NYCU
Deep Learning Lab-1
Back Propagation

學號 ： 314551113

姓名 ： 劉哲良

Github : Lab1-BackPropagation

Content

Introduction

在本次作業中，實作了一個簡單的深度神經網路，並完成了 Forward Propagation 與 Back Propagation 的流程，以模擬模型的訓練過程。除此之外，也額外加入了多種 Activation Function 與 Optimizer，以觀察它們對模型訓練效果的影響與優化效果。

為了驗證模型的表現與泛化能力，訓練資料分別使用助教提供的 Linear Dataset 以及 XOR Dataset。本作業的主要目的是深入理解 Back Propagation 的運作原理與細節，並透過嘗試調整模型結構、參數以及 Hyperparameters（如 Learning rate、Activation Functions 與 Optimizer ）來優化訓練過程。

此外，透過實際實作與實驗，我們能更直觀地掌握深度學習中梯度下降的概念，並進一步了解不同設計選擇對於收斂速度與最終準確率的影響，為未來更複雜模型的開發打下基礎。

Implement Detail

Network Architecture

- Input Layer 含 2 個節點
- 兩層 Hidden Layer，可調整單元數
- Output Layer 含 1 個節點
- 全連接結構（fully connected）

```python
class Model:

    def __init__(
        self,
        input_size=2,
        output_size=1,
        hidden_layers_size=10,
        activation="sigmoid",
        optimizer="SGD",
    ):
        self.losses = []
        self.layers = []
        # Build model
        self.layers.append(
            Linear_Layer(input_size, hidden_layers_size, activation, optimizer)
        )
        self.layers.append(
            Linear_Layer(hidden_layers_size, hidden_layers_size, activation, optimizer)
        )
        self.layers.append(
            Linear_Layer(hidden_layers_size, output_size, activation, optimizer)
        )
```

Activation Functions

- 提供 Sigmoid、ReLU、Tanh 三種 Activation Funtions 選擇

- 同時實作各 Activation functions 的微分以利之後計算梯度使用

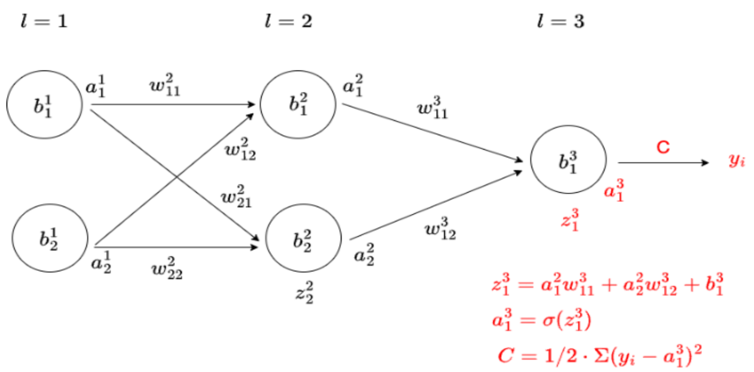
```python
# Activative functions and their derivatives
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))


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


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


def derivative_relu(x):
    return np.where(x >= 0, 1, 0)


def tanh(x):
    return np.tanh(x)


def derivative_tanh(x):
    return 1.0 - np.square(np.tanh(x))
```

Backpropagation

- Loss Function 採用 MSE（Mean Squared Error）
- 計算各層梯度
- 根據梯度更新 Weights 與 Bias

符號定義

- C：：Loss Function ，此模型中使用 Mean Square Error（MSE）
- σ ：Activation Function
- b^l ：第 l 層的 bias
- w^l ：第 l 層的 weight
- z^l ：從第 $l-1$ 層 到第 l 層經由 weight 和 bias 運算後得到的結果
- a^l ：σ（z^l）
- δ^l ：$\frac{\partial C}{\partial z^l}$



對於每一層 Layer，我們需要去計算其δ 值。假設我們已知δ^l，我們可以藉由此值推導出 δ^{l-1}，根據連鎖律(Chain Rule)，$\delta^{l-1} = \frac{\partial C}{\partial z^{l-1}} = \frac{\partial C}{\partial z^l} \times \frac{\partial z^l}{\partial a^{l-1}} \times \frac{\partial a^{l-1}}{\partial z^{l-1}}$。

$\frac{\partial C}{\partial z^l}$ 即為 δ^l，為我們已知

$\frac{\partial z^l}{\partial a^{l-1}}$根據觀察可得知為$w^l$

$\frac{\partial a^{l-1}}{\partial z^{l-1}}$即為 $\sigma'(z^{l-1})$，σ'為 Activation function 之微分

下方圖片中的*delta* 就是參照以上推導而計算出，知道δ 值後即可得知所有 weight 和 bias 的梯度，如下方證明，可知 bias 的梯度即為δ^l，weight 的梯度為$\delta^l \times a^l$。

$$\frac{\partial C}{\partial b^l} = \frac{\partial C}{\partial a^l} \times \frac{\partial a^l}{\partial z^l} \times \frac{\partial z^l}{\partial b^l} = \delta^l$$

$$\frac{\partial C}{\partial w^l} = \frac{\partial C}{\partial a^l} \times \frac{\partial a^l}{\partial z^l} \times \frac{\partial z^l}{\partial w^l} = \delta^l \times a^l$$

也就是下方程式碼 dW 和 db 的計算方式，該程式碼為 Linear_Layer 類別的 backward 實作

```python
def backward(self, upstream_delta, learning_rate=0.01):
    # check whether the activation function is empty
    if not self.activation:
        delta = upstream_delta
        # If no activation function, delta is just upstream_delta
    else:
        delta = upstream_delta * derivative_activation_map[self.activation](self.a)

    # Calculate gradients of weights and bias
    dW = np.dot(self.input.T, delta)
    db = np.sum(delta)

    # Update weights and bias
    if self.optimizer == "SGD":
        self.weights -= dW * learning_rate
        self.bias -= db * learning_rate
    elif self.optimizer == "Adagrad":
        # For the purpose of best training, learning rate should be adjusted according to the gradients
        # If gradients are small, learning rate should be larger, vice versa
        self.total_grad_w += np.square(dW)
        self.total_grad_b += np.square(db)
        self.weights -= (
            dW * learning_rate / np.sqrt(self.total_grad_w + self.epsilon)
        )
        self.bias -= db * learning_rate / np.sqrt(self.total_grad_b + self.epsilon)
    elif self.optimizer == "Momentum":
        self.v_weight = self.momentum * self.v_weight + dW * learning_rate
        self.v_bias = self.momentum * self.v_bias + db * learning_rate
        self.weights -= self.v_weight
        self.bias -= self.v_bias

    return np.dot(delta, self.weights.T)  # Return delta for the previous layer
```

並在 Model 中對每一層 Layer 呼叫 backward function 實現 Back propagation

```python
def backward(self, delta, learning_rate=0.01):
    # Backward pass
    for layer in reversed(self.layers):
        delta = layer.backward(delta, self.learning_rate)
```

Extra Implementation

- 支援多種 Optimizer，包括 SGD、Momentum、AdaGrad

  Optimizer 的實作可參考上方圖片

  - **SGD**：直接使用 Learning Rate 乘上梯度來更新 Weights

  - **Momentum**：透過累積梯度的速度向量（v_weight, v_bias），幫助模型跳脫 Local Mimimum 並讓收斂路徑更平滑。

    $$V_t \leftarrow \beta V_{t-1} - \eta \frac{\partial L}{\partial W}$$

    $$W \leftarrow W + V_t$$

  - **Adagrad**：根據以往累積的梯度平方和來動態調整 Learning Rate。當某些參數需要較大或較小的更新時特別有幫助。

    $$W \leftarrow W - \eta \frac{1}{\sqrt{n+\epsilon}} \frac{\partial L}{\partial W}$$

    $$n = \sum_{r=1}^{t} \left(\frac{\partial L_r}{\partial W_r}\right)^2$$

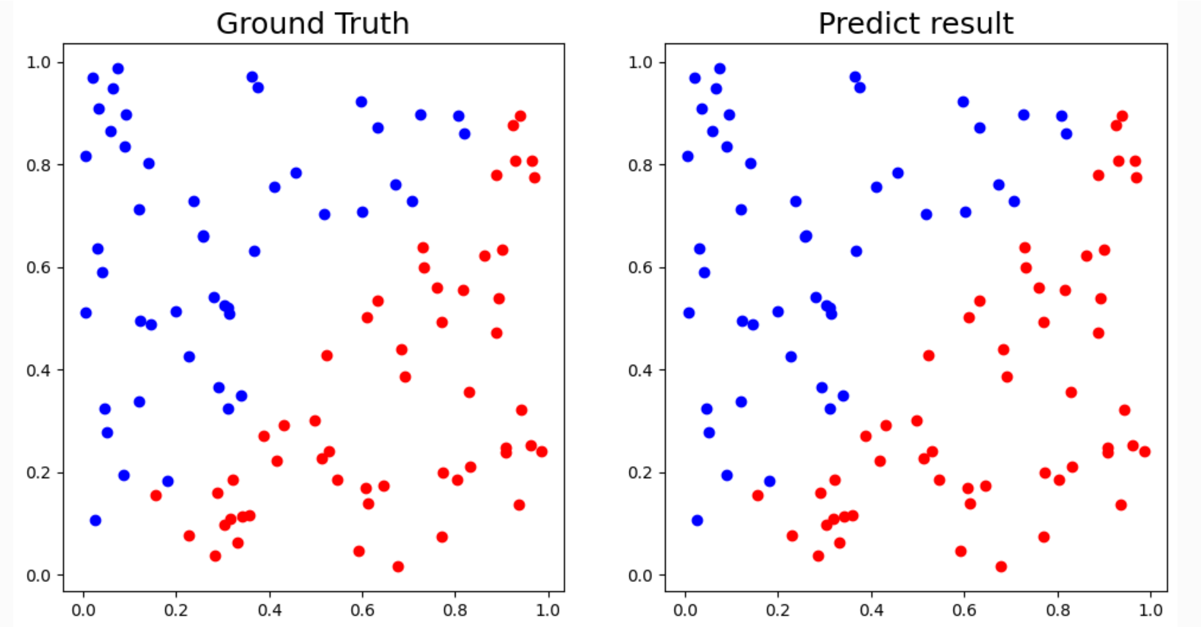
    $$W \leftarrow W - \eta \frac{1}{\sqrt{\sum_{r=1}^{t} \left(\frac{\partial L_r}{\partial W_r}\right)^2 + \epsilon}} \frac{\partial L}{\partial W}$$

- 可調整參數如 Learning Rate、Hidden Layer 點數、Activation Functions 類型等

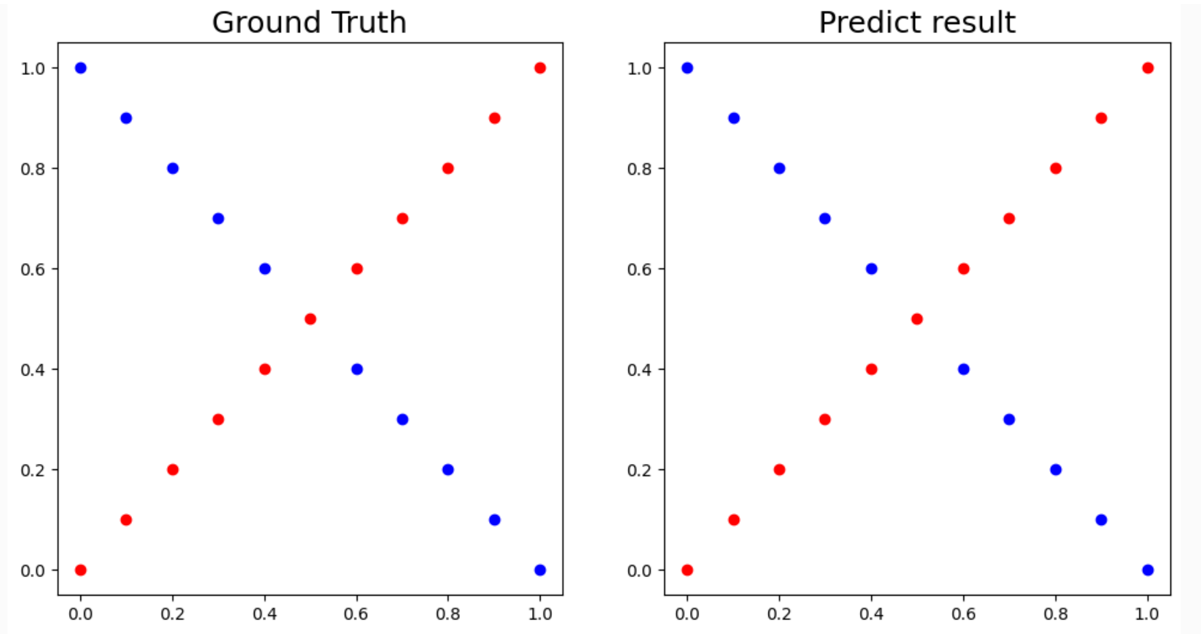
Experimental Results

Screenshot and comparison figure

Linear



XOR



Show the accuracy of your prediction

Hyperparameters

Activation Function	Optimizer	Learning Rate	Hidden_Units
tanh	SGD	0.01	10

Linear Dataset

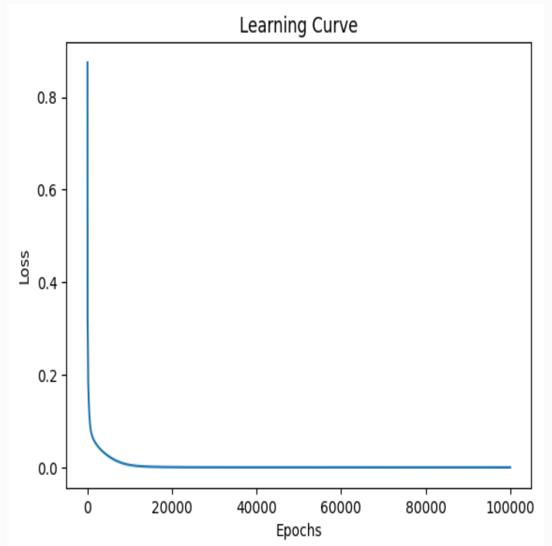
```
Iter88 |    Ground Truth: [0] |    Predict: [0.02985598] |
Iter89 |    Ground Truth: [0] |    Predict: [-0.00030772] |
Iter90 |    Ground Truth: [0] |    Predict: [-0.10596692] |
Iter91 |    Ground Truth: [0] |    Predict: [0.01921551] |
Iter92 |    Ground Truth: [0] |    Predict: [0.15833155] |
Iter93 |    Ground Truth: [1] |    Predict: [0.99975412] |
Iter94 |    Ground Truth: [0] |    Predict: [-0.00655765] |
Iter95 |    Ground Truth: [0] |    Predict: [0.00890729] |
Iter96 |    Ground Truth: [1] |    Predict: [0.99854244] |
Iter97 |    Ground Truth: [0] |    Predict: [0.00630843] |
Iter98 |    Ground Truth: [1] |    Predict: [0.91700917] |
Iter99 |    Ground Truth: [1] |    Predict: [0.99966345] |
Iter100 |    Ground Truth: [0] |    Predict: [-0.06874714] |
Loss = 0.0060 Accuracy = 100.00%
```

XOR Dataset

```
Iter13 |    Ground Truth: [1] |    Predict: [0.99260072] |
Iter14 |    Ground Truth: [0] |    Predict: [0.0029742] |
Iter15 |    Ground Truth: [1] |    Predict: [0.99988569] |
Iter16 |    Ground Truth: [0] |    Predict: [-0.00441735] |
Iter17 |    Ground Truth: [1] |    Predict: [0.99993517] |
Iter18 |    Ground Truth: [0] |    Predict: [0.00217426] |
Iter19 |    Ground Truth: [1] |    Predict: [0.99992022] |
Iter20 |    Ground Truth: [0] |    Predict: [0.00027366] |
Iter21 |    Ground Truth: [1] |    Predict: [0.99988022] |
Loss = 0.0000 Accuracy = 100.00%
```

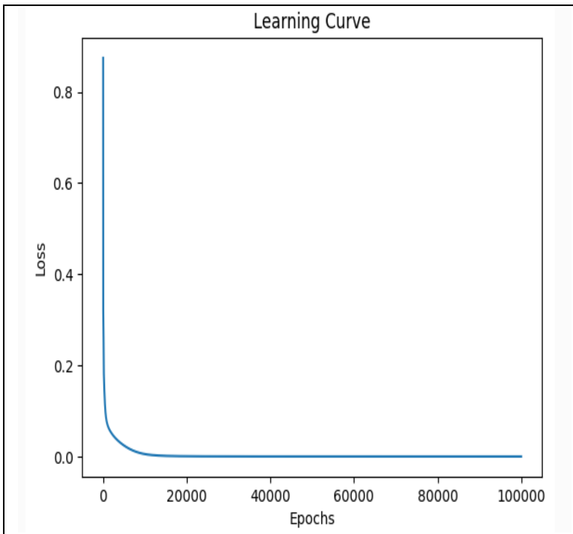
Learning curve (loss-epoch curve)

Linear



Epoch 1000, Loss: 0.0710
Epoch 2000, Loss: 0.0512
Epoch 3000, Loss: 0.0397
Epoch 4000, Loss: 0.0311
Epoch 5000, Loss: 0.0241
Epoch 6000, Loss: 0.0183
Epoch 7000, Loss: 0.0136
Epoch 8000, Loss: 0.0100
Epoch 9000, Loss: 0.0072
Epoch 10000, Loss: 0.0052
Epoch 11000, Loss: 0.0038
Epoch 12000, Loss: 0.0029
Epoch 13000, Loss: 0.0022
Epoch 14000, Loss: 0.0017
Epoch 15000, Loss: 0.0014
Epoch 16000, Loss: 0.0011
Epoch 17000, Loss: 0.0009
Epoch 18000, Loss: 0.0007

XOR



Epoch 1000, Loss: 0.0710
Epoch 2000, Loss: 0.0512
Epoch 3000, Loss: 0.0397
Epoch 4000, Loss: 0.0311
Epoch 5000, Loss: 0.0241
Epoch 6000, Loss: 0.0183
Epoch 7000, Loss: 0.0136
Epoch 8000, Loss: 0.0100
Epoch 9000, Loss: 0.0072
Epoch 10000, Loss: 0.0052
Epoch 11000, Loss: 0.0038
Epoch 12000, Loss: 0.0029
Epoch 13000, Loss: 0.0022
Epoch 14000, Loss: 0.0017
Epoch 15000, Loss: 0.0014
Epoch 16000, Loss: 0.0011

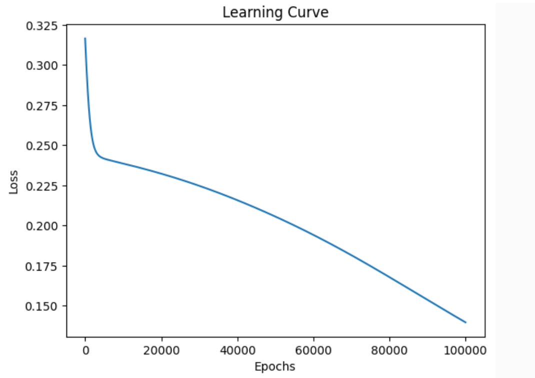
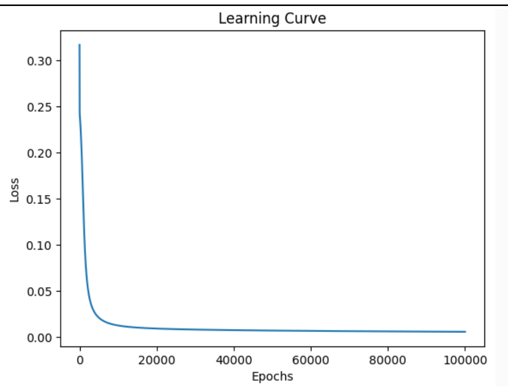
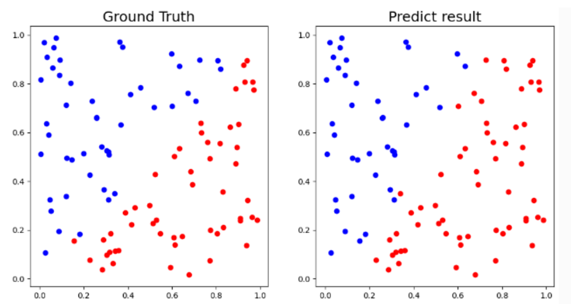
Discussions

Try different learning rates

Fixed Hyperparameters

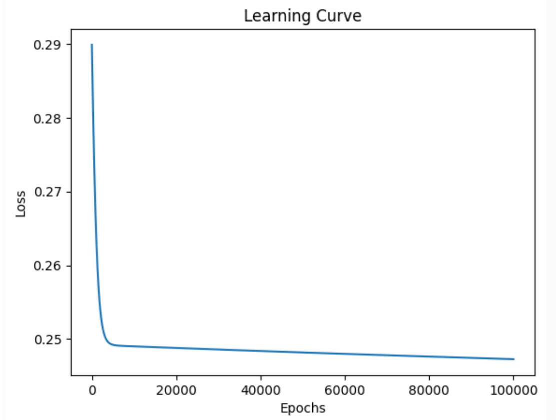
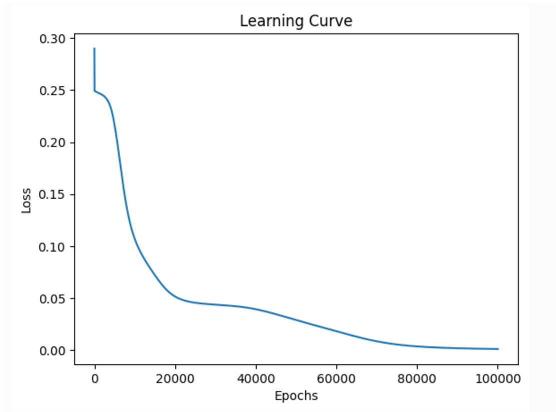
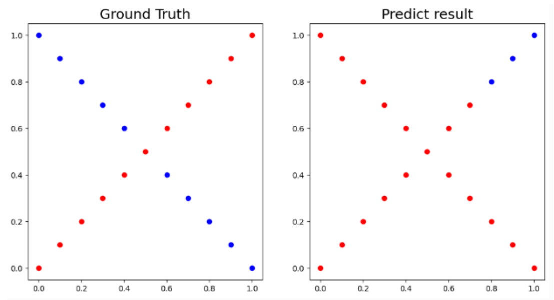
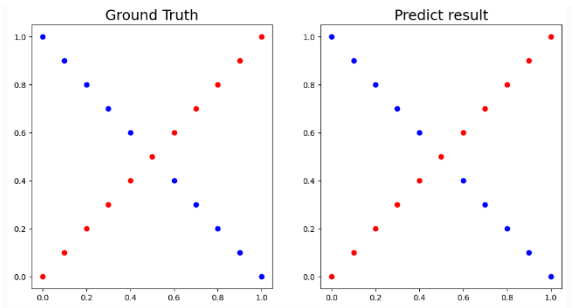
Activation Function	Optimizer	Hidden_Units
sigmoid	SGD	10

Linear

Learning Rate 0.001	Learning Rate 0.1
	
	

可以觀察到，當 learning_rate 較小時，梯度更新速度較為緩滿，也導致準確率有些微下降的情況。learning_rate 較大時，梯度更新速度會快上許多，準確率仍保持 100%，在 Linear Dataset 中，learning rate 對準確率的影響並不大。

XOR

Learning Rate 0.001	Learning Rate 0.1
	
	

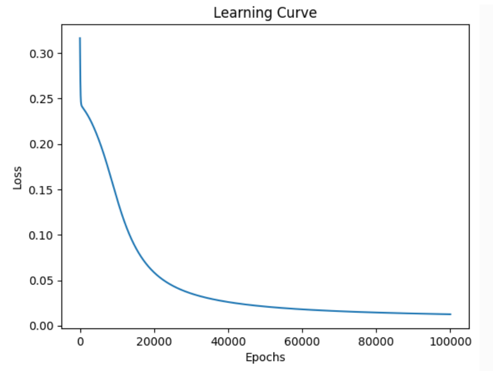
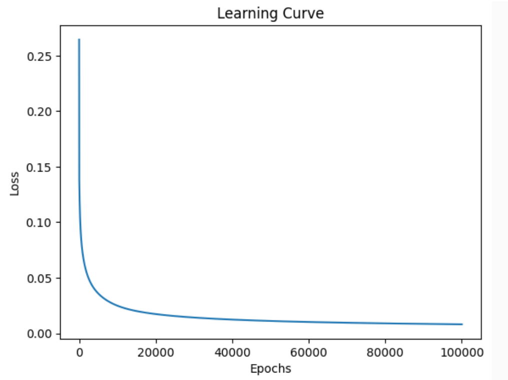
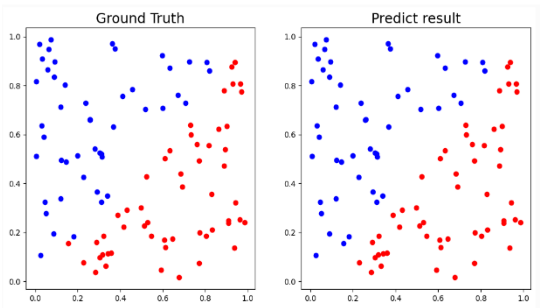
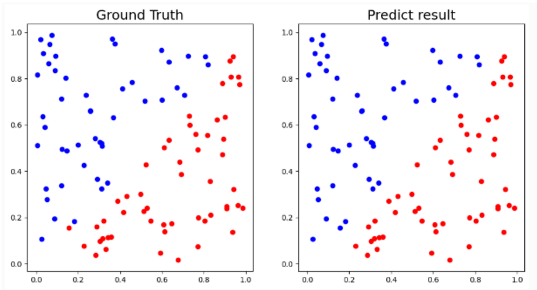
在 XOR Dataset 中，learning rate 對於訓練的影響就相比 Linear 較為明顯，首先，同樣地，learning rate 對於 loss 的影響相同，learning rate 大，loss 下降速度較快而且較為陡峭，learning rate 小，loss 下降速度慢且平緩。顯著的是，在此 Dataset 中，小的 learning rate 只能達到準確率約 38%，learning rate 數值大時卻能達到 100%。

Try different numbers of hidden units

Fixed Hyperparameters

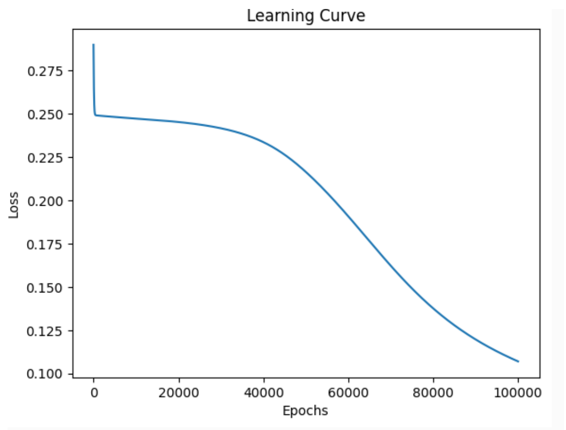
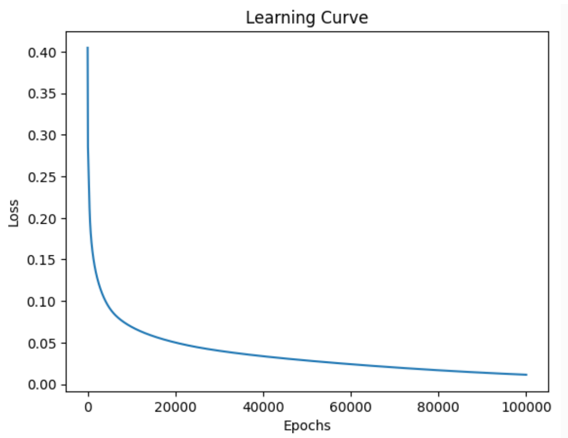
Activation Function	Optimizer	Learning Rate
sigmoid	SGD	0.01

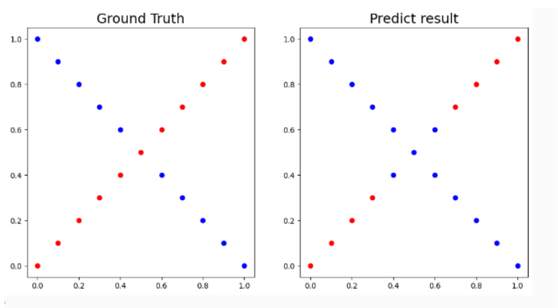
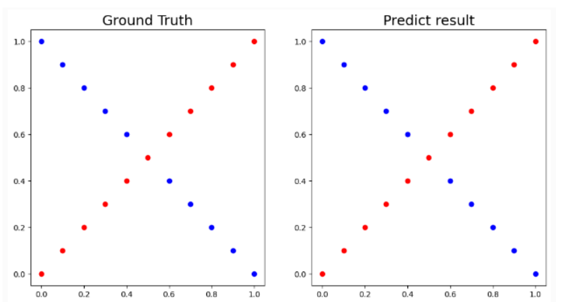
Linear

<table>
<tr><th>Hidden Layer 10</th><th>Hidden Layer 100</th></tr>
<tr><td>
</td><td>
</td></tr>
<tr><td>
</td><td>
</td></tr>
</table>

在 Linear Dataset 中，訓練結果沒有明顯的差異，但是可以明顯感覺到 hidden_layer 數增加時，訓練模型時間也明顯上升。再者，從 learning_curve 可以看出，當層數越多時，模型的 loss 下降也比較穩定。

XOR

<table>
<tr><th>Hidden Layer 10</th><th>Hidden Layer 100</th></tr>
<tr><td>
</td><td>
</td></tr>
</table>

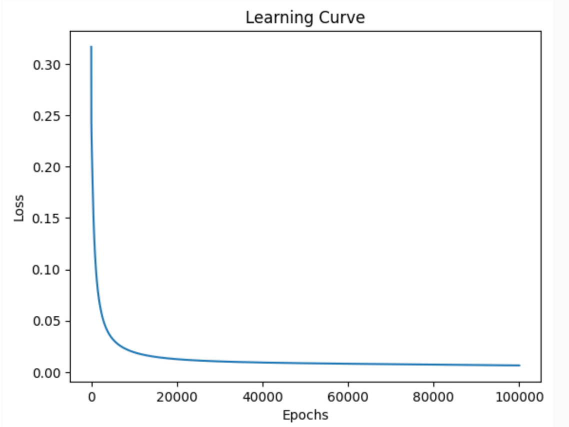
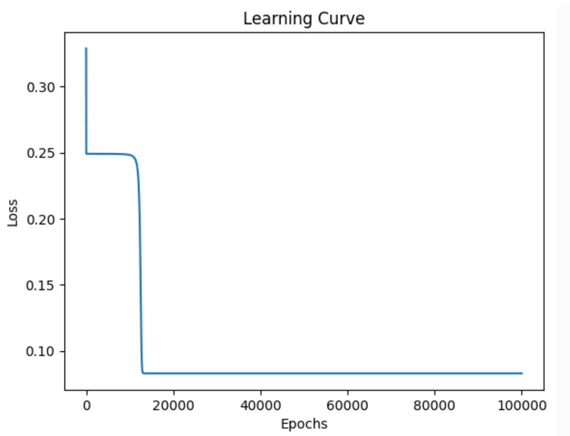
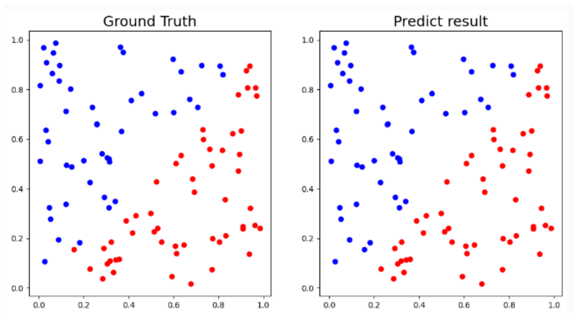
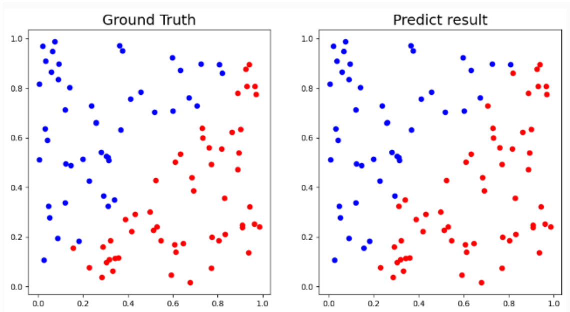
相較於 Linear Dataset，提高 hidden layer 層數對於 XOR Dataset 有更好的訓練效果，能觀察到，層數較多，loss 能明顯下降，準確率也明顯提升。

Try without activation functions

Fixed Hyperparameters

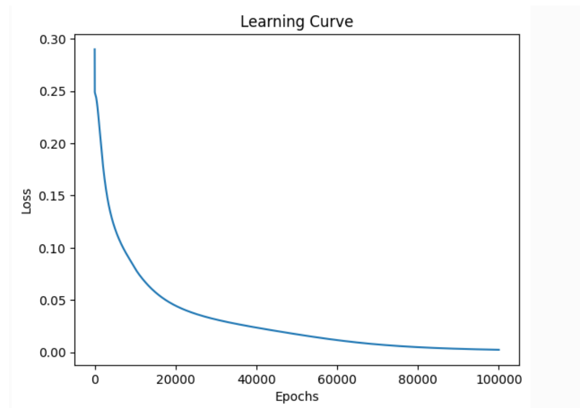
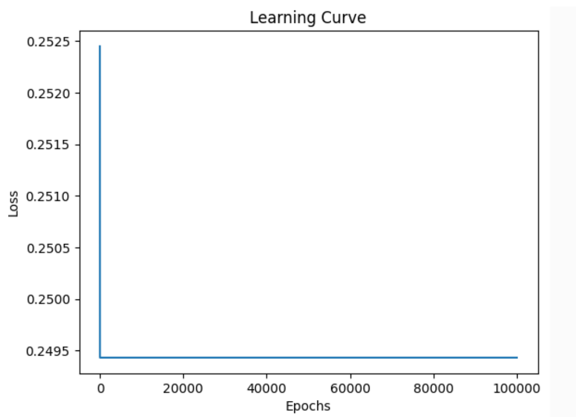
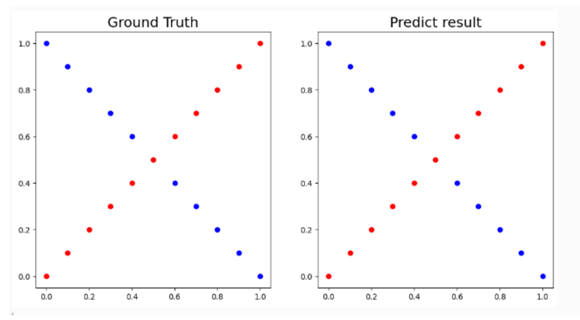
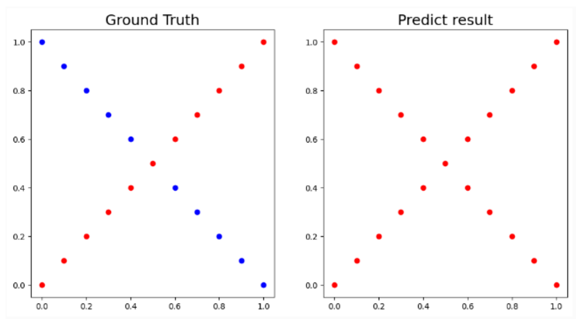
Hidden Layer	Optimizer	Learning Rate
10	Adagrad	0.01

Linear

With Activation Function (Sigmoid)	Without Activation Function
	
	

沒有 Activation Function 在 Linear Dataset 影響不大，但可以看出影響了 loss 下降變得不穩定，相較於有 Activation Function，沒有使用 Activation Function 的準確率也下降了些許。

XOR

With Activation Function (Sigmoid)	Without Activation Function
	
	

相較於 Linear Dataset，有無 Activation Function 對於 XOR Dataset 的影響非常大，沒有 Activation Function 基本上 Train 不了 model。

Extra Implementation Discussions

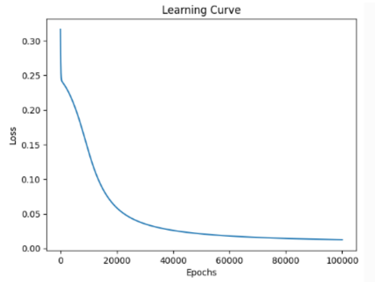
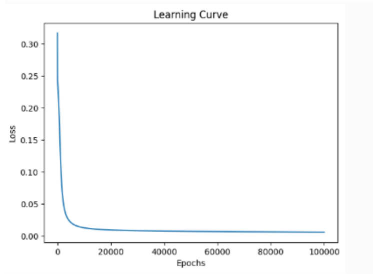
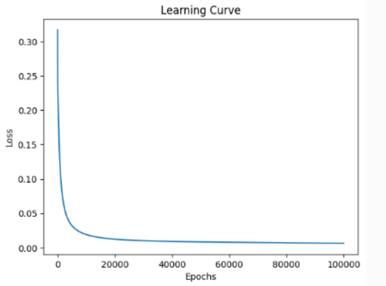
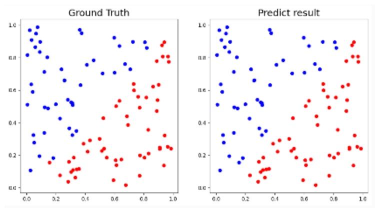
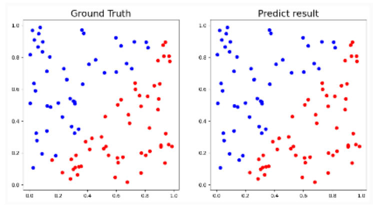
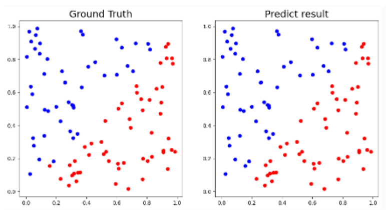
Try different optimizers

除了一般的 SGD，在本次作業中我額外實作了 Adagrad 以及 Momentum，下面會比較使用其他 optimizer 對於訓練的影響

Fixed Hyperparameters

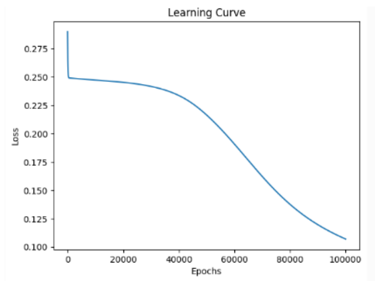
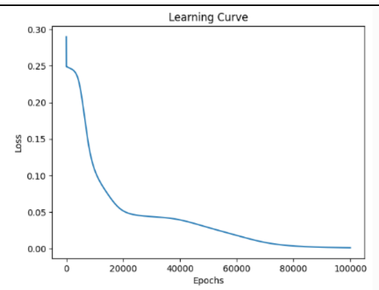
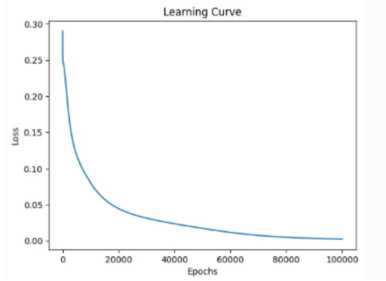
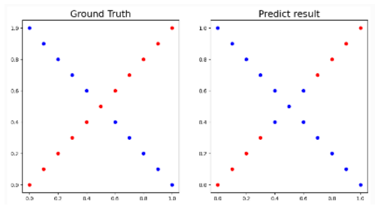
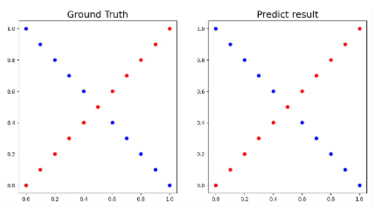
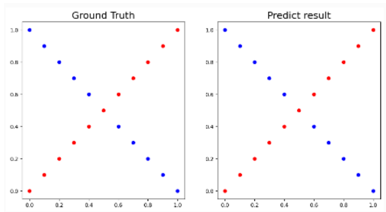
Hidden Layer	Activation Function	Learning Rate
10	Sigmoid	0.01

Linear

SGD	Momemtum	Adagrad
		
		

Optimizer 對於 Linear Dataset 的準確率沒有太大的影響，但可以看出與其使用單純的 SGD，其他兩者的 Loss 下降速度較快。

XOR

SGD	Momemtum	Adagrad
		
		

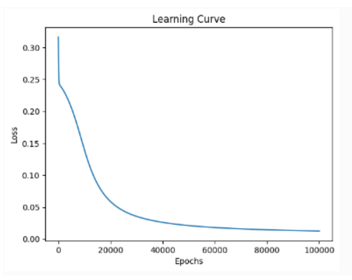
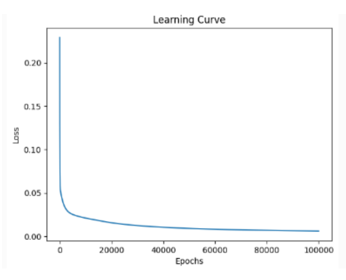
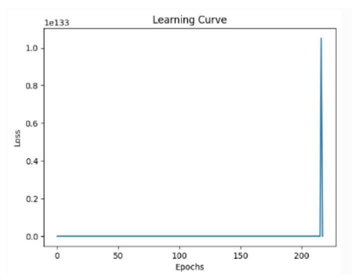
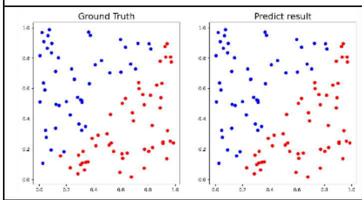
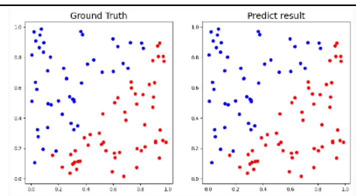
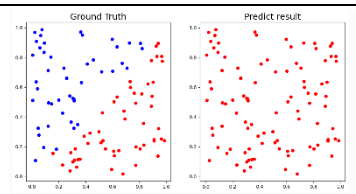
XOR Dataset 在使用 SGD 的情況下無法訓練出一個好的 Model，其他兩個 Optimizer 反而能讓 Model 準確率達到 100%

Try different activation functions

Fixed Hyperparameters

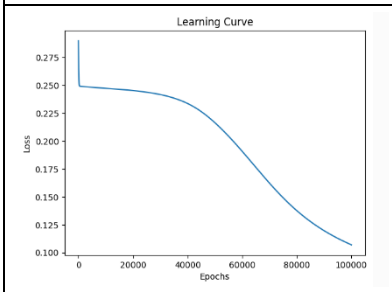
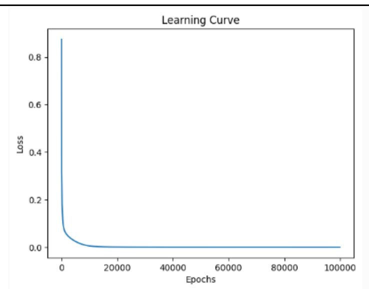
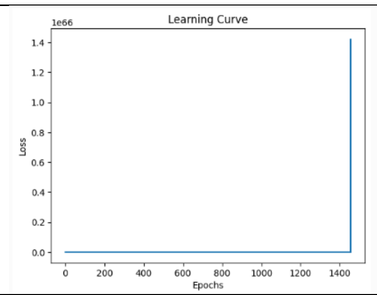
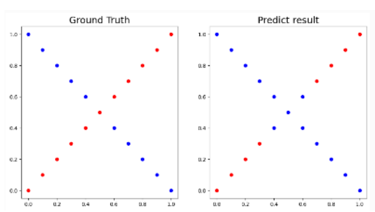
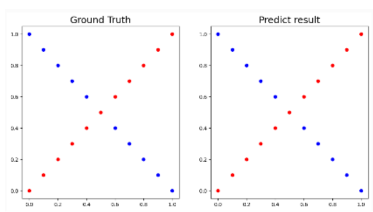
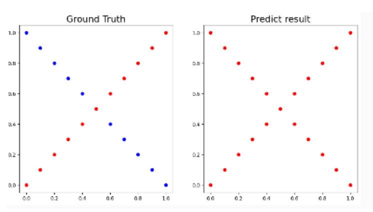
Hidden Layer	Optimizer	Learning Rate
10	SGD	0.01

Linear

Sigmoid	Tanh	Relu
		
		

可以觀察到使用 tanh 的效果最好，使用 relu 反而無法訓練 Model，Sigmoid 能訓練出不錯的 Model，但相較於 tanh 略遜了一點。

XOR

Sigmoid	Tanh	Relu
		
		

基本上與 Linear Dataset 的結果相同

Questions

A. What are the purposes of activation functions?

使得訓練出來的模型不會只是線性函數，使用可微分的 activation function，可以經由多層神經元的疊加產生非線性效果，使神經網路能夠逼近複雜的實際數據分布，解決更困難的分類或回歸問題。

B. What if the learning rate is too large or too small?

當 learning rate 太小時，參數更新幅度太小，導致訓練速度非常緩慢，可能需要非常多次迭代才能收斂。

當 learning rate 太大時，參數更新幅度過大，可能導致 loss 函數震盪甚至發散，無法收斂到最小值。

C. What are the purposes of weights and biases in a neural network?

Weight 決定神經元之間的連結的強度並控制輸入對於輸出的影響程度，Bias 能夠移動 Activation Functions，增加模型的靈活性，兩者都能調整模型函數的形狀以 Fitting 實際數據分布。

Reference

[1] https://datasciocean.tech/deep-learning-core-concept/backpropagation-explain/

[2]
https://medium.com/%E9%9B%9E%E9%9B%9E%E8%88%87%E5%85%94%E5%85%94%E7%9A%84%E5%B7%A5%E7%A8%8B%E4%B8%96%E7%95%8C/%E6%A9%9F%E5%99%A8%E5%AD%B8%E7%BF%92ml-note-sgd-momentum-adagrad-adam-optimizer-f20568c968db