Deep Learning Lab1:Backpropagation

311512064 鄧書桓

1. Introduction:

本次的實驗要我們生成兩層 hidden layer 的神經網路,並建議使用 linear 與 sigmoid 來實現,其中並不能使用相關的 network 套件，如 pytorch 或是 tensor。在計算出 forward propagation 並預測出結果後，還要根據自己設計的 backpropagation 來更新權重，並需要疊代更新到準確率至少 90%才算成功, 其中需要對 learning rate 與其他超參數有一定的理解與調整才能實現。

2. Experiment setups:

A. Sigmoid functions

由 sigmoid 的公式$\sigma(x) = 1 / (1 + \exp(-x))$可知，其在 forward propagation 時有縮小數據的功能，能夠保證數據幅度不會差距過大，但其微分為$\sigma(x)(1-\sigma(x))$，會容易出現梯度消失與較長運算時間等問題。

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

def derivative_sigmoid(self, x):
    return np.multiply(x, 1-x)
```

B. Neural Network

這邊的設計流程為:

首先先初始化 network 中的所有權重、learning rate 與其他超參數。接著將資料從 input layer 經過 forward 值計算出該預測的 loss 後，從 output layer 進行 back propagation 來依序更新各個權重。最後重複執行上述動作直到到達最大的 epoch 或是準確率超過 90%。

```
class simple_CNN(object):
    def __init__(self, active_function = "sigmoid", learning_rate = 0.01, epoches = 500, optimization = "SGD"):

        # set the size of layer
        self.ip_size = 2
        self.h1_size = 50
        self.h2_size = 50
        self.op_size = 1

        #init the weight
        # self.w1 = np.random.randn(self.ip_size, self.h1_size)
        # self.w2 = np.random.randn(self.h1_size, self.h2_size)
        # self.w3 = np.random.randn(self.h2_size, self.op_size)
        self.w1 = np.random.normal(size = (self.ip_size, self.h1_size))
        self.w2 = np.random.normal(size = (self.h1_size, self.h2_size))
        self.w3 = np.random.normal(size = (self.h2_size, self.op_size))

        #other class variable
        self.Loss_list = []
        self.x_train = []
        self.y_train = []
        self.datatype = ""
        self.learning_rate = learning_rate
        self.epoches = epoches
        self.active_function = active_function
        self.optimization = optimization
```

上圖為一開始的參數初始化，這邊的 input 與 output layer 都是固定好的，而中間的隱藏層我都選擇 50。而 weight 初始化的部分，採取高斯分布來做初始化，能夠獲得平均值為 0 且標準差為

```
def forward(self):
    if self.active_function == "sigmoid" :
        self.x1 = np.dot(self.x_train, self.w1) + self.b1
        # self.x1 = np.dot(self.x_train, self.w1)
        self.a1 = self.sigmoid(self.x1)
        self.x2 = np.dot(self.a1, self.w2) + self.b2
        # self.x2 = np.dot(self.a1, self.w2)
        self.a2 = self.sigmoid(self.x2)
        self.x3 = np.dot(self.a2, self.w3) + self.b3
        # self.x3 = np.dot(self.a2, self.w3)
        self.y_pred = self.sigmoid(self.x3)
```

上圖為我所設計的 hidden layer，這邊使用簡易的 wx+b 來實現，並搭配 sigmoid 來做 active function。加上 bias 的原因為能夠使其之後在優化 weight 時對於誤差的收斂速度能夠大幅提升。

C. Backpropagation

概念來實現，其公式如下:

$$\frac{\partial J}{\partial x} = \frac{\partial J}{\partial h}\frac{\partial h}{\partial z}\frac{\partial z}{\partial x}$$
$$= c'(h)g'(z)f'(x)$$

其中我的 loss function 是選用 MSE，因此其偏微分為-2*(y_

propagation。值得注意的是，在實作時，各個 weight 的偏微分中的大小千奇百怪，若用老師上課教的公式直接帶進去的話，會導致大小不符而出現 error，因使還需要在適當的地方加上 transpose。

```python
def back_propogation(self):
    self.Loss_list.append(self.MSE_Loss(self.y_train, self.y_pred))

    if self.active_function == "sigmoid" :
        loss_grad = -2*(self.y_train - self.y_pred)
        w3_forward = loss_grad * self.derivative_sigmoid(self.y_pred)
        w3_grad = np.dot(self.a2.T, w3_forward)
        w2_forward = w3_forward.dot(self.w3.T)*self.derivative_sigmoid(self.a2)
        w2_grad =  np.dot(self.a1.T, w2_forward)
        w1_forward = np.dot(w2_forward,self.w2.T)*self.derivative_sigmoid(self.a1)
        w1_grad = np.dot(self.x_train.T,w1_forward)
```

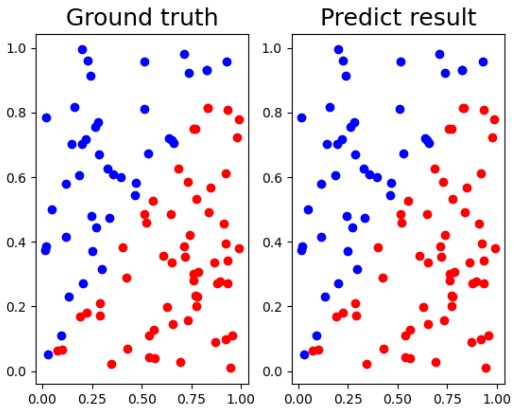
```python
elif(self.optimization == "SGD"):
    self.w3 -= self.learning_rate * w3_grad
    self.w2 -= self.learning_rate * w2_grad
    self.w1 -= self.learning_rate * w1_grad
    self.b3 -= self.learning_rate * w3_forward
    self.b2 -= self.learning_rate * w2_forward
    self.b1 -= self.learning_rate * w1_forward
```

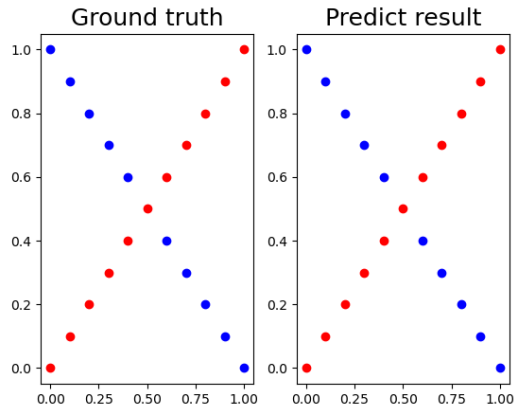
上圖為此部分的程式碼，由於還有使用 bias，因此有將 back propagation 拆成兩半，以便 bias 的更新。

3. Result of your testing

A. Screenshot and comparison figure

Linear VS XOR





在使用 sigmoid 當 active function 與 SGD 當 optimization 時，對於上面兩個非線性與線性的簡單輸入時，都能夠分類出來，並預測出與實際結果一模一樣的答案。

B. Show the accuraucy of my prediction

i. linear

```
[2.22551973e-03]
[9.99863992e-01]
[2.03601860e-03]
[2.90325625e-04]
[9.91093825e-01]
[9.98963400e-01]
[9.98992812e-01]
[9.99693319e-01]
[6.62490289e-03]
[1.25452688e-02]
[1.23270388e-02]
[1.76578324e-02]
[1.30748556e-02]
[9.92778670e-01]
[1.36619449e-02]
[9.99983620e-01]
[9.44374717e-03]
[9.82231082e-01]
[2.99948344e-03]]
Accurarcy is : 100.0%
```

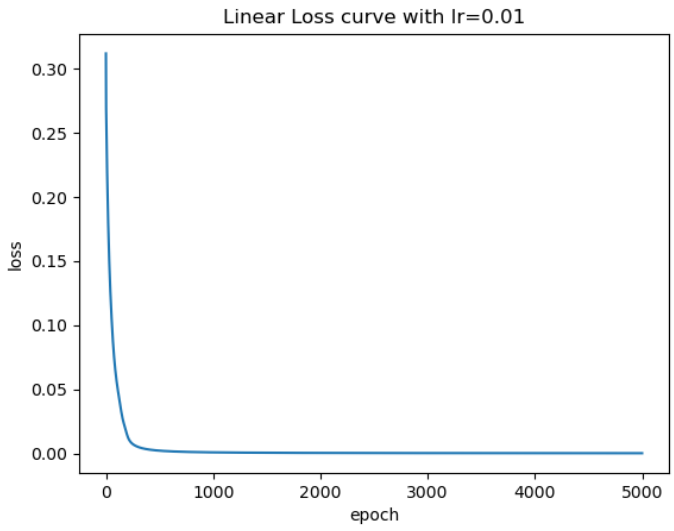
ii. XOR

```
[0.97582071]
[0.0202606 ]
[0.98703388]
[0.01737198]
[0.97634064]
[0.01981266]
[0.01056841]
[0.97578432]
[0.01827603]
[0.9956468 ]
[0.02604302]
[0.96928789]
[0.0207206 ]
[0.98066887]
[0.02176337]
[0.97978341]]
Accurarcy is : 100.0%
```

由上面 i 與 ii 可知，在使用 sigmoid 當 active function 與 SGD 當 optimization 時，準確率都可以達到 100%。

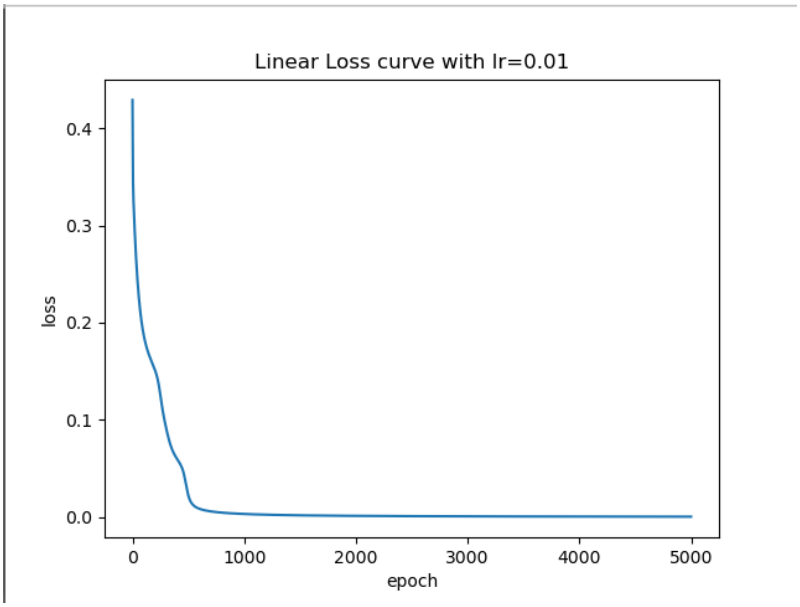
C. Learning curve(loss, epoch curve)

i. Linear



```
epoch 500 linear loss : 0.0022661772861661826
epoch 1000 linear loss : 0.0007390672095713333
epoch 1500 linear loss : 0.0004200021055682247
epoch 2000 linear loss : 0.00028704800864669705
epoch 2500 linear loss : 0.00021544492794013035
epoch 3000 linear loss : 0.00017115156408429498
epoch 3500 linear loss : 0.00014124740371539328
epoch 4000 linear loss : 0.00011980072202141738
epoch 4500 linear loss : 0.00010372317398414177
epoch 5000 linear loss : 9.125537316955902e-05
```

ii. XOR



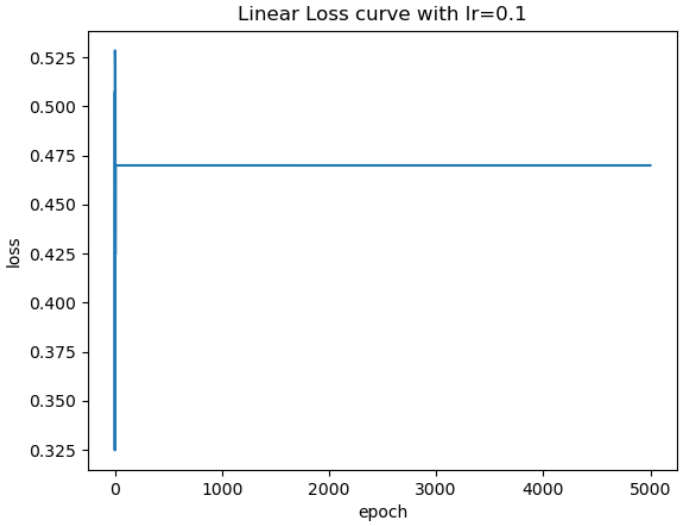

```
epoch 500 XOR loss : 0.008055228621595944
epoch 1000 XOR loss : 0.0022167607441936927
epoch 1500 XOR loss : 0.0012842763814787102
epoch 2000 XOR loss : 0.0008974228617941828
epoch 2500 XOR loss : 0.0006866390076633313
epoch 3000 XOR loss : 0.0005545081839955889
epoch 3500 XOR loss : 0.0004641694289802641
epoch 4000 XOR loss : 0.0003986247430381257
epoch 4500 XOR loss : 0.0003489651795751185
epoch 5000 XOR loss : 0.00031007877544335854
```

由上面四張圖可知，在訓練簡單的線性資料時，其收斂的速度超級快，且都很平緩; 反之，在訓練 XOR 這種非線性資料時，其收斂速度較低，且有部分地方有抖動，這應該是因為 local minimun 的問題，需要花多點時間才能收斂出來。

4. Discussion

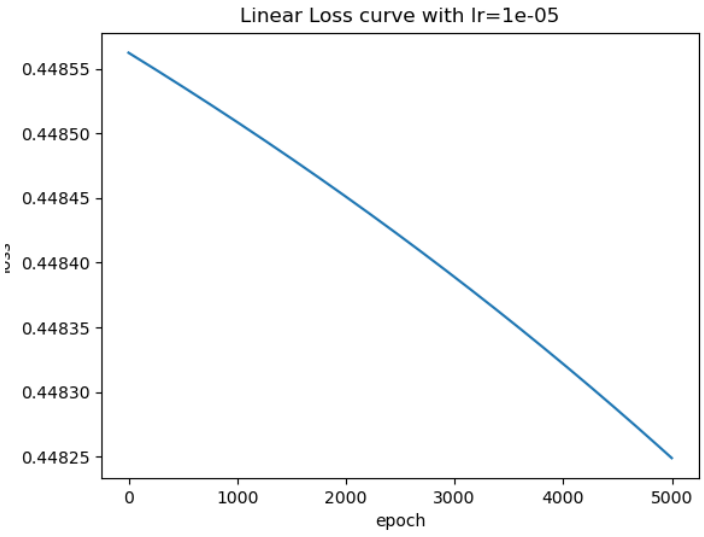
A. Try different learning rates

i. Learning rate :0.1



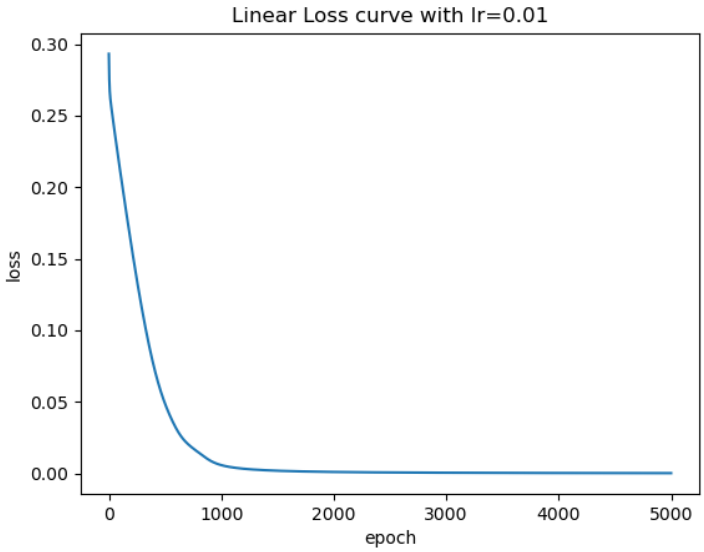
上面為線性輸入時的 loss 曲線圖，由於 learning rate 過大，會導致其不斷在最小值附近震盪，導致無法收斂，因此還需要再將 learning rate 調低。

ii. Learning rate:0.00001



上面為 learning rate 過小時會發生的問題，會造成收斂速度過慢，且容易卡在 local minimal，因此還需要將 learning rate 調高。

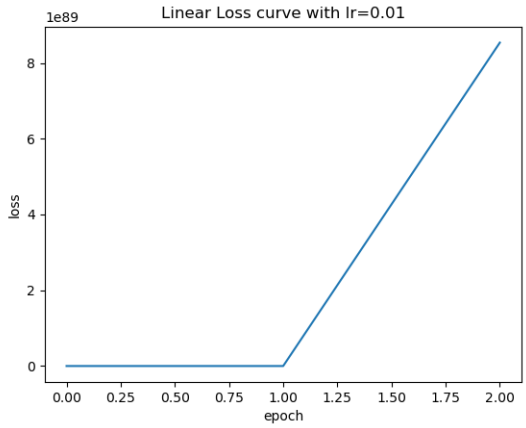
B. Try different numbers of hidden units

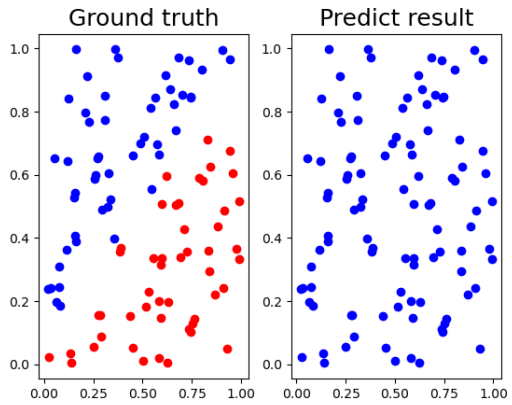


上面為 hidden layer 都為 5、輸入為 linear 的 loss 曲線圖，與原本 hidden layer 為 50 的狀況相比，其收斂速度大幅下降，我認為是因為中間的隱藏 weight 變少所導致，需要各個 weight 更加準確才有辦法預測出來。

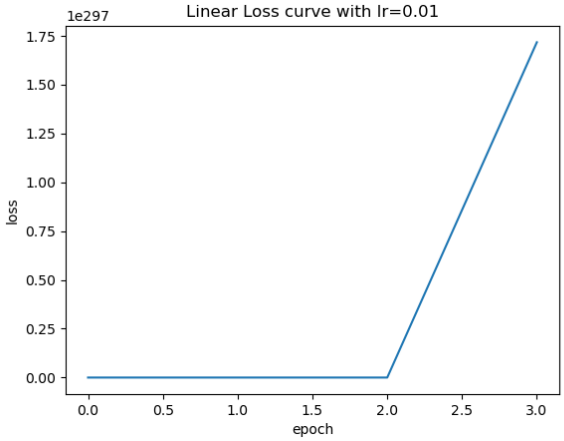
C. Try without activation functions

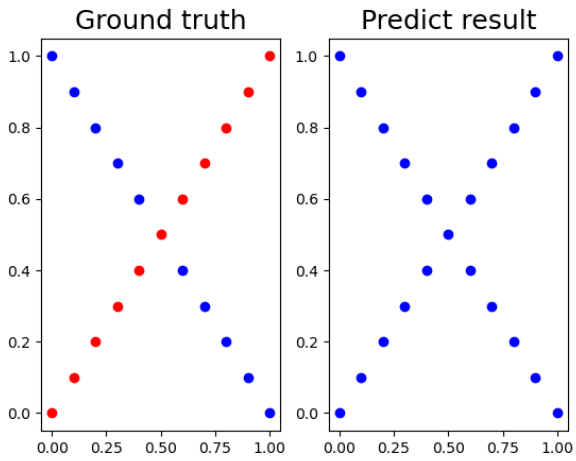
i. Linear





ii. XOR





由於這邊並沒有使用 active function，導致無法預測非線性的輸入(active function 使 weight 能夠處理非線性的特徵)，因此 active function 很有存在的必要。

5. Extra

A. Implement different optimization

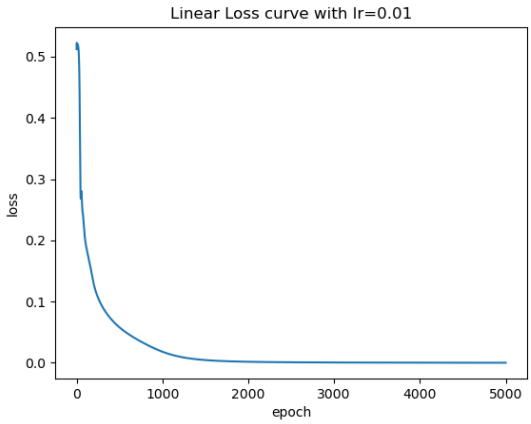
這邊使用 Adam 來配合 momentum、beta1 與 beta2 來動態調整 learning rate，使其不會被 local minimal 給卡住。下面為程式碼:

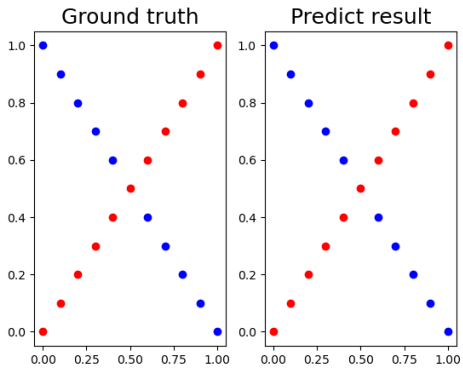
```
class Adam_class:
    def __init__(self, learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-8):
        self.learning_rate = learning_rate
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.m = None
        self.v = None
        self.t = 0

    def update(self, w, grad_wrt_w):
        self.t += 1
        if self.m is None:
            self.m = np.zeros_like(w)
            self.v = np.zeros_like(w)

        self.m = self.beta1 * self.m + (1 - self.beta1) * grad_wrt_w
        self.v = self.beta2 * self.v + (1 - self.beta2) * (grad_wrt_w ** 2)
        m_hat = self.m / (1 - self.beta1 ** self.t)
        v_hat = self.v / (1 - self.beta2 ** self.t)
        w -= self.learning_rate * m_hat / (np.sqrt(v_hat) + self.epsilon)
        return w
```

```python
if (self.optimization == "Adam"):
    self.adam_w3 = Adam_class()
    self.adam_w2 = Adam_class()
    self.adam_w1 = Adam_class()
    self.adam_b3 = Adam_class()
    self.adam_b2 = Adam_class()
    self.adam_b1 = Adam_class()
    self.beta1 = 0.9
    self.beta2 = 0.999
    self.epsilon=1e-8
    self.m = None
    self.v = None
    self.t = 0
```

```python
# #updating weight
if (self.optimization == "Adam"):
    self.w3 = self.adam_w3.update(self.w3, w3_grad)
    self.w2 = self.adam_w2.update(self.w2, w2_grad)
    self.w1 = self.adam_w1.update(self.w1, w1_grad)
    self.b3 -= self.adam_b3.update(self.b3, w3_forward)
    self.b2 -= self.adam_b2.update(self.b2, w2_forward)
    self.b1 -= self.adam_b1.update(self.b1, w1_forward)
```

以下為在 XOR 下的訓練情況:





與 SGD 相比，Adam 的收斂速度較慢，是因為動態調整 learning rate 的緣故，但卻可以更好地適應具有不同梯度範圍的參數; 反之，SGD 對於較小的數據集或較簡單的模型時，表現較佳。

B. Implement different activation functions

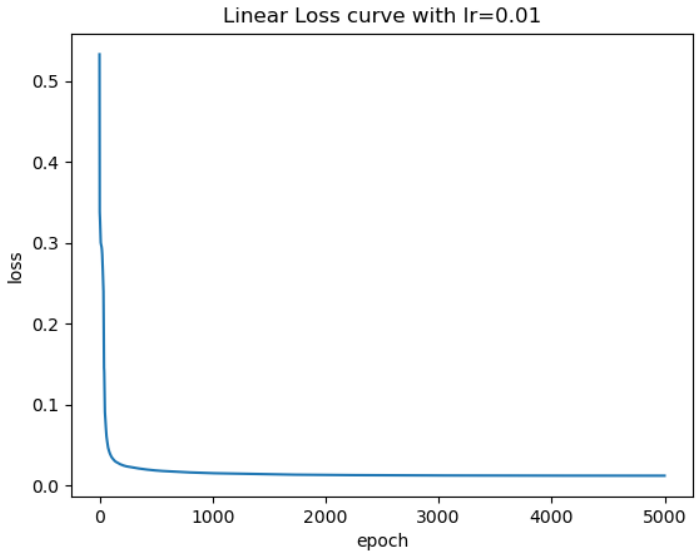
這邊有使用 Relu 來做 activation fumction，但由於其位於零點時會求不出斜率(nan)的問題，這邊還使用 Adam 來當作 optimization，起可以避免此情況的發生。以下為程式碼:

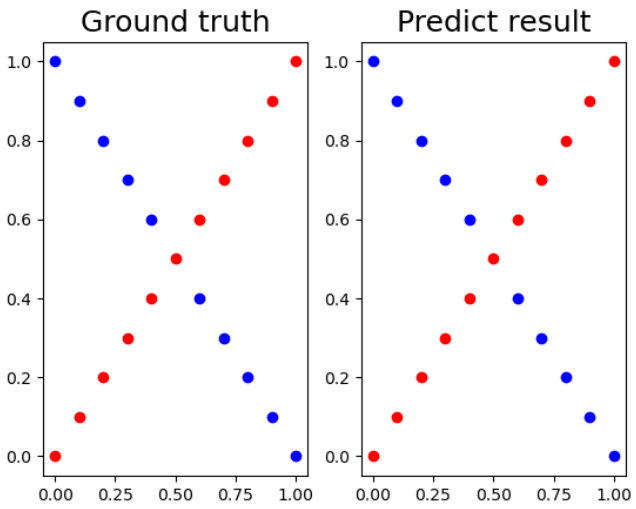
```
elif self.active_function == "Relu":
    self.x1 = np.dot(self.x_train, self.w1) + self.b1
    # self.x1 = np.dot(self.x_train, self.w1)
    self.a1 = self.Relu(self.x1)
    self.x2 = np.dot(self.a1, self.w2) + self.b2
    # self.x2 = np.dot(self.a1, self.w2)
    self.a2 = self.Relu(self.x2)
    self.x3 = np.dot(self.a2, self.w3) + self.b3
    # self.x3 = np.dot(self.a2, self.w3)
    self.y_pred = self.sigmoid(self.x3)
```

```
elif self.active_function == "Relu" :
    loss_grad = -2*(self.y_train - self.y_pred)
    w3_forward = loss_grad * self.derivative_sigmoid(self.y_pred)
    w3_grad = np.dot(self.a2.T, w3_forward)
    w2_forward = w3_forward.dot(self.w3.T)*self.derivative_Relu(self.a2)
    w2_grad =  np.dot(self.a1.T, w2_forward)
    w1_forward = np.dot(w2_forward,self.w2.T)*self.derivative_Relu(self.a1)
    w1_grad = np.dot(self.x_train.T,w1_forward)
```

另外，由於其一半為線性的特性，在最後一層的 active function 我使用 sigmoid 來達成，避免過度的線性導致無法對 XOR 這種非線性的數據集做預測。

以下為在 XOR 資料集的結果:





因為拿來訓練的資料並不複雜，使用 Relu 這種半線性的 active fumction 還能夠成功，另外，也可以使用 Leakly Relu 來避免斜率計算不出來的問題。