HW01

資工科碩一 殷暐智 313551087

Introduction

在本次實驗中為透過 numpy 來設計神經網路來分類 Input 資料，規則為須使用兩層 hidden layer 來實作，並包含神經網路中學習的細節，如 forward、Backpropagation、activate function、optimize 的實作。在 Input 資料的部分則是預先提供的 linear、XOR_easy，在實驗上分別使用兩者來進行不同的實驗設置來比較彼此在不同的 learning rates、hidden units、activation functions 下的差異性。

Experiment setups :

A. Sigmoid functions

Sigmoid 根據定義公式如下:

$$\sigma(X) = \frac{1}{1 + e^{-1}}$$

接下來也需實作 Sigmoid 的偏微分，根據此處得推導可以得出如下:

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$$

在程式的實作上就不用再添加 sigmoid()，因為 x 已是 sigmoid(X)的輸出

```
# Sigmoid
def sigmoid(x):
    return 1/(1 + np.exp(-x))
def sigmoid_derivative(x):
    return x * (1 - x)
```

B. Neural network

在神經網路的設計上使用兩層 hidden layer，分別包含 10 個 Neural，並且預設使用 10000 epoch、0.1 的 learning rate 和 sigmoid 作為 activation function。

```
hidden_size1 = 10
hidden_size2 = 10
epochs = 10000
learning_rate = 0.1
```

```
model = NeuralNetwork()
model.add_layer(Layer(input_size, hidden_size1))
model.add_layer(Layer(hidden_size1, hidden_size2))
model.add_layer(Layer(hidden_size2, output_size))
```

在計算 loss 上則使用 MSE 作為 loss 的計算方式

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

C. Backpropagation

在 Backpropagation 的部分透過 Neural Network 底下的 backward 去呼叫逐層 layer 進行 backward，由於每一層都是透過前一層計算出來的值去乘上 activation function 的偏微分去更新 w 和 b，最後一層則是透過計算y與$\hat{y}$之間的 loss 偏微分。

class Neural Network:

```
def backward(self, loss_derivative, learning_rate,optimize,decay_rate):
    # backward
    for layer in reversed(self.layers):
        loss_derivative = layer.backward(loss_derivative, learning_rate,optimize,decay_rate)
```

class Layer:

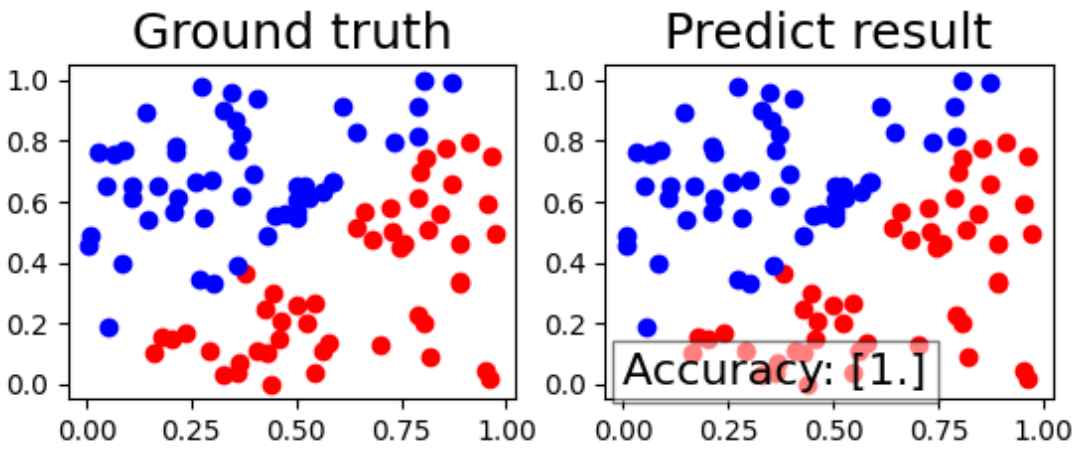
```
def backward(self, loss_derivative, learning_rate,optimize,decay_rate):
    if self.activation_function == "sigmoid":
        activation_derivative = sigmoid_derivative(self.output)
    elif self.activation_function == "ReLU":
        activation_derivative = ReLU_derivative(self.output)
    else:
        raise ValueError("Unsupported activation function")

    delta = loss_derivative * activation_derivative
    dw = np.dot(self.input.T, delta)
    db = np.sum(delta)
    if optimize == "SGD":
        self.W -= learning_rate * dw
        self.b -= learning_rate * db
    elif optimize == "momentum":
        self.v_w = decay_rate * self.v_w + learning_rate * dw
        self.W -= self.v_w
        self.v_b = decay_rate * self.v_b + learning_rate * db
        self.b -= self.v_b
    else:
        raise ValueError("Unsupported optimize function")
    return np.dot(delta, self.W.T)
```

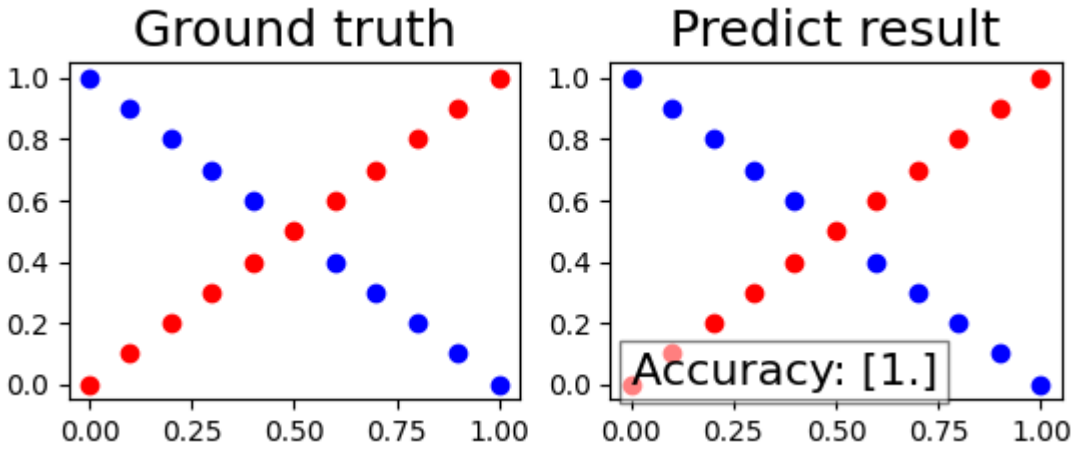
Results of your testing :

A. **Screenshot and comparison figure**

Linear data:



XOR data:



B. **Show the accuracy of your prediction**

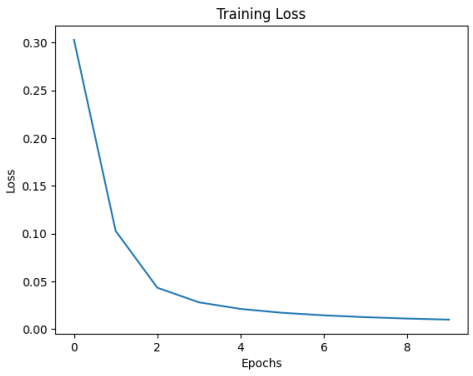
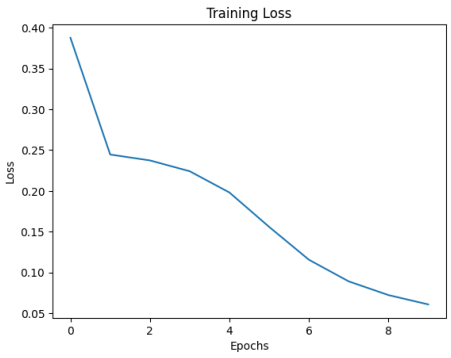
Linear data:

```
Accuracy: [1.]
Prediction :
Iter 1 | Ground truth: [1] | Prediction: [0.95912896]
Iter 2 | Ground truth: [1] | Prediction: [0.64641015]
Iter 3 | Ground truth: [1] | Prediction: [0.99958021]
Iter 4 | Ground truth: [1] | Prediction: [0.96738516]
Iter 5 | Ground truth: [1] | Prediction: [0.78663123]
Iter 6 | Ground truth: [1] | Prediction: [0.9957539]
Iter 7 | Ground truth: [0] | Prediction: [0.00042813]
Iter 8 | Ground truth: [0] | Prediction: [0.00090254]
Iter 9 | Ground truth: [0] | Prediction: [0.00187183]
Iter 10 | Ground truth: [0] | Prediction: [0.00629601]
```

XOR data:

```
Accuracy: [1.]
Prediction :
Iter 1 | Ground truth: [0] | Prediction: [0.05260297]
Iter 2 | Ground truth: [1] | Prediction: [0.98512438]
Iter 3 | Ground truth: [0] | Prediction: [0.0722881]
Iter 4 | Ground truth: [1] | Prediction: [0.9664167]
Iter 5 | Ground truth: [0] | Prediction: [0.13193126]
Iter 6 | Ground truth: [1] | Prediction: [0.9076514]
Iter 7 | Ground truth: [0] | Prediction: [0.25397888]
Iter 8 | Ground truth: [1] | Prediction: [0.74749249]
Iter 9 | Ground truth: [0] | Prediction: [0.37681844]
Iter 10 | Ground truth: [1] | Prediction: [0.51181465]
```

C. Learning curve (loss, epoch curve)

Linear data	XOR data
	
Epoch 0| Loss: 0.30290271591785506 | Accuracy: [0.5] Epoch 1000| Loss: 0.10289837945399788 | Accuracy: [0.97] Epoch 2000| Loss: 0.04341921615670831 | Accuracy: [0.99] Epoch 3000| Loss: 0.028117638542062964 | Accuracy: [0.99 Epoch 4000| Loss: 0.02121152512637285 | Accuracy: [0.99] Epoch 5000| Loss: 0.017185538643961436 | Accuracy: [0.99 Epoch 6000| Loss: 0.01451216394470429 | Accuracy: [1.] Epoch 7000| Loss: 0.012595068454554253 | Accuracy: [1.] Epoch 8000| Loss: 0.011149536124919996 | Accuracy: [1.] Epoch 9000| Loss: 0.01002043635274688 | Accuracy: [1.]	Epoch 0| Loss: 0.38761885645517985 | Accuracy: [0.47619048] Epoch 1000| Loss: 0.24453853947501428 | Accuracy: [0.4761904 Epoch 2000| Loss: 0.23731522663717813 | Accuracy: [0.6190476 Epoch 3000| Loss: 0.22406522772434118 | Accuracy: [0.8095238 Epoch 4000| Loss: 0.19796563460140057 | Accuracy: [0.9523809 Epoch 5000| Loss: 0.15577459361433882 | Accuracy: [0.9047619 Epoch 6000| Loss: 0.11553167866564379 | Accuracy: [1.] Epoch 7000| Loss: 0.08903674063093023 | Accuracy: [1.] Epoch 8000| Loss: 0.07223173494893431 | Accuracy: [1.] Epoch 9000| Loss: 0.06080965836982505 | Accuracy: [1.]

D. Anything you want to present

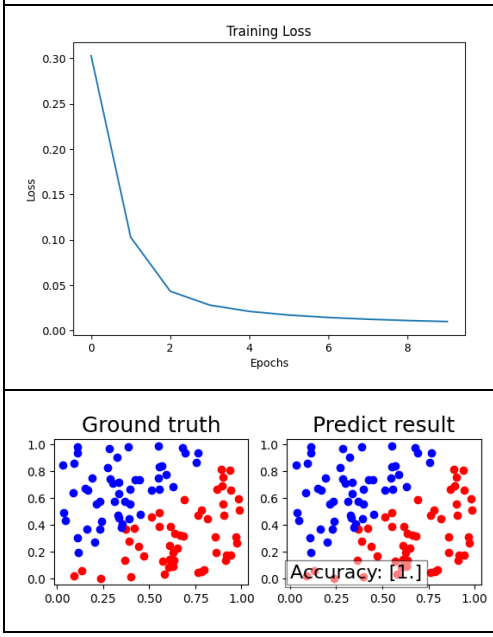
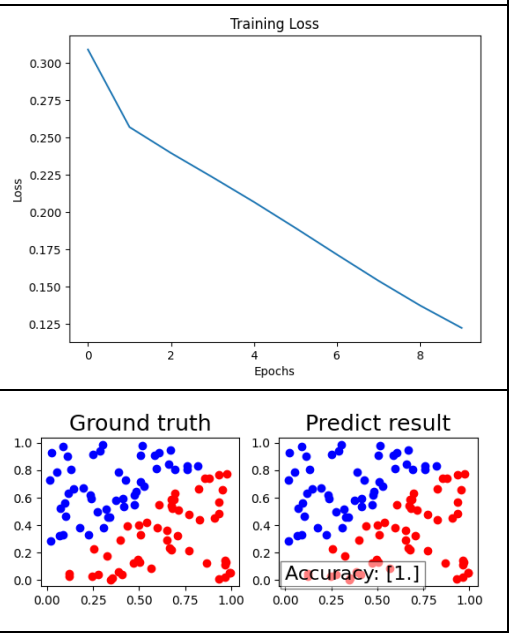
更具實驗結果可以看到在相同的設置下 Linear data 相較於 XOR data 更早收斂，同時 loss 也來的更低，表示出 Linear data 相較於 XOR data 更容易學習。

Discussion

A. Try different learning rates

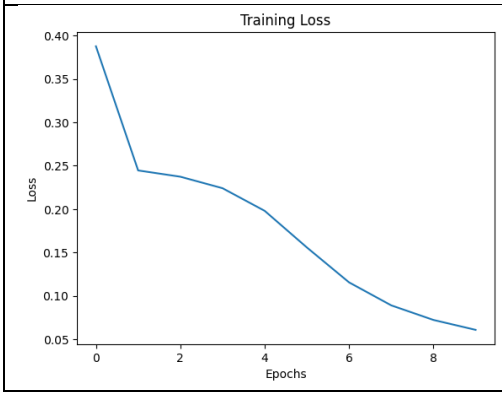
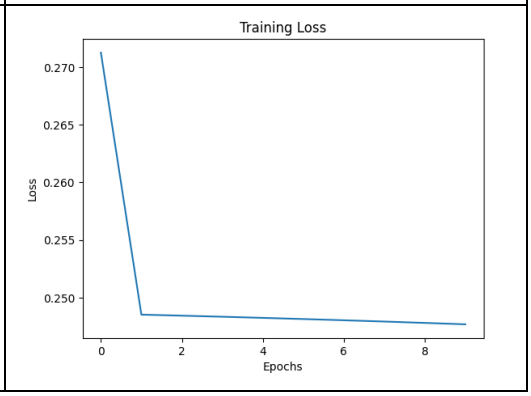
Linear data:

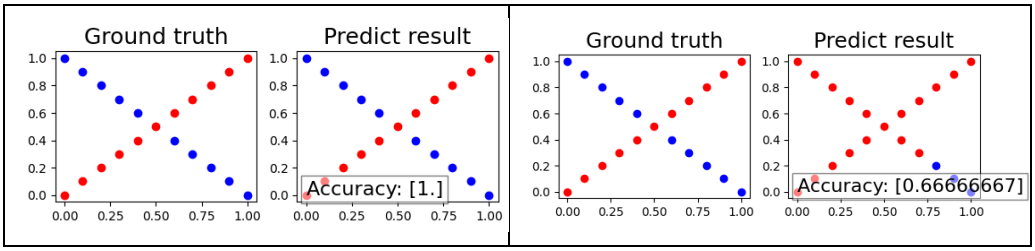
不同的學習率會造成不同的學習時間，在 0.01 時就需要更多的 epoch 來去訓練，在 accuracy 上也沒有造成其他的影響。

Lr=0.1	Lr=0.01
	

XOR data:

在 XOR data 的資料上可以發現雖然很快就收斂，但是可以看到準確度大幅的下降，所以可以發現在 gradient 下降的過程中尚未到達最低點，需要更長的訓練來達到最低點。

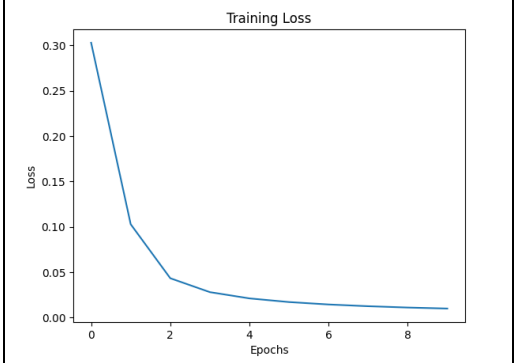
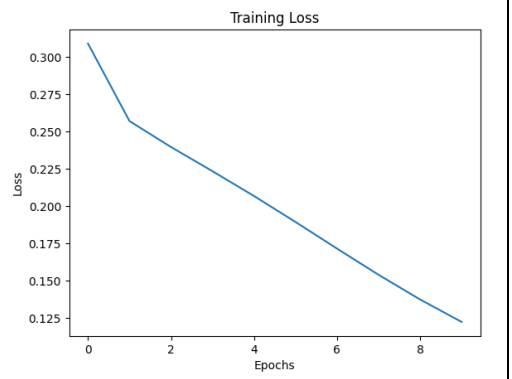
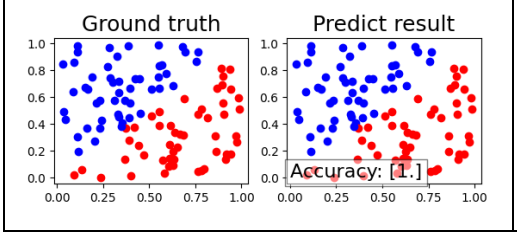
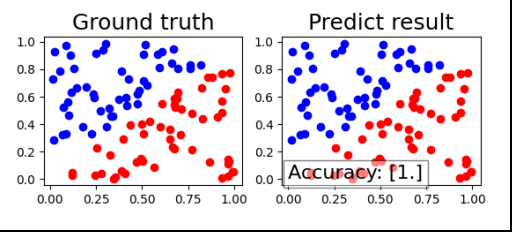
Lr=0.1	Lr=0.01
	



B. Try different numbers of hidden units

Linear data:

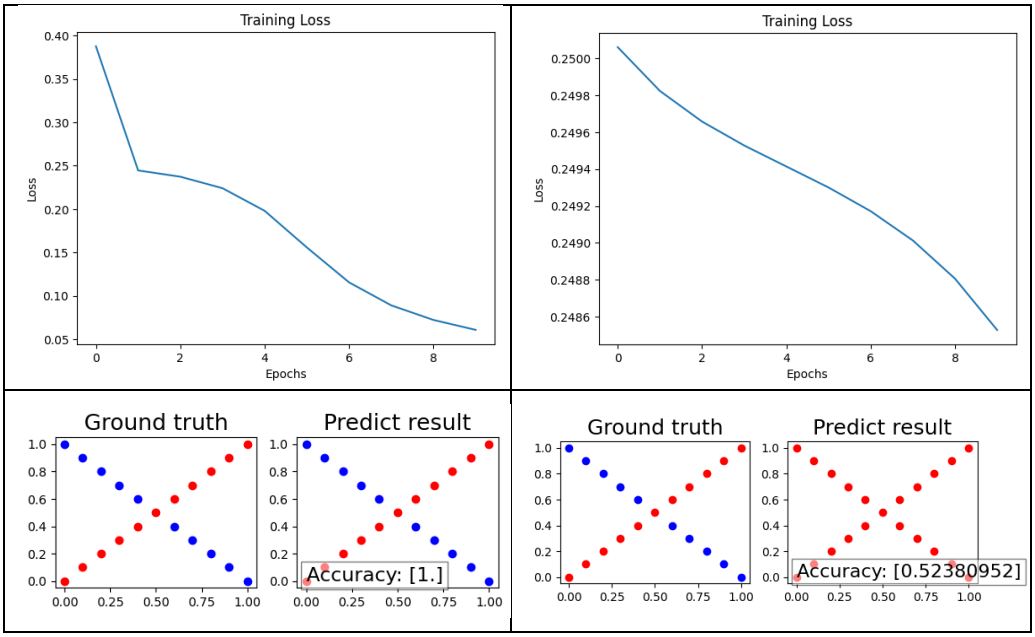
在使用不同的 hidden units 下看到 Accuracy 保持一樣，但是在 loss 上還是呈現下降的趨勢，未達到收斂。

Hidden Unit=10	Hidden Unit=2
	
	

XOR data:

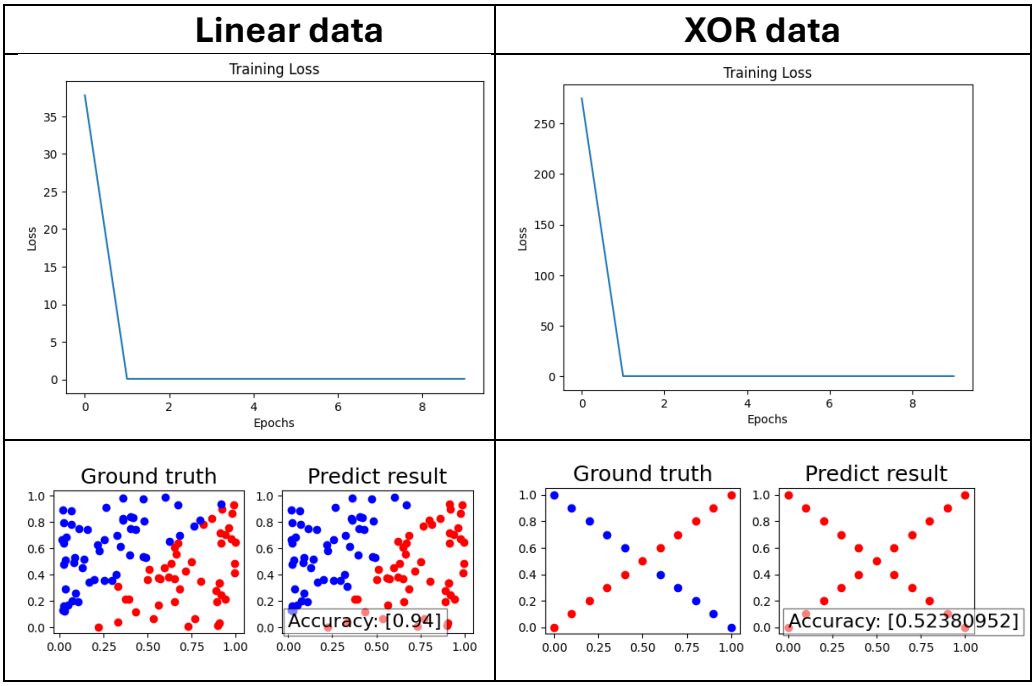
在 XOR data 的資料上與 Linear data 一樣可以看到 loss 還未收斂，同時在準確度上也大幅下降。

Hidden Unit =10	Hidden Unit =2



C. Try without activation functions

在不使用 activation functions 的情況下在 linear 還能達到一定的準確度，但是在 xor 這種非線性的資料下則是大約一半的準確略



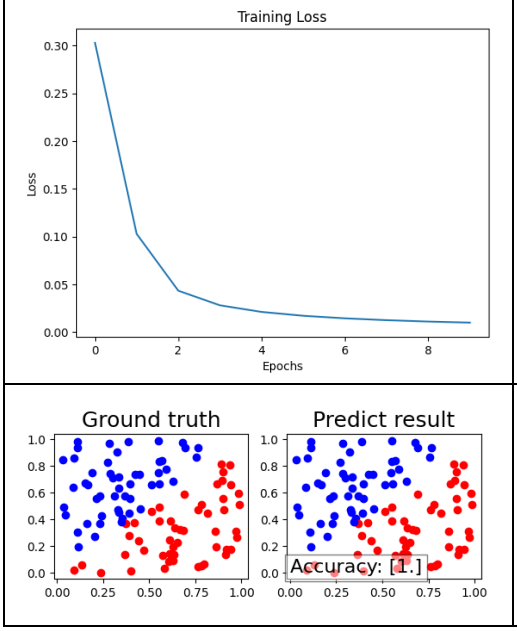
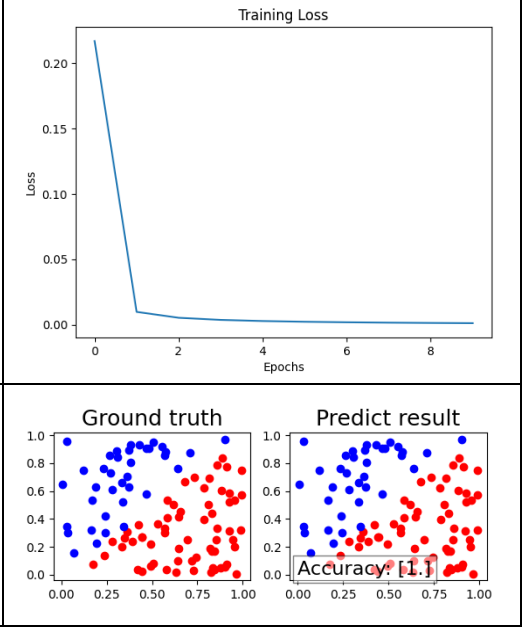
D. Anything you want to share

在嘗試 activation functions 時一開始按照原本設定的訓練會出現只有前幾個 epoch 有 loss，後面的 epoch 則是出現 nan，一開始會以為程式碼的是設計上出現問題，但後來發現是梯度爆炸導致消失，所以後來透過減少神經元和減小學習率就可以解決此問題。

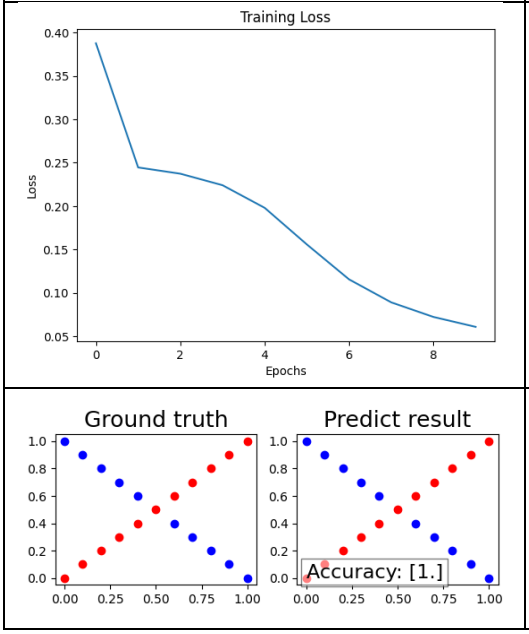
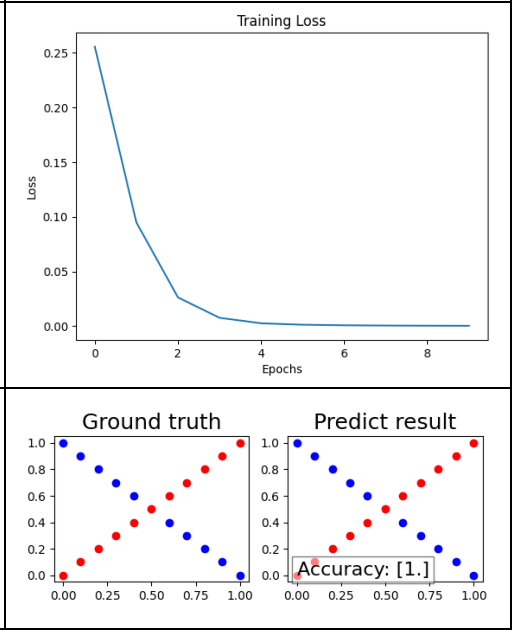
Extra

A. Implement different optimizers.

Linear data:

SGD	momentum
	

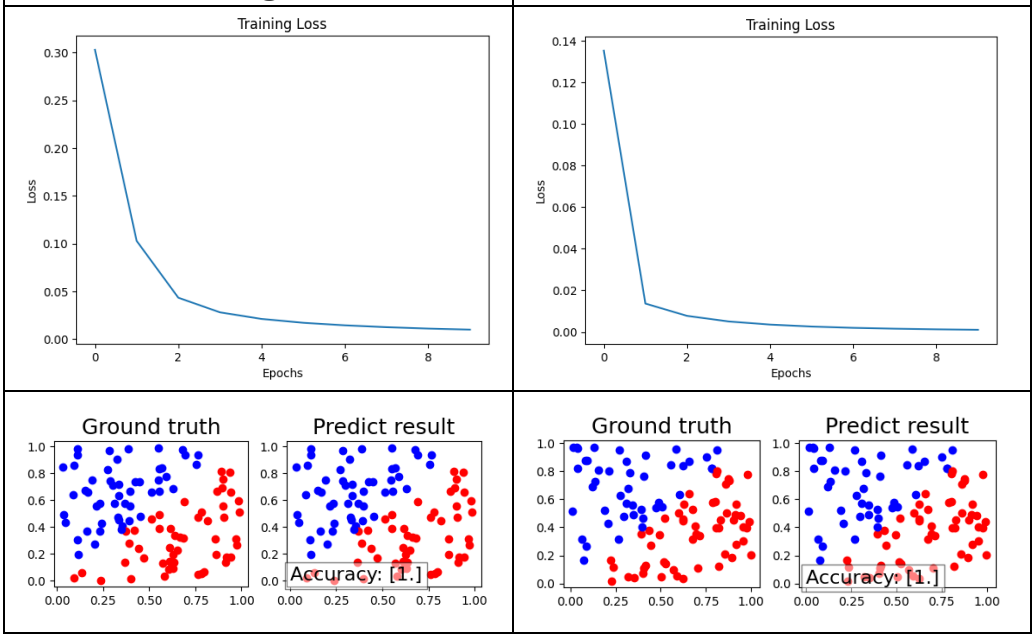
XOR data:

SGD	momentum
	

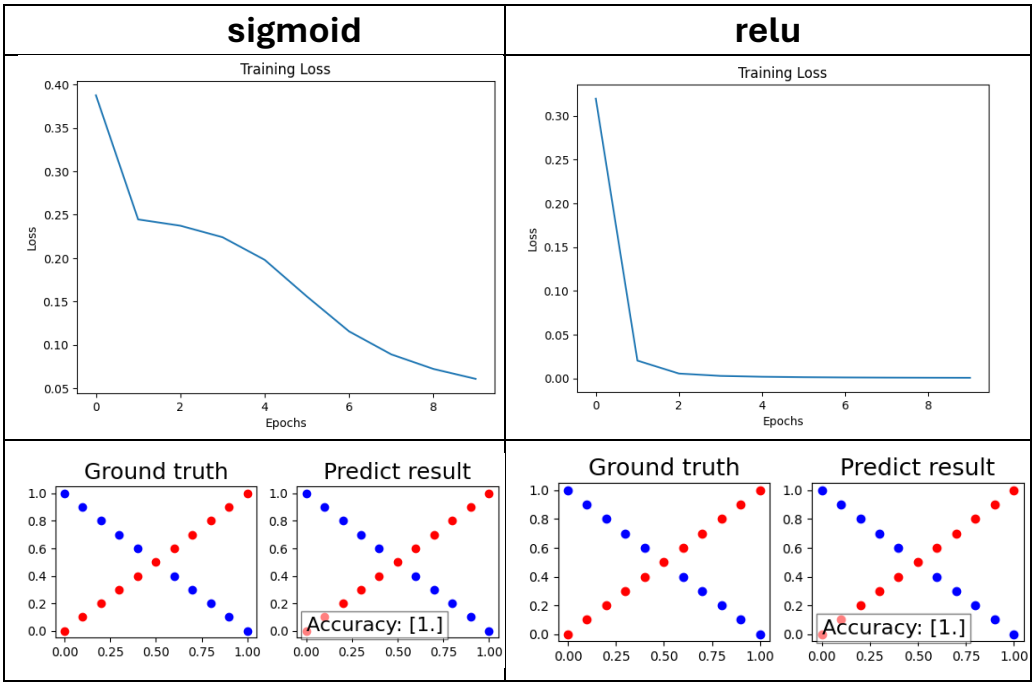
在準確度上並沒有差異，但是 loss curve 上可以發現 momentum 可以更快的收斂。

B. Implement different activation functions.

Linear data:

sigmoid	relu
	

XOR data:

sigmoid	relu
	

在使用不同的 activation functions 上原先全部的輸出都會接上 sigmoid，在這裡每一層都使用 relu 作為 activation function，只有輸出層一樣是接上 sigmoid，從結果來看可以發現 loss 更快的下降與收斂。

C. Implement convolutional layers.

Linear data:

```
input_size = 2
hidden_size1 = 10
hidden_size2 = 10
output_size = 1
epochs = 100
learning_rate = 0.1

model = NeuralNetwork()
# logging.debug("X shape: %s", x.shape)
model.add_layer(Conv_Layer(3, 1,1, activation_function="sigmoid"))
model.add_layer(FlattenLayer())
model.add_layer(Layer(input_size, output_size, activation_function="sigmoid"))
```

XOR data:

```
input_size = 2
hidden_size1 = 10
hidden_size2 = 10
output_size = 1
epochs = 500
learning_rate = 1

model = NeuralNetwork()
# logging.debug("X shape: %s", x.shape)
model.add_layer(Conv_Layer(3, 1,1, activation_function="sigmoid"))
model.add_layer(FlattenLayer())
model.add_layer(Layer(input_size, output_size, activation_function="sigmoid"))
```

將輸入去做 convolutional 做完之後加上 flatten，以便於接上全連接層，在架構使用都用一樣的，唯一差別在於 epoch 和 lr 的數值差異，由於用原始 linear 的參數時會導致，XOR 的準確率不到 50%，因此在這裡加上更多的 epoch 和增加 lr。

Linear	XOR