Introduction:

在這個實驗，我們只能用Numpy和Python標準的libraries來實作一個能前後傳播的網絡，網絡要求是利用兩層隱藏層學習兩個輸入的特徵來判斷正確的輸出。

以兩個亂數輸入作為座標X和Y，然後通過兩個有權重的隱藏層，繼而輸出比對實際結果得出誤差，從而反向回饋至隱藏層來更新權重，最後反覆進行前後傳播和權重更新，直到得出的預期輸出和實際輸出的誤差收斂。

在這個實驗中向前傳播是簡單的線性函數，難點在於反向傳播的理解、實作和權重更新的方法，這就是為什麼這個實驗叫做Backpropagation，下面的實驗設置也會對這次的實驗步驟進行講解。

Experiment Setup:

A. Sigmoid Function:

從數學上理解:

sigmoid 函數: $\sigma(\mathrm{x}) = \frac{1}{1+e^{-x}}$

sigmoid 函數的導數及其推導如下：$\sigma'(\mathrm{x}) = \left(\frac{1}{1+e^{-x}}\right)'$

$$\sigma'(\mathrm{x}) = \left(\frac{1}{1+e^{-x}}\right)' = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{e^{-x}+1-1}{(1+e^{-x})^2} = \frac{e^{-x}+1-1}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}}\left(1-\frac{1}{1+e^{-x}}\right)$$
$$= \sigma(\mathrm{x})\big(1-\sigma(\mathrm{x})\big)$$

其實就是其自身乘上1-sigmoid，至於為什麼要得出sigmoid函數的導數呢？下面Backpropagation會進行講解。

在圖形上的理解：

sigmoid函數是從0到1的一個軟飽和激活函數如下圖圖1，導數一般來說就是表達連續函數的變化量，在簡單的直線方程y=wx+b，則是斜率w，其表達x變化時y所對應的變化量。所以，當sigmoid函數在趨向於0或1時其 $y = \sigma(\mathrm{x})$對應的變化也是十分少，意味著w趨向於0，而從0變化到1的過程中，變化是最為劇烈的，相對其變化量也會漸漸增大，直到中間σ(x)=0.5時達到最高，然後慢慢減少，最後趨向於0。其導數的圖示如下圖圖2。

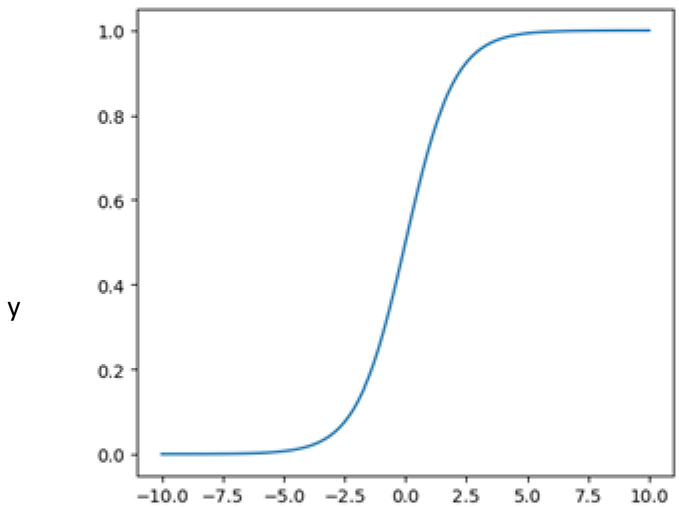


圖 1

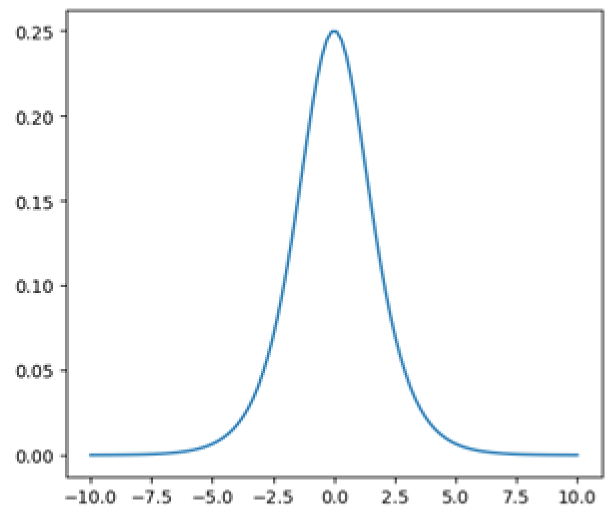


圖 2

x

在程式中我定義了sigmoid的函數如下：

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


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

B. Neural Network:

神經網絡顧名意思就是又神經元構成的網絡，每一個神經元在數學上就是一個函數。而在本次的作業中，函數就是sigmoid，用多個不同參數/變化率的神經元來構成隱藏層，從而建立起神經網絡。以下圖3是本次作業的神經網絡結構：

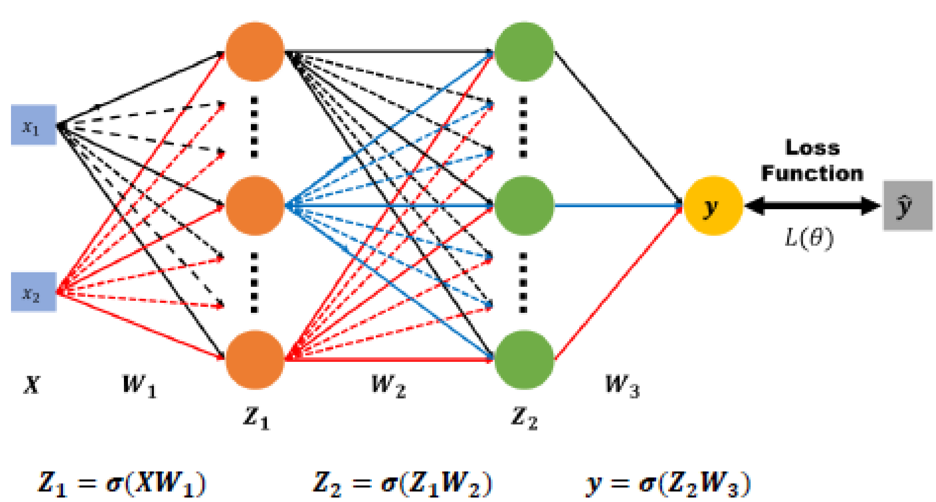


圖3

在圖中我們可以得知網絡的輸入有2個x_1和x_2，再透過簡單的直線方程W_1X輸入到第一層隱藏層，得出$Z_1 = \sigma(XW_1)$，繼而再用另一個直線方程W_2Z_1輸入到第二層隱藏層輸出$Z_2 = \sigma(Z_1W_2)$，最後再通過W_3Z_2來得出神經網絡輸出$y = \sigma(Z_2W_3)$。

傳播過程為下：

$$X \longrightarrow X_1 = XW_1 \longrightarrow Z_1 = \sigma(X_1) \longrightarrow X_2 = Z_1W_2 \longrightarrow Z_2 = \sigma(X_2) \longrightarrow X_3 = Z_2W_3 \longrightarrow y = \sigma(X_3)$$

所以實際上的程式碼也是沿著這個傳播來設計，以下為X輸入至第一層隱藏層Layer前的向前傳播($X \rightarrow XW_1$)：

```
class SigmoidLayer:
    def __init__(self):
        pass

    def forward(self, x):
        return sigmoid(x)
```

在乘上參數W_1後，再通過隱藏層的sigmoid 函數來輸出至第二層的隱藏層($XW_1 \rightarrow Z_1 = \sigma(XW_1)$)，其中sigmoid的程式碼在上面A. Sigmoid Function 已有提及：

```
class Layer:
    def __init__(self, input_size, output_size, lr, dataset_number=1):
        self.input_size = input_size
        self.output_size = output_size
        self.lr = lr  # learning rate
        np.random.seed(dataset_number)
        self.w = np.random.rand(input_size, output_size)  # create random weight according
to layer size
        self.b = np.zeros(output_size)  # bias

    def forward(self, x):
        # print('input: {}, weight: {}'.format(x.shape, self.w.shape))
        return np.add(np.dot(x, self.w), self.b)  # wx+b
```

後面其實就是重覆以上兩個步驟($X \rightarrow XW_1 \rightarrow Z_1 = \sigma(XW_1)$)直至得出神經網絡輸出($y = \sigma(Z_2W_3)$)，所以在程式碼上我用了一個Class Network 把上面提及到的Sigmoid Layer 和 Layer組合起來，組合程式碼如下：

```
class Network:
    def __init__(self, input_size, hidden_layer_size, output_size, learning_rate,
activation_function):
        # input size = 2, output size = 1, total layer = 2, hidden layer size = any
        self.input_size = input_size
        self.hidden_layer_size = hidden_layer_size
        self.output_size = output_size
        self.learning_rate = learning_rate
        if activation_function == 'Sigmoid':
            activation_layer = SigmoidLayer()
        elif activation_function == 'Relu':
            activation_layer = ReluLayer()
        else:
            activation_layer = NoActivationFunction()
        self.network = [Layer(input_size, hidden_layer_size[0], learning_rate),
activation_layer,  # Input layer, sigmoid
                        Layer(hidden_layer_size[0], hidden_layer_size[1], learning_rate),
activation_layer,  # hidden layer, sigmoid
```

```
                        Layer(hidden_layer_size[1], output_size, learning_rate),
                                 activation_layer]  # hidden layer, sigmoid(output)
        # network process: x -> 1st layer (wx -> z1=Wx -> a1=sigmoid(z1)) -> 2nd layer ->
z2=Wa2 -> output=sigmoid(a2)

    def forward(self, x):
        layer_output = []
        data_input = x
        for each_layer in self.network:  # to load each layer of network
            layer_output.append(each_layer.forward(data_input))  # append output data to
layer_output
            data_input = layer_output[-1]  # last column of output data is next input data
        layer_output = [x] + layer_output  # add input data to output data for later use
        return layer_output
```

其中activation_layer指的是激活函數，這裡可先以sigmoid代入理解，leraning_rate學習率會在Backpropagation處在再解釋。

$$X \longrightarrow X_1 = XW_1 \longrightarrow Z_1 = \sigma(X_1) \longrightarrow X_2 = Z_1W_2 \longrightarrow Z_2 = \sigma(X_2) \longrightarrow X_3 = Z_2W_3 \longrightarrow y = \sigma(X_3)$$

以上就建立了一個簡單的神經網絡結構，但這時這個神經網絡其實並不完整，因為它只是一個簡單的直線方程和Sigmoid函數的組合，要使神經網絡學習就要讓他知道什麼時候要學習，要學習什麼？下一部分就會對學習進行講解。

C. Backpropagation:

反向傳播意思是反向傳播神經網絡，直觀的理解是以y視為輸入，X作為輸出，把神經網絡倒過來看。這其實是因為我們想要神經網絡學習，一般而言，神經網絡都會有訓線資料和測試資料，這其實是想給神經網絡學習輸入和輸出的關係所需的資料。

直觀的理解是我們知道直線方程式Y=WX的輸入X和輸出Y，就能求出W的值作為這個函數的參數，在下一個X輸入時就能得到輸出Y。如上所述，我們會有訓練資料，知道確實的相應的輸出答案，根據這個真實正確的答案與我們神經網絡的預測輸出答案作出比較，這樣我們就知道其中的誤差為多少，再依照這個差距來調整我們的網絡。

這裡我會多標記一個函數X_i方便一步一步理解(見下圖4)：

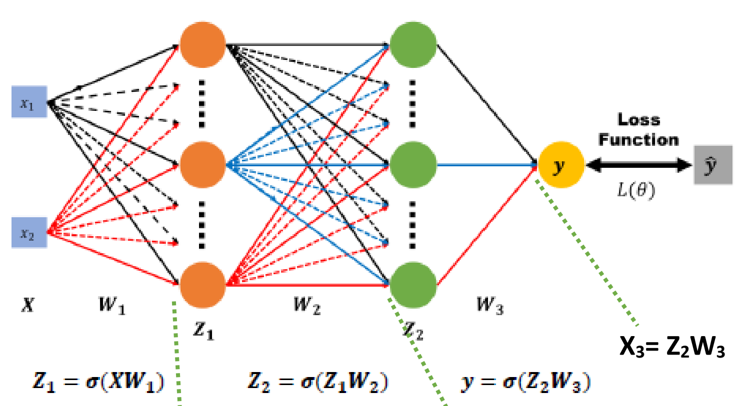


圖4

實際數學上要如何做更新，由於我們得知訓練資料的正確答案和神經網絡的預測答案，我們可以求出正確和預測的差，即誤差Loss，這裡我們用較為簡單的Mean Square Error(MSE)作為 Loss $L = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$來表達，其中$\hat{y}$為正確輸出，$y$為預測輸出。藉著MSE我們可以得到y的變化量對L的影響：

$$\frac{\partial L}{\partial y} = \frac{2}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i) \tag{1}$$

因為$y = \sigma(X_3)$ ，我們可以得到：

$$\frac{\partial y}{\partial X_3} = \sigma(X_3)\cdot(1-\sigma(X_3)) \tag{2}$$

又因為$X_3 = Z_2W_3$，我們可以得到：

$$\frac{\partial X_3}{\partial Z_2} = \frac{\partial(Z_2W_3)}{\partial Z_2} = W_3 \tag{3}$$

為了得到W的變化量對C的影響，我們還需要：

$$\frac{\partial Z_2}{\partial X_2} = \sigma(\mathrm{X}_2)\cdot\left(1-\sigma(\mathrm{X}_2)\right) \tag{4}$$

$$\frac{\partial X_2}{\partial W_2} = \frac{\partial(Z_1W_2)}{\partial W_2} = Z_1 \tag{5}$$

結合(1)、(2)、(3)、(4)、(5)，我們可以得到X_2和W_2的變化量對C的影響 ：

$$\frac{\partial L}{\partial X_2} = \frac{\partial Z_2}{\partial X_2}\frac{\partial X_3}{\partial Z_2}\frac{\partial y}{\partial X_3}\frac{\partial L}{\partial y} \tag{6}$$

$$\frac{\partial L}{\partial W_2} = \frac{\partial X_2}{\partial W_2}\frac{\partial L}{\partial X_2} \tag{7}$$

然後根據得出的變化量來更新神經網絡內的參數W_2，同樣使用較為簡單的線性方程，其中lr為learning rate，用來決定每次更新參數的幅度：

$$W_2 = W_2 - lr * \frac{\partial L}{\partial W_2} \tag{8}$$

Z_1的變化量對X_2的影響：

$$\frac{\partial X_2}{\partial Z_1} = \frac{\partial(Z_1W_2)}{\partial Z_1} = W_2 \tag{9}$$

X_1的變化量對Z_1的影響，W_1的變化量對X_1的影響：

$$\frac{\partial Z_1}{\partial X_1} = \sigma(\mathrm{X}_1)\cdot(1-\sigma(\mathrm{X}_1)) \tag{10}$$

$$\frac{\partial X_1}{\partial W_1} = \frac{\partial(XW_1)}{\partial W_1} = X \tag{11}$$

結合(6)、(9)、(10)、(11)，我們可以得到W_1的變化量對C的影響 ：

$$\frac{\partial L}{\partial W_1} = \frac{\partial X_1}{\partial W_1}\frac{\partial Z_1}{\partial X_1}\frac{\partial X_2}{\partial Z_1}\frac{\partial L}{\partial X_2}$$

同樣用(8)來更新參數W_1：

$$W_1 = W_1 - lr * \frac{\partial L}{\partial W_1} \tag{12}$$

由上述推導的式子可發現(2)(3)和(4)(5)、(10)(11)其實是不斷重複的，所以我在程式中把他們整合起來，如下：

```
class SigmoidLayer:
        # :
def backward(self, x, d_input):
    return derivative_sigmoid(x) * d_input  # derivative output layer dC/dz = dC/dy * dy/dz
chain rule
# dy/dz - d_fwd(d_sigmoid), d_input, dC/dy - (y_pred - y_hat)
```

```
class Layer:
    # :
def backward(self, fwd_input, d_input):  # forward input and derivative input
    # print('w: {}, d_input: {}'.format(self.w.shape, d_input.shape))
    d_output = d_input @ self.w.T  # array_Y * w_Transpose as Y and w is (y_size, 1) array
    # print('fwd_input: {}, d_input: {}'.format(fwd_input.shape, d_input.shape))
    d_w = fwd_input.T @ d_input  # fwd_input is derivative sigmoid function
    d_b = d_input.mean(axis=0)  # take mean value at axis = 0
    # print('fwd_input: {}, d_input: {}'.format(fwd_input.shape, d_input.shape))
    # print('weight: {}, d_b: {}, w: {}'.format(d_w.shape, d_b.shape, self.w.shape))
    self.w += -self.lr * d_w
    self.b += -self.lr * d_b
    return d_output
```

```
class Network:
    # :
def backward(self, fwd_input, d_input):
    for each_layer in reversed(range(len(self.network))):  # to load each layer inside
network, backward
        d_input = self.network[each_layer].backward(fwd_input[each_layer], d_input)
        # fwd_input by recorded data from forward calculation on each layer
        # d_input = derivative input result
```

通過Network中定義的網絡，Network backward只需要反序的跑每一層的backward function（如下），而在每一層的網絡中，把上面推導的公式對應寫入backward的function中，Network便會根據排列順序執行，最後再建立一個迴圈輸入訓練所需的次數epoch便可以不停更新參數直到訓練次數結束。

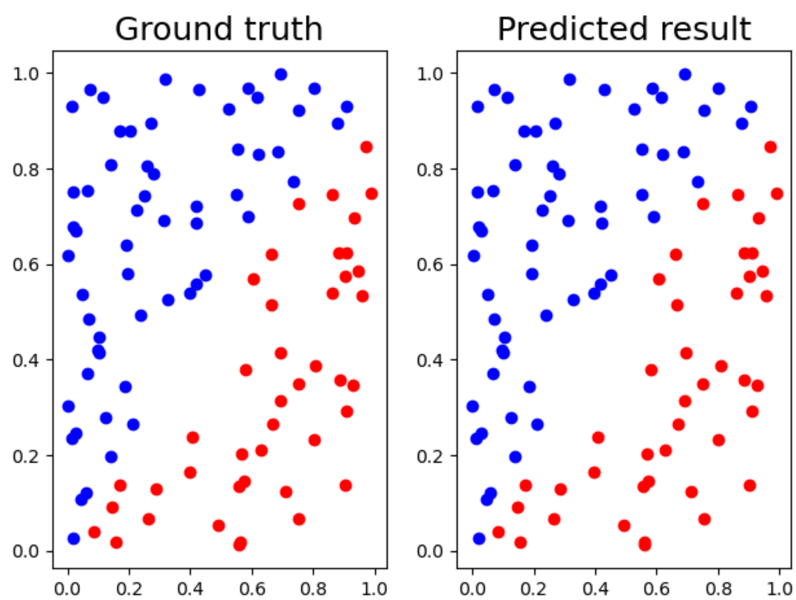
$$X \longleftarrow X_1 = XW_1 \longleftarrow Z_1 = \sigma(X_1) \longleftarrow X_2 = Z_1W_2 \longleftarrow Z_2 = \sigma(X_2) \longleftarrow X_3 = Z_2W_3 \longleftarrow y = \sigma(X_3)$$

Results of testing

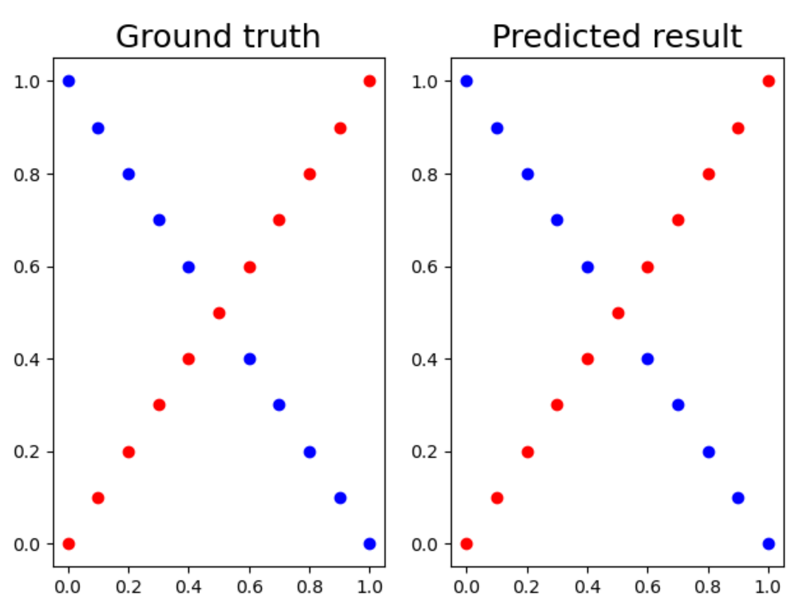
A. Screenshot and Comparison Figure

Learning rate = 1, Hidden Layer Unit = 4, 8 (默認值，如後面沒有提及的參數用這個設定)

Linear:



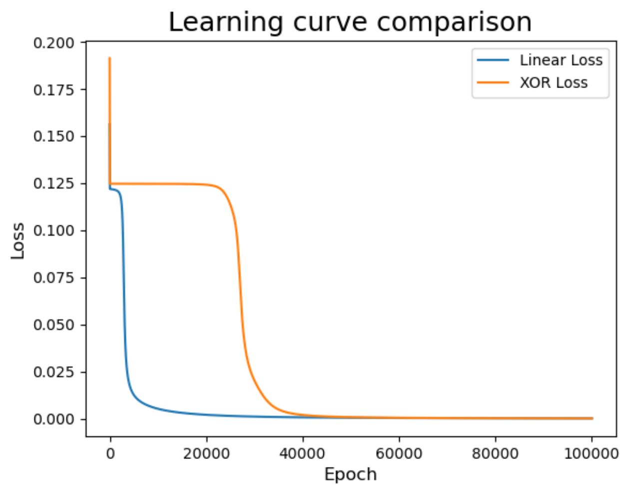
XOR:



B. Show the accuracy of prediction

```
Linear accuracy: 100.00%
XOR accuracy: 100.00%
```

C. Learning curve (loss, epoch curve)

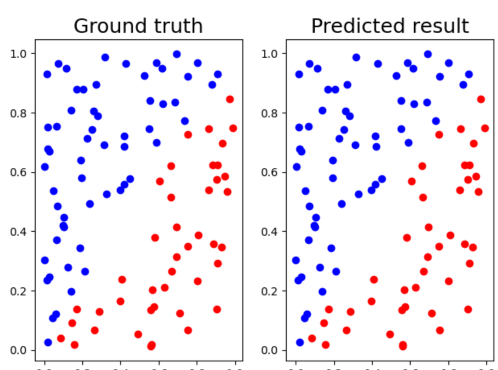


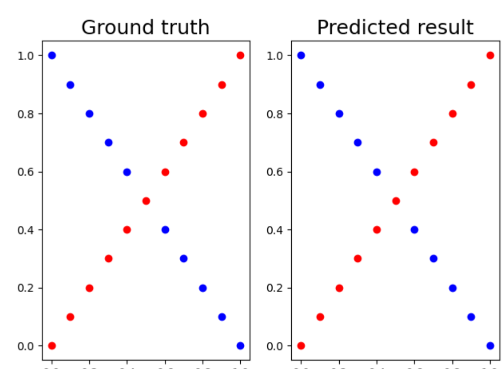
Learning rate = 1

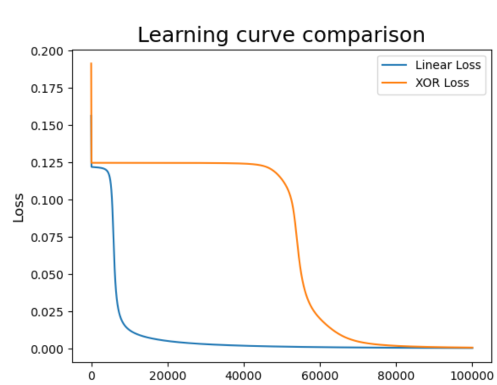
Discussion

A. Try different learning rates

Learning rate = 0.5



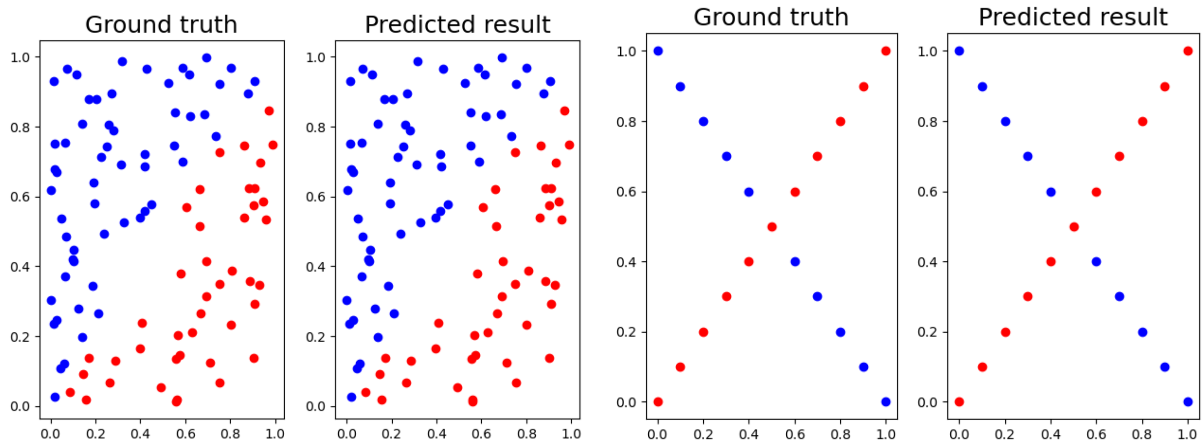


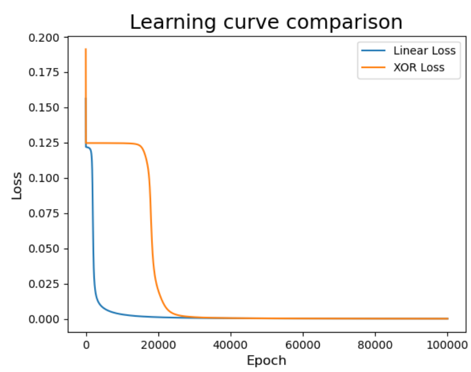


Linear accuracy: 100.00%

XOR accuracy: 100.00%

Learning rate = 1.5





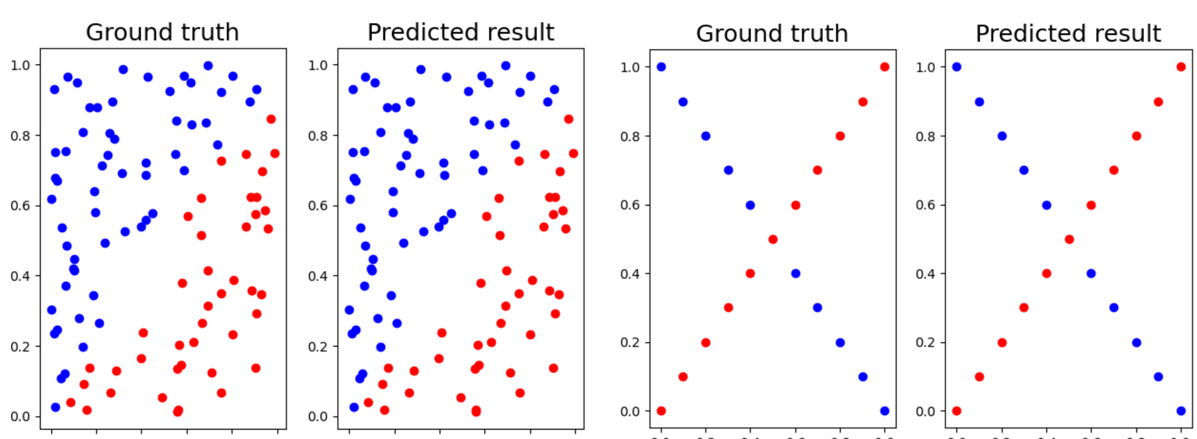
Linear accuracy: 100.00%

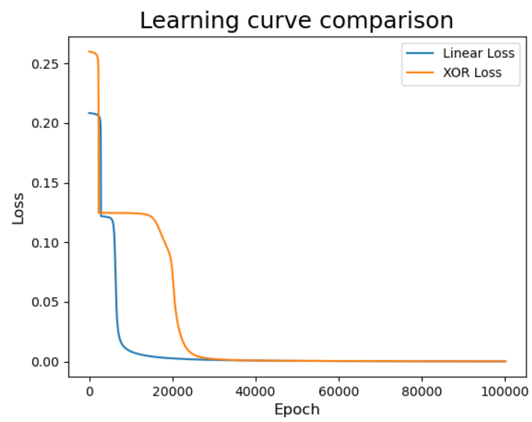
XOR accuracy: 100.00%

很明顯的可以看到，不同learning rate 的學習速度是不一樣的，learning rate 越大學習越快，learning rate 越小學習越慢。

B. Try different numbers of hidden units

Hidden Layer Unit = 8, 16

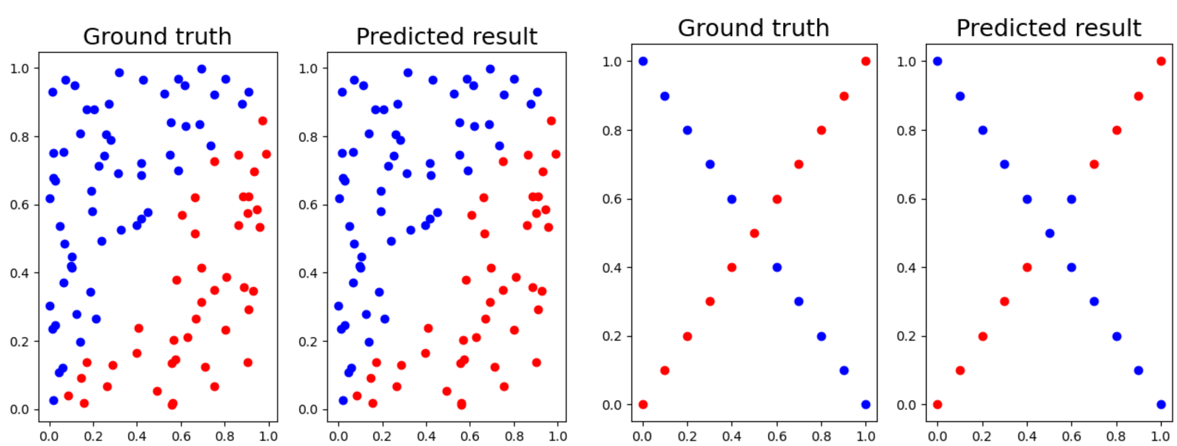


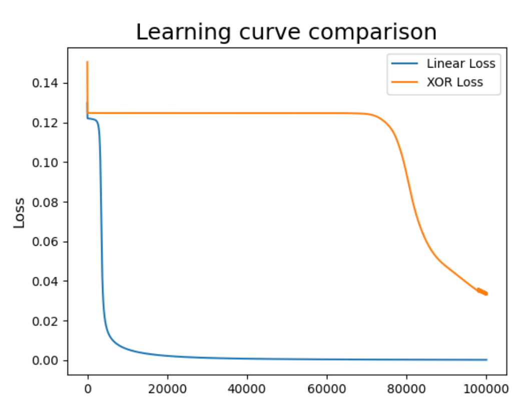


Linear accuracy: 100.00%

XOR accuracy: 100.00%

Hidden Layer Unit = 2, 4



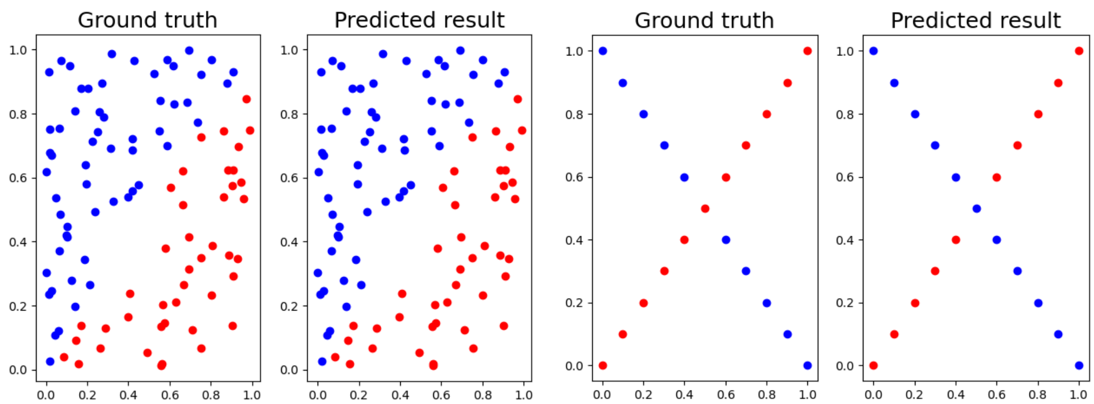


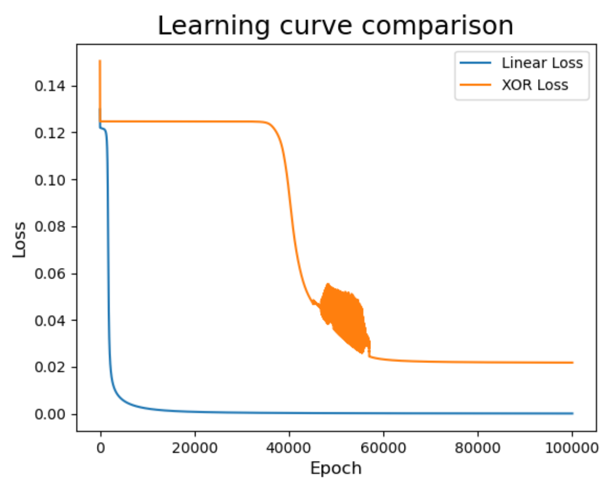
Linear accuracy: 100.00%

XOR accuracy: 90.48%

越多隱藏層單位數會增加神經網絡的學習參數，從而加快學習的速度，反之亦然，可見太少的隱藏層神經元的學習速度很慢，要更多epoch來訓練。那麼，如果我加大學習率來加快學習速度又如何呢？

Hidden Layer Unit = 2, 4 with learning rate = 2



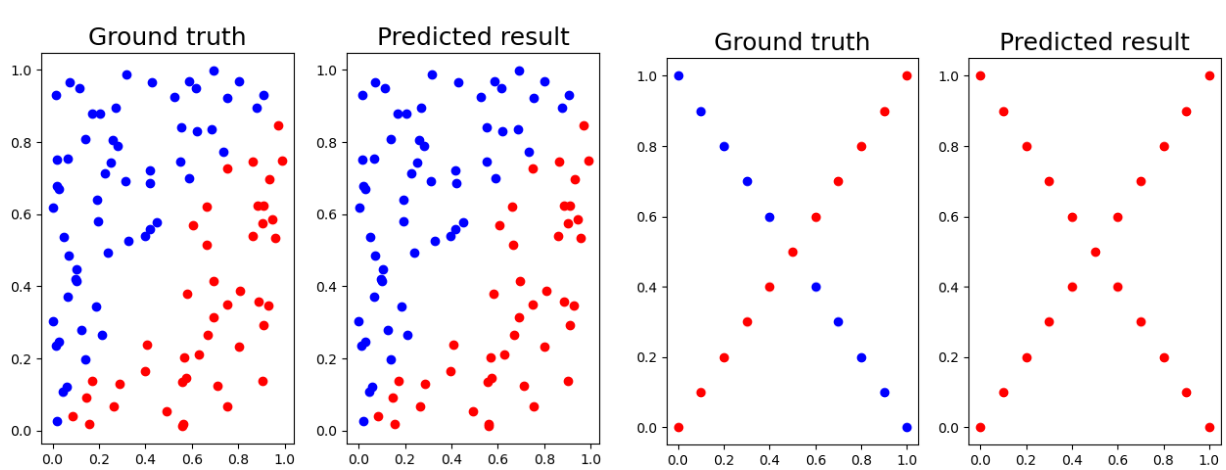


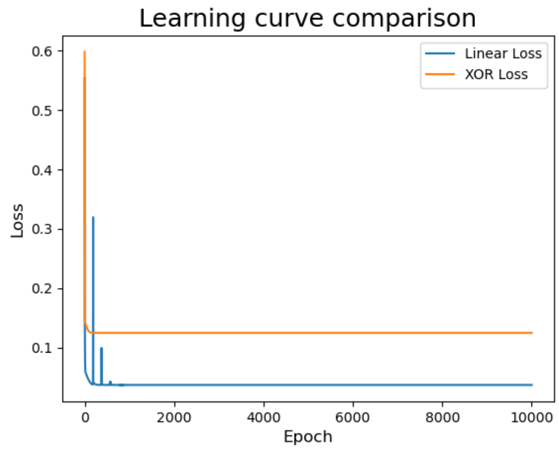
Linear accuracy: 100.00%

XOR accuracy: 95.24%

可以見到整體而言加大學習率是有幫助的，但我們發現在epoch 40000 to 60000 損失在上下跳動，這是很明顯的學習率過大導致的，看來這個微型網絡的極限就是這裡了。

C. Try without activation functions





Linear accuracy: 100.00%

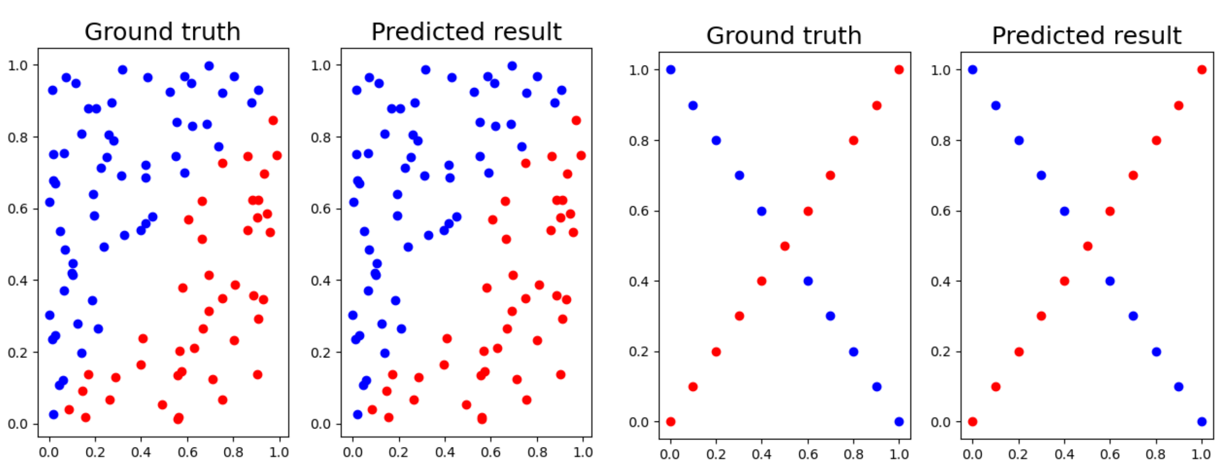
XOR accuracy: 52.38%

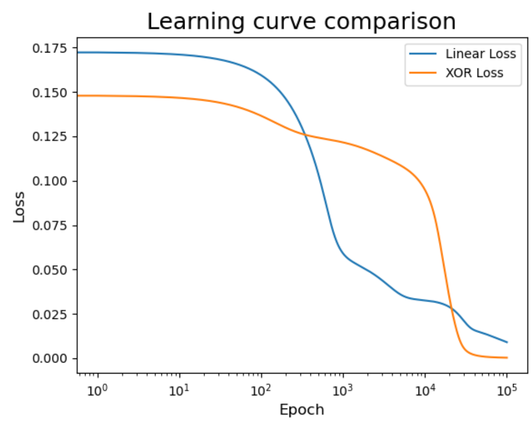
在沒有激活函數下linear data的學習仍然能保持水平，但XOR 的輸入側無法減少損失，原因是無論怎樣改變直線的斜率，都無法有效的區分交叉的數據，激活函數就是為了彈性地區分資料而加入的。

Extra

A. Implement different optimizers

Stochastic Gradient Descent (SGD) with activation function = tanh and lr = 0.03

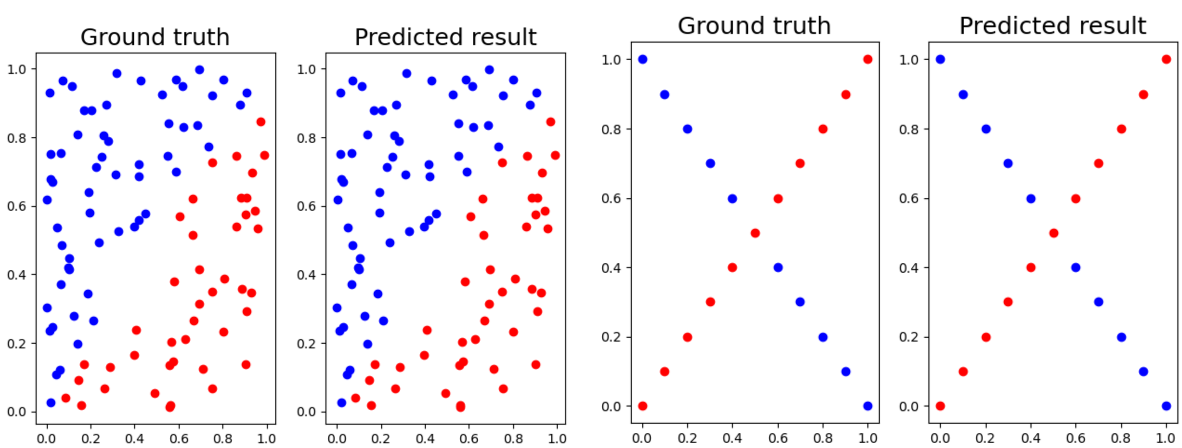


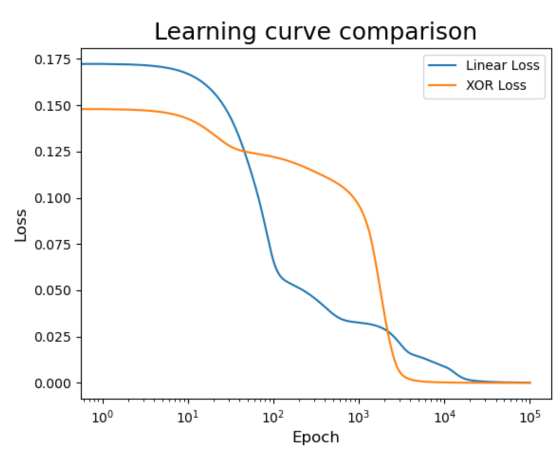


Linear accuracy: 100.00%

XOR accuracy: 100.00%

Momentum with activation function = tanh and lr = 0.03

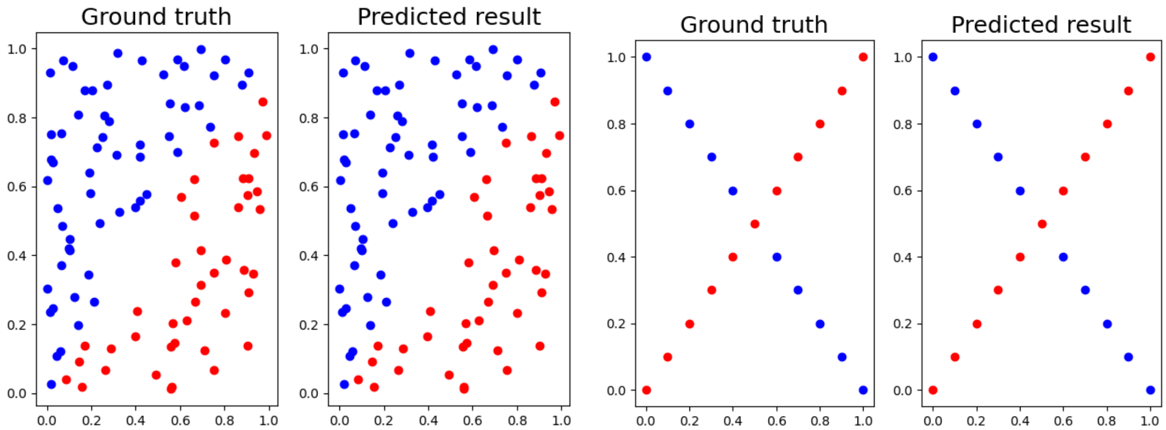


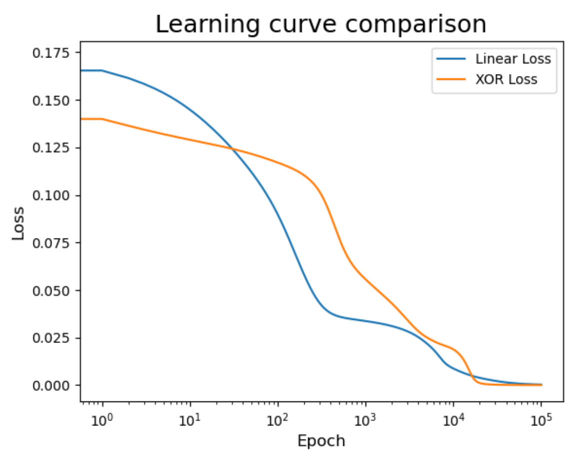


Linear accuracy: 100.00%

XOR accuracy: 100.00%

Adagrad with activation function = tanh and lr =0.03



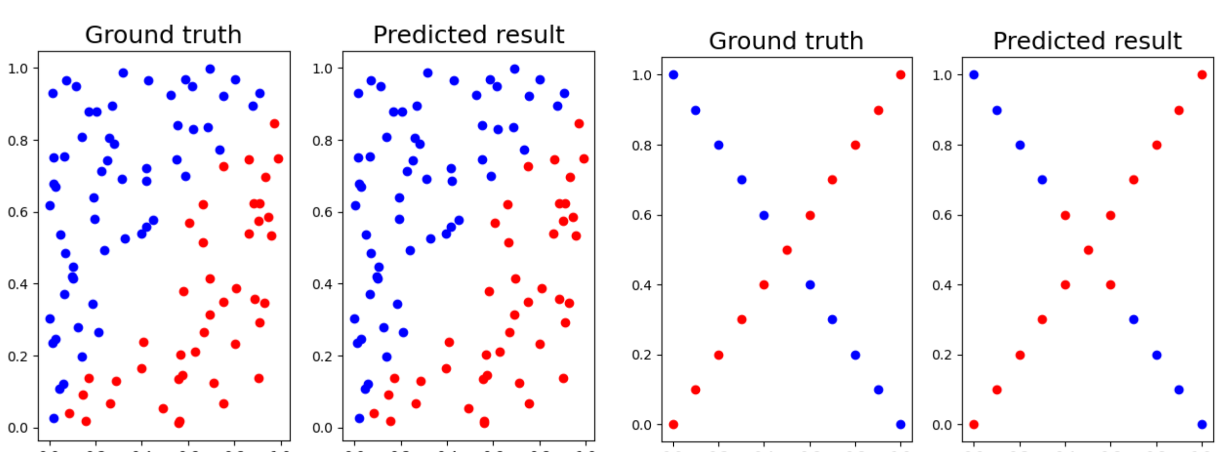


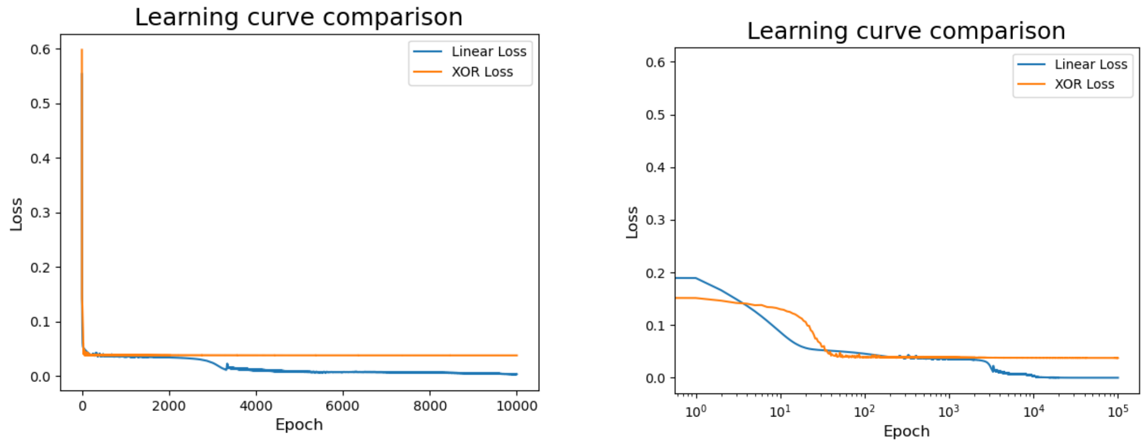

```
Linear accuracy: 100.00%
XOR accuracy: 100.00%
```

在比較後得出結論，Adagrad > Momentum > SGD

B. Implement different activation functions.

Relu:



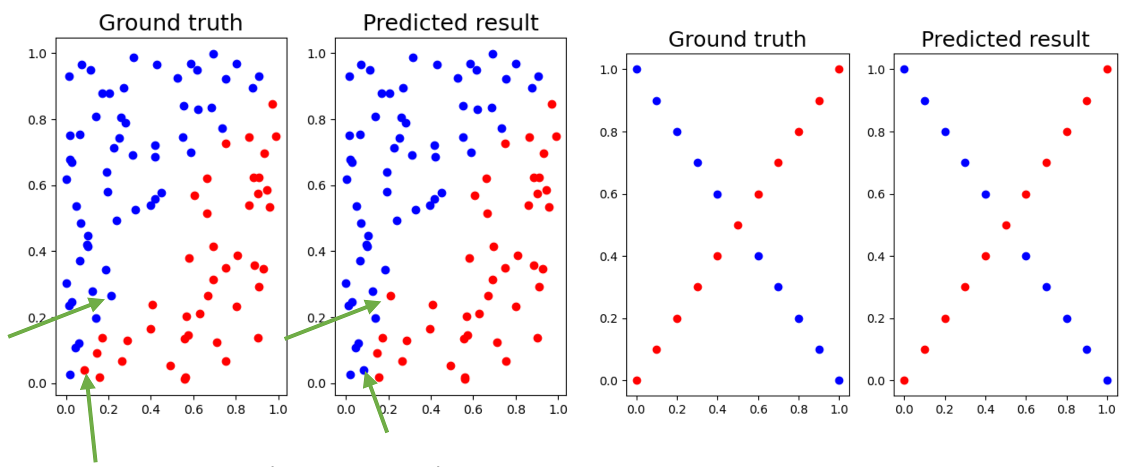


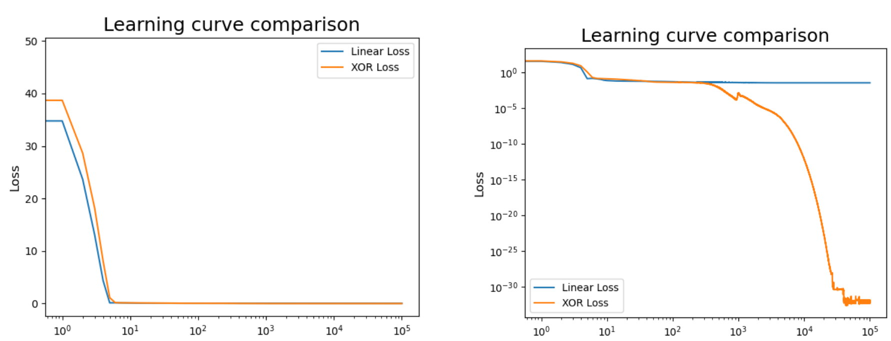
Linear accuracy: 100.00%

XOR accuracy: 90.48%

我們很明顯看到relu在和sigmoid相同設定環境下很快收斂，但一直沒法再下降，這有可能是參數不夠，或學習率太高。

調整Hidden Layer Unit = 8, 16



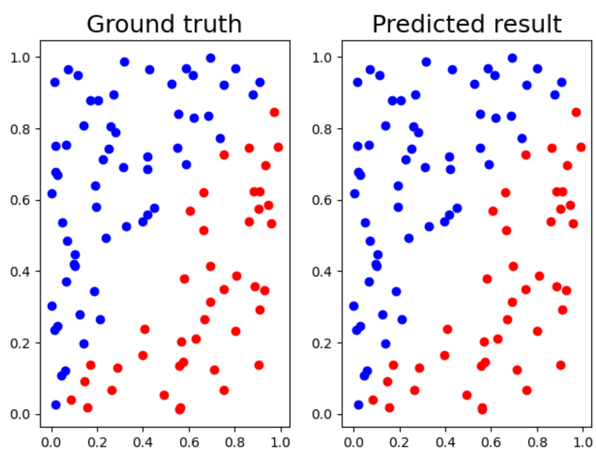


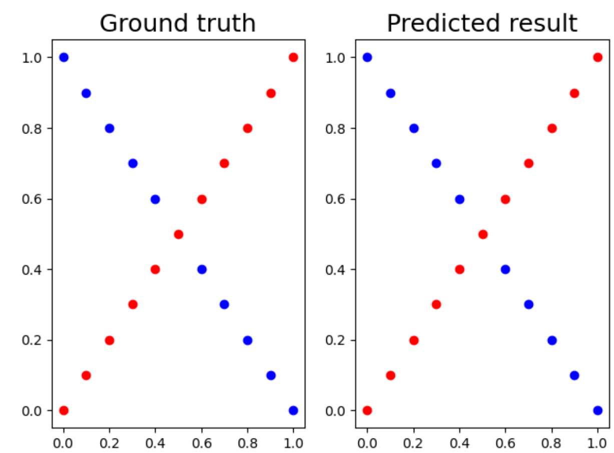
Linear accuracy: 96.00%

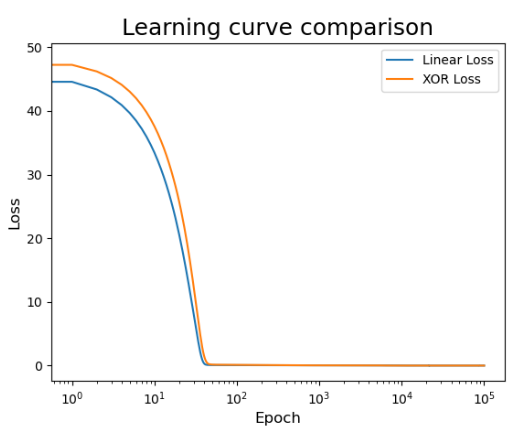
XOR accuracy: 100.00%

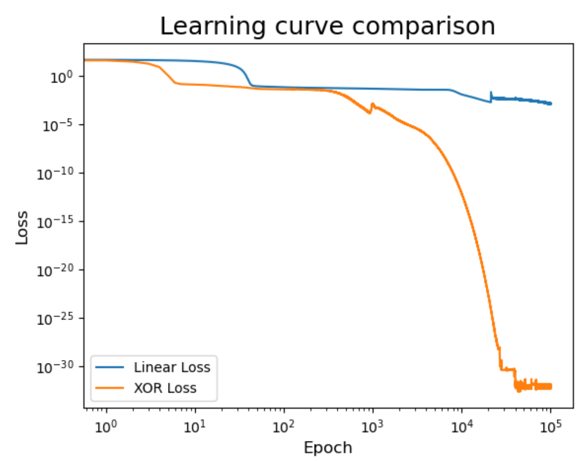
值得注意的是在Loss圖中看似趨近於0的linear loss，其實有2點並未完全正確，這可能表示learning rate 在這個新網絡有點太高。

調整Hidden Layer Unit = 8, 16, learning rate = 0.1







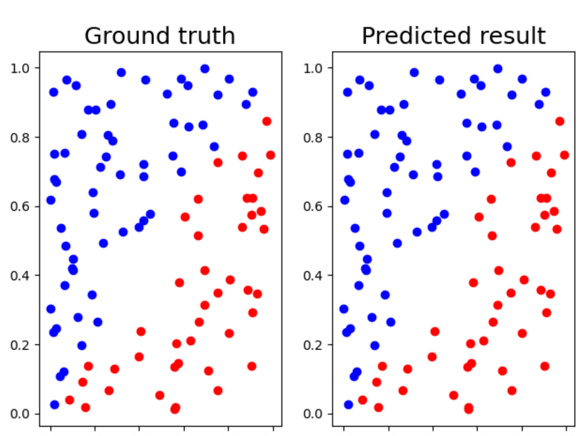


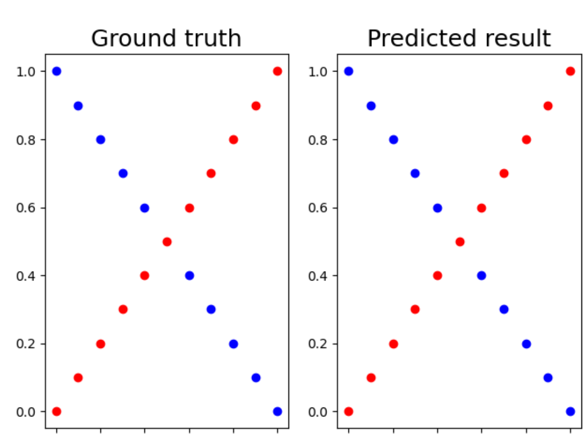
Linear accuracy: 100.00%

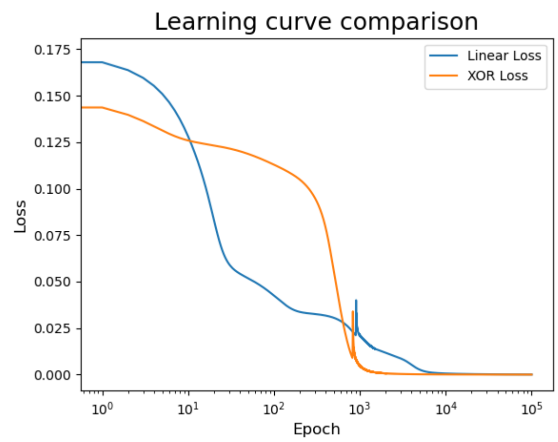
XOR accuracy: 100.00%

在調整linear data learning rate後，Loss在後面產生了動盪，看來是之前learning rate 太大，導致錯失了最優解的地方，調小學習率後成功降低損失loss，準確率也達100%了。

Tanh:







Linear accuracy: 100.00%

XOR accuracy: 100.00%

在比較後得出結論，Tanh > Relu > Sigmoid

此次作業結建了簡單的神經網絡，並運用了不一樣的元件和更改參數去觀察變化，大大提升了對神經網絡的實際了解，當中最難理解的backpropagation也花費很多時間才能確實理解，把進導過程詳細表達後更能體會各個參數對網絡的影響，當中learning rate、activation function、hidden layer unit、optimizer各施其職令神經網絡能有更好的改善是一人工智慧的一個理程碑。

後記，coding 可能和報告有出入，因為是邊做邊打的，請以 py 檔為準。