epoch : loss = 0.24647790519187202
Acc = 0.94	Acc = 0.52

對比 Learning rate = 0.1 與 Learning rate = 0.01 的結果可以發現 Learning rate = 0.01 太小了，導致 loss 還沒有收斂到最小值，要增加訓練次數才能得到更好的結果。其中 XOR data 的 loss 最終落在 0.246，若所有數據都由隨機的方式猜測 loss 將會貼近 0.25，說明了這個神經網路並沒有

從訓練資料中得到優化，在測試資料中的準確率也只有 0.52，代表訓練非常不良，應調整訓練參數。

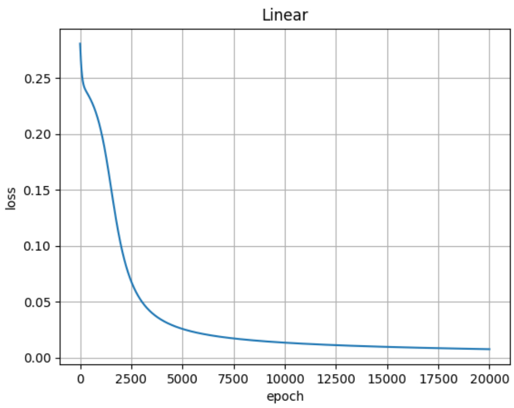
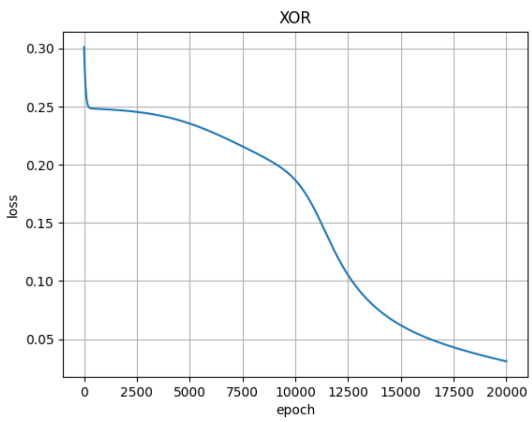
Learning rate = 1 （實驗組）	
Linear data	XOR data
	
20000 epoch : loss = 0.0004890141339678335	20000 epoch : loss = 9.617784260566798e-05
Acc = 0.99	Acc = 1.0

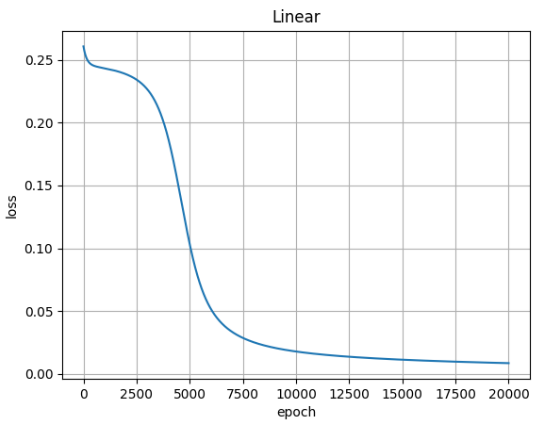
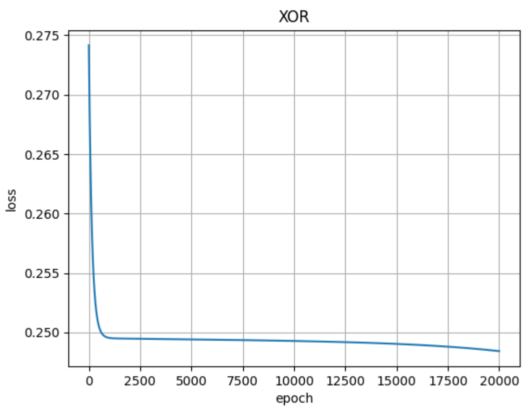
兩個資料源在 Learning rate = 1 都取得了不錯的表現，比起 Learning rate = 0.1，loss 也收斂到更小的數值，但是在 Linear data 中的準確率卻沒有比較好，推測是出現了 Over fitting 的現象，應該減少訓練次數。

Learning rate = 1 （驗證組）	
Linear data	XOR data
	
15000 epoch : loss = 0.0007520826722752352	15000 epoch : loss = 0.000147078690746444
Acc = 1.0	Acc = 1.0

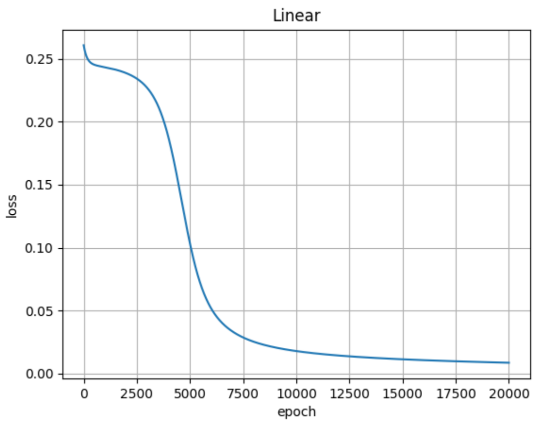
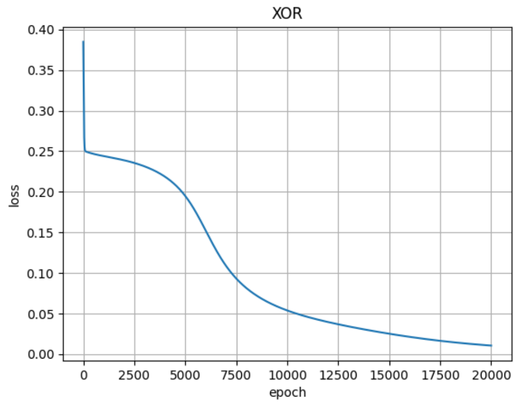
將訓練次數下降至 15000 次變得到了 1.0 的準確率，說明 Learning rate = 1 更加有效率。

B. Try different numbers of hidden units

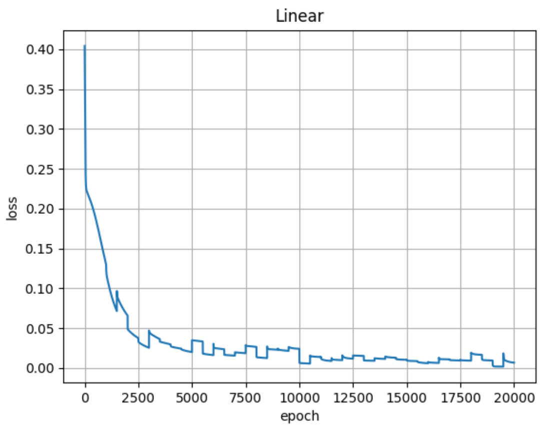
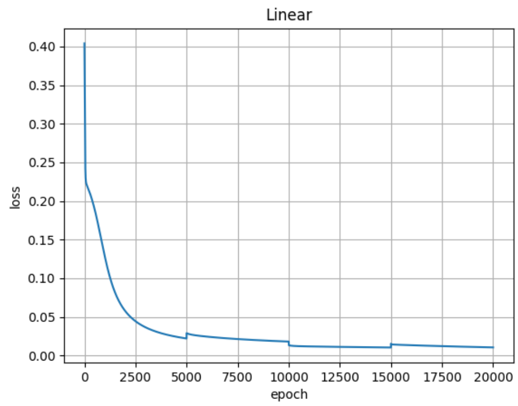
Units = 4 (對照組)	
Linear data	XOR data
	
20000 epoch : loss = 0.007732903474659361	20000 epoch : loss = 0.030861509686194014
Acc = 1.0	Acc = 1.0

Units = 2 (實驗組)	
Linear data	XOR data
	
20000 epoch : loss = 0.008657442206315567	20000 epoch : loss = 0.24843621581680947
Acc = 0.99	Acc = 0.52

下降成 2 units 在 XOR data 上會失去精準度，表示模型太過簡易，無法達成任務要求。

<table>
<tr><th colspan="2">Units = 8 (實驗組)</th></tr>
<tr><td>Linear data</td><td>XOR data</td></tr>
<tr><td>
</td><td>
</td></tr>
<tr><td>20000 epoch : loss = 0.007127570333514275</td><td>20000 epoch : loss = 0.01064488716425127</td></tr>
<tr><td>Acc = 0.98</td><td>Acc = 1.0</td></tr>
</table>

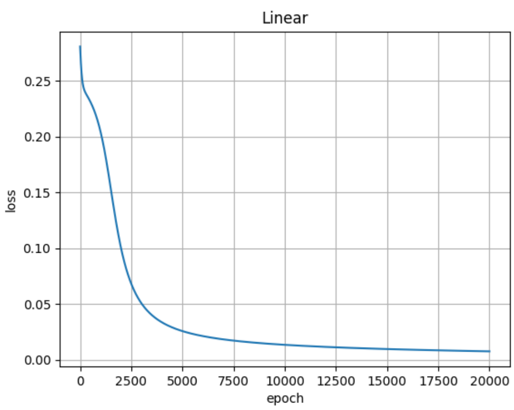
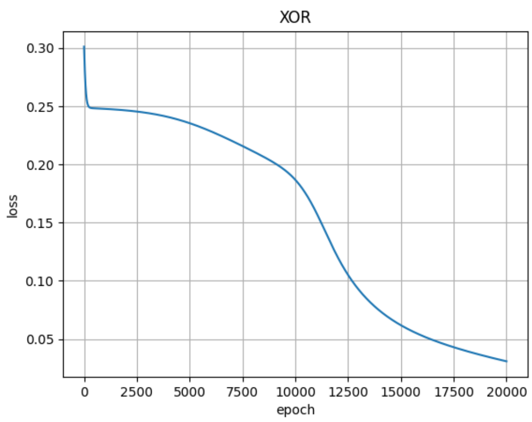
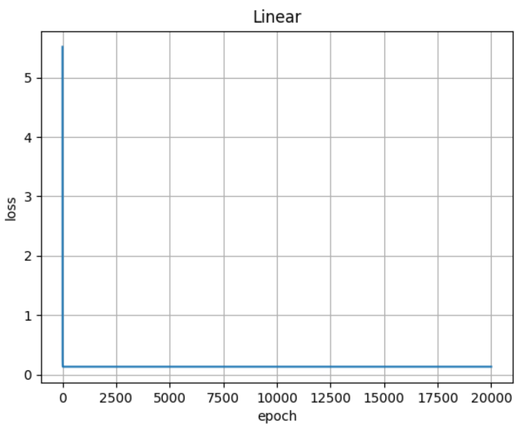
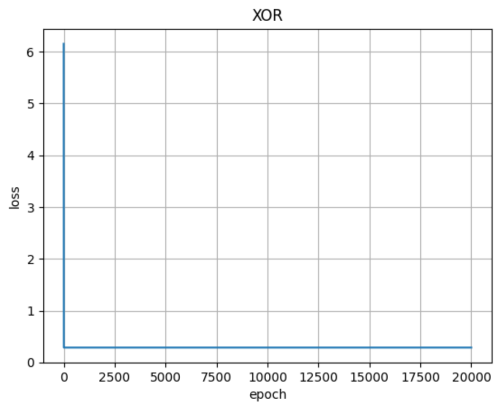
將 units 提升一倍後在 Linear data 上並沒有得到更好的表現，反而下降的 2 個百分比，推測是因為訓練資料沒有隨著模型擴增而增加。

<table>
<tr><th colspan="2">Units = 8 (驗證組)</th></tr>
<tr><td>Linear data</td><td>Linear data</td></tr>
<tr><td>
</td><td>
</td></tr>
<tr><td>20000 epoch : loss = 0.010447685355981123</td><td>20000 epoch : loss = 0.01057658215122545</td></tr>
<tr><td>每 500 epoch 生成新的訓練資料</td><td>每 5000 epoch 生成新的訓練資料</td></tr>
<tr><td>Acc = 0.99</td><td>Acc = 1.0</td></tr>
</table>

```python
if ((data_source=='Linear')&(data_reproduction)):
    input_data , label = generate_linear_train(seed=j+3)
```

通過在訓練過程中加入新的訓練資料，可以改善這個問題，雖然會讓學習曲線出現震盪，但是最終獲得較好的準確率。

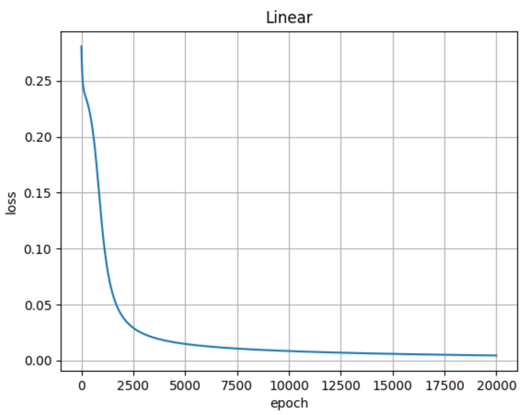
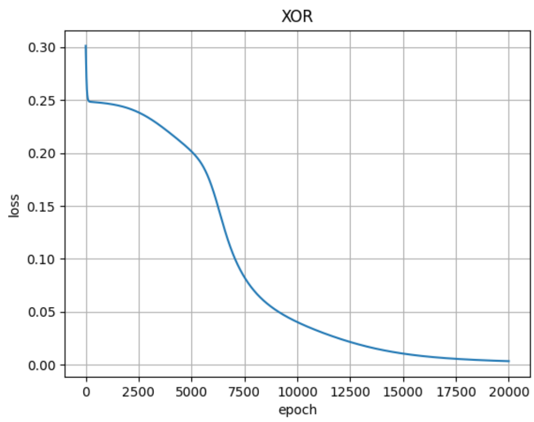
C. Try without activation functions

Sigmoid activation function (對照組)	
Linear data	XOR data
	
20000 epoch : loss = 0.007732903474659361	20000 epoch : loss = 0.030861509686194014
Acc = 1.0	Acc = 1.0
Without activation function (實驗組)	
Linear data	XOR data
	
20000 epoch : loss = 0.13689657367274508	200000 epoch : loss = 0.2887139107611548
Acc = 0.86	Acc = 0.333

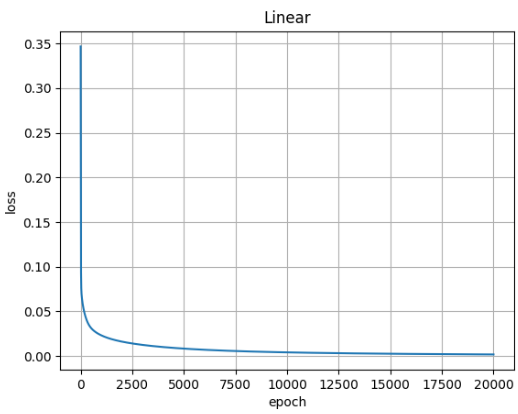
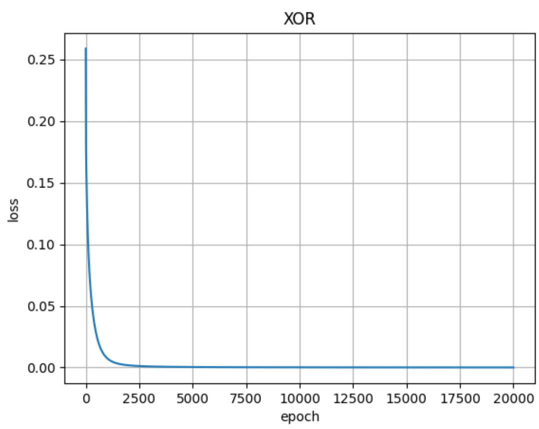
不使用激勵函數梯度絕對值會快速下降(下降幅度應該更大，但是我為了防止梯度過大因此加入了上下限，功能類似 torch.clamp( ))，導致無法更新參數。

5. Extra

A. Implement different optimizers.

gradient descent with momentum	
Linear data	XOR data
	
200000 epoch : loss = 0.004492213994639331	200000 epoch : loss = 0.003352094072948981
Acc = 1.0	Acc = 1.0

B. Implement different activation functions.

tanh activation function	
Linear data	XOR data
	
200000 epoch : loss = 0.0018610136203773735	200000 epoch : loss = 7.47451314505538e-05
Acc = 0.99	Acc = 1.0

C. Implement convolutional layers.

我有實現簡易的卷積神經網路以及相關計算，但是我認為卷積層並不適合用於眼前的任務，單一一筆 Input Data 是 shape 為 (1,2) 的陣列，在不使用 Padding 的前提下要對這樣的陣列進行卷積，卷積核的大小只能設定為 （1,2）或是 （1,1），但是這樣的運算並不能體現出「特徵提取」的精髓，因為在數學上，只是將原本的矩陣乘上一組係數而已，因此我另外定義了輸入資料。

卷積神經網路最長被使用在圖像辨識，因此我假設輸入資料是 10 張 28x28 的 RGB 圖像。

```
input_data = np.random.randn(10, 3, 28, 28)
```

此卷積神經網路共有兩層卷積層以及活化層

```
def forward(self, input_data):
    x = self.conv1.forward(input_data)
    x = self.relu1.forward(x)
    x = self.conv2.forward(x)
    x = self.relu2.forward(x)
    return x
```

通過卷積運算得到預測結果

```
# 卷積運算
for b in range(batch_size):
    for c_out in range(self.output_channels):
        for i in range(0, output_height, self.stride):
            for j in range(0, output_width, self.stride):
                input_slice = padded_input[b, :, i:i+self.kernel_size, j:j+self.kernel_size]
                output_data[b, c_out, i, j] = np.sum(input_slice * self.weights[c_out]) + self.bias[c_out]

return output_data
```