NYCU DL Lab1 – Backpropagation

312551068 張紀睿

1. Introduction

在本次作業中實作了兩層 hidden layer 的 Neural Network，並透過程式實現 forward, back propagation 以及不同的 activation function 與 optimizer，利用 Linear Data 以及 XOR Data 對神經網路進行訓練，並嘗試不同訓練設定，以了解各項參數對於模型的影響。

2. Experiment setups

A. Sigmoid functions

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

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

B. Neural network

```
class Model:
    def __init__(self, input_size= 2, hidden_size= 10, output_size= 1, lr=0.01,
                 optim = "SGD", activate = "ReLU", show_epoch = 10000, decay = 0.9):
        self.layer1 = Layer(input_size, hidden_size, activate)
        self.layer2 = Layer(hidden_size, hidden_size, activate)
        if activate == "None":
            self.output = Layer(hidden_size,output_size, activate)
        else:
            self.output = Layer(hidden_size,output_size, "Sigmoid")
        self.lr = lr
        self.loss = []
        self.show_epoch = show_epoch
        self.epoch = 0
        self.optim = optim
        self.decay = decay

    def train(self,x,y,epoch = 100000):
        self.epoch += epoch
        for i in range(epoch):
            output = self.output.forward(self.layer2.forward(self.layer1.forward(x)))
            loss, grad = MSELoss(output, y)
            self.layer1.backward(self.layer2.backward(self.output.backward(grad, self.lr, self.optim, self.decay)
                                 , self.lr, self.optim, self.decay), self.lr, self.optim, self.decay)
            self.loss.append(loss)
            if i%self.show_epoch == 0:
                print(f"epoch {i} loss : {loss}")
        self.prediction = output
        plt.subplot(2,1,1)
        plt.title("Learning Curve", fontsize = 18)
        plt.xlabel("Epoch")
        plt.ylabel("Loss")
        plt.plot(loss)
        return output

    def show_result(self,x,y):
        import matplotlib.pyplot as plt
        plt.plot(self.loss)
        print(f"Accuracy : {sum((self.prediction > 0.5)== (y==1))/y.size}")
        print("Prediction : ")
        for i in range(y.size):
            print(f"Iter{i+1} |    Ground truth: {y[i]} |     prediction: {self.prediction[i]} |")
        show_result(x,y,self.prediction>0.5)
```

建立模型時可傳入要設定的參數、Activate Function、Optimizer 等等，再由 class Layer 建立 hidden layer(詳細設定將於後續介紹)。使用 model.train Function 即可進行訓練，並於訓練時顯示 loss。訓練完成後，透過 model.show_result 來顯示 testing 結果。

C. Back propagation

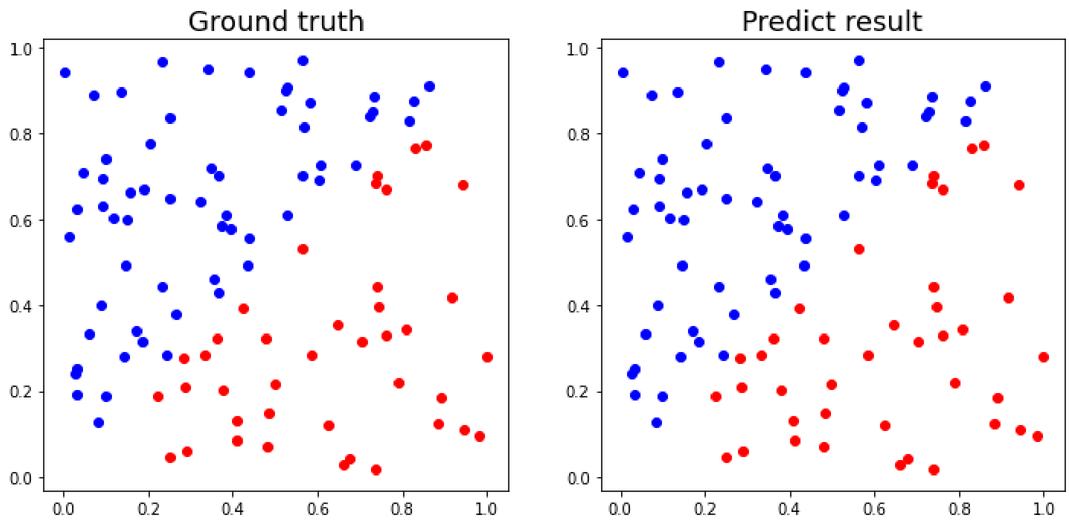
```python
class Layer:
    def __init__(self, input_size, output_size, activate = "Sigmoid"):
        self.input_size = input_size
        self.output_size = output_size
        self.activate = activate
        self.v_w = 0
        self.v_b = 0
        self.total_w = 0
        self.total_b = 0
        #initialize weight and bias
        self.w = np.random.randn(input_size, output_size) #shape(2,n)
        self.b = np.random.randn(1, output_size)/100 #shape(1,n)
    def foward(self,x):
        self.x = x
        z = np.dot(x,self.w) +self.b
        if self.activate == "Sigmoid":
            z = sigmoid(z)
        elif self.activate == "ReLU":
            z = ReLU(z)
        elif self.activate == "tanh":
            z = tanh(z)
        else:
            pass
        self.z = z

        return z
    def backward(self, upstream_grad, lr=0.01, optim = "momentum", decay = 0.9):
        if self.activate == "Sigmoid":
            grad = upstream_grad * derivative_sigmoid(self.z)
        elif self.activate == "ReLU":
            grad = upstream_grad * derivative_ReLU(self.z)
        elif self.activate == "tanh":
            grad = upstream_grad * derivative_tanh(self.z)
        else:
            grad = upstream_grad

        if optim == "SGD":
            self.b -= np.sum(grad) * lr
            self.w -= np.dot(self.x.T, grad) *lr
        elif optim == "momentum":
            self.v_w = decay * self.v_w + lr * np.dot(self.x.T, grad)
            self.v_b = decay * self.v_b + lr * np.sum(grad)
            self.b -= self.v_b
            self.w -= self.v_w
        elif optim == "AdaGrad":
            self.total_w += np.dot(self.x.T, grad) ** 2
            self.total_b += np.sum(grad) ** 2
            self.b -= np.sum(grad) * lr / (np.sqrt(self.total_b) + 1e-8)
            self.w -= np.dot(self.x.T, grad) * lr / (np.sqrt(self.total_w) + 1e-8)

        return np.dot(grad,self.w.T)
```

Hidden layer 經過 forward 取得輸出並由 loss function 算出 upstream gradient 後，透過 class layer 的 backward function 進行 back propagation，根據所選的 activation function 計算 gradient 再一步步回傳；更新權重時也會依所選 optimizer 的不同而使用不同的計算方式。

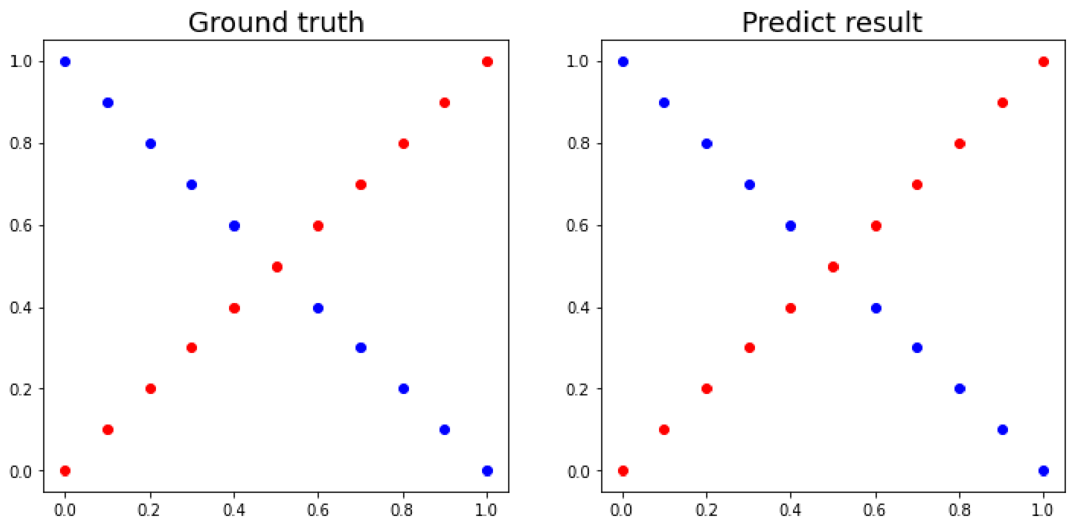
3. Result of your testing

A. Screenshot and comparison figure

Linear:



XOR:



B. Show the accuracy of your prediction

Lr = 0.1, Hidden unit = 10, Optimizer = SGD, Activate function: sigmoid

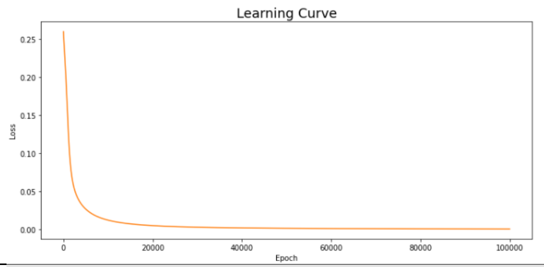
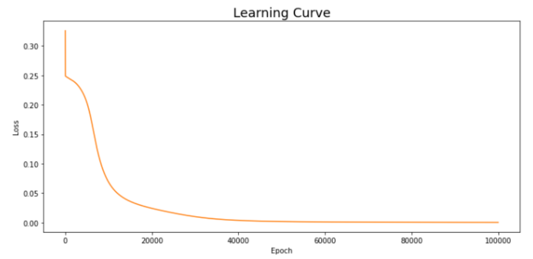
Linear:

```
Accuracy : [1.]
Prediction :
Iter1 |    Ground truth: [0] |    prediction: [2.58807061e-06] |
Iter2 |    Ground truth: [0] |    prediction: [1.85841029e-07] |
Iter3 |    Ground truth: [1] |    prediction: [0.99999997] |
Iter4 |    Ground truth: [1] |    prediction: [0.99999997] |
Iter5 |    Ground truth: [0] |    prediction: [2.68923182e-07] |
Iter6 |    Ground truth: [0] |    prediction: [0.00430186] |
Iter7 |    Ground truth: [1] |    prediction: [0.99993948] |
Iter8 |    Ground truth: [0] |    prediction: [3.32817058e-07] |
Iter9 |    Ground truth: [0] |    prediction: [3.98129433e-07] |
Iter10 |    Ground truth: [0] |    prediction: [5.48180533e-07] |
```

XOR:

```
Accuracy : [1.]
Prediction :
Iter1 |     Ground truth: [0] |     prediction: [0.00698727] |
Iter2 |     Ground truth: [1] |     prediction: [0.99995585] |
Iter3 |     Ground truth: [0] |     prediction: [0.00731864] |
Iter4 |     Ground truth: [1] |     prediction: [0.99996129] |
Iter5 |     Ground truth: [0] |     prediction: [0.00774476] |
Iter6 |     Ground truth: [1] |     prediction: [0.99995884] |
Iter7 |     Ground truth: [0] |     prediction: [0.00910553] |
Iter8 |     Ground truth: [1] |     prediction: [0.99987822] |
Iter9 |     Ground truth: [0] |     prediction: [0.01832702] |
Iter10 |     Ground truth: [1] |     prediction: [0.95972369] |
```

C. Learning curve (loss, epoch curve)

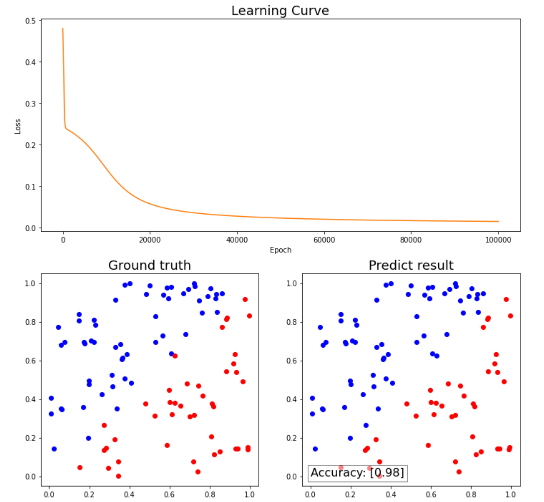
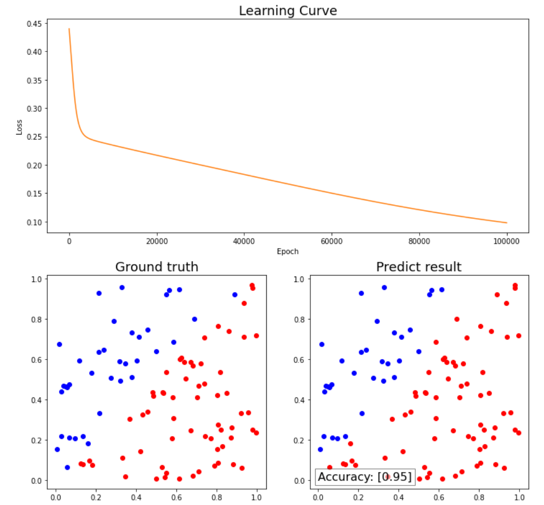
Linear	XOR
	
epoch 0 loss : 0.2597932327475446 epoch 10000 loss : 0.012081257957771586 epoch 20000 loss : 0.004912970646488129 5 epoch 30000 loss : 0.0028297077871038334 epoch 40000 loss : 0.0018988833990451603 epoch 50000 loss : 0.001387911487010281 epoch 60000 loss : 0.0010717812890988993 epoch 70000 loss : 0.0008602174868596041 epoch 80000 loss : 0.0007104931924252216 epoch 90000 loss : 0.0005999848871505063	epoch 0 loss : 0.32600742840664376 epoch 10000 loss : 0.06876065230916775 epoch 20000 loss : 0.02420025119350303 6 epoch 30000 loss : 0.010053466709353786 epoch 40000 loss : 0.0037527954730983562 epoch 50000 loss : 0.0017987619675186 45 epoch 60000 loss : 0.0010675425588003809 epoch 70000 loss : 0.0007228508178834796 epoch 80000 loss : 0.0005319704413179335 epoch 90000 loss : 0.0004140076805439179

4. Discussion

A. Try different learning rates

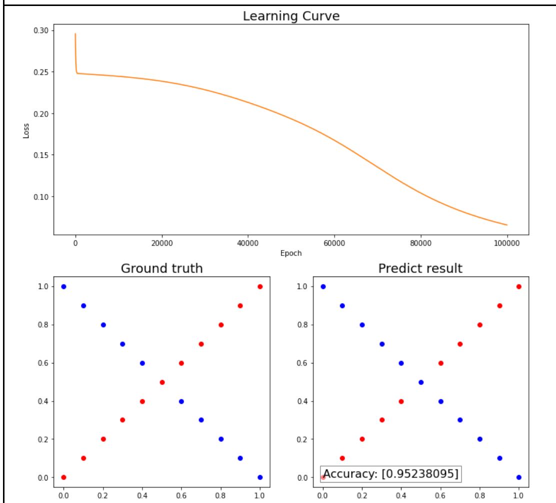
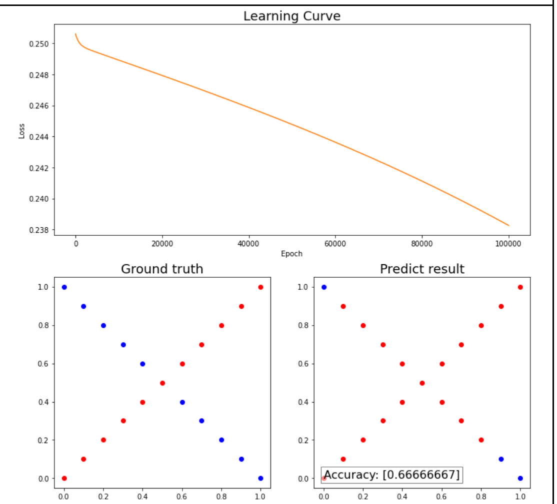
Hidden unit = 10, Optimizer = SGD, Activate function: sigmoid

Linear:

<table>
<tr><th>Lr = 1e-2</th><th>Lr = 1e-3</th></tr>
<tr><td>
</td><td>
</td></tr>
</table>

Lr 越小，模型參數更新幅度就越小，因此 loss 下降的速度越慢，準確率無法保持 100%。

XOR:

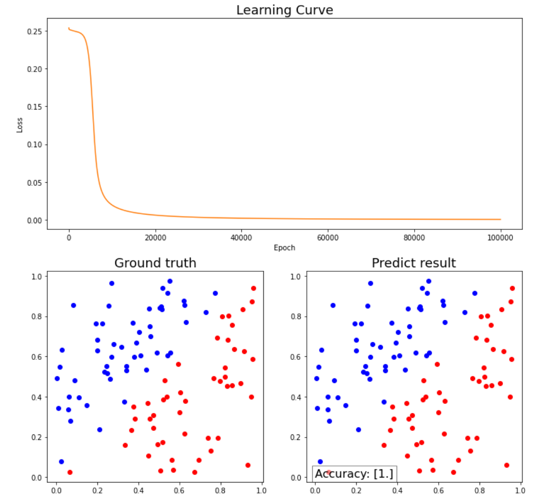
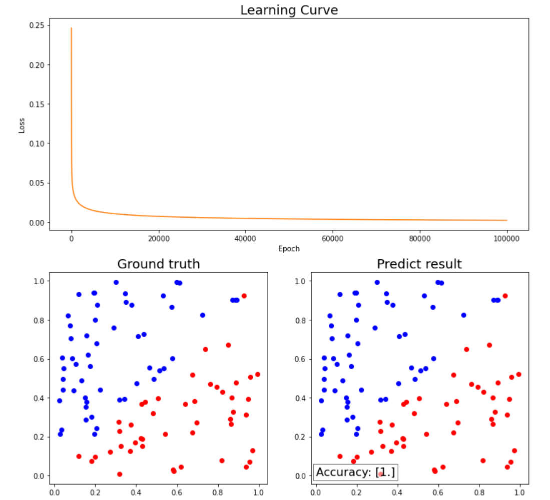
<table>
<tr><th>Lr = 1e-2</th><th>Lr = 1e-3</th></tr>
<tr><td>
</td><td>
</td></tr>
</table>

XOR data 所需的訓練量較大，lr 較小的情況下 accuracy 會大幅下降，尤其到 1e-3 時模型訓練量明顯不足。

<u>B. Try different numbers of hidden units</u>

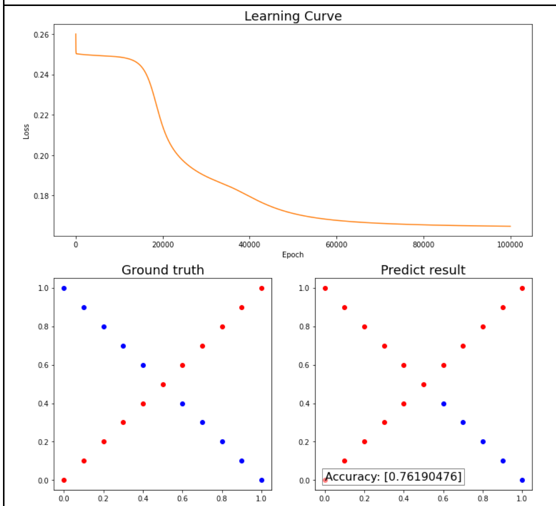
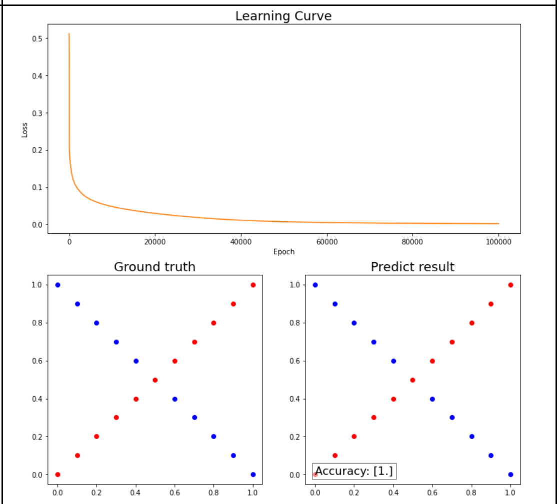
Lr = 0.1, Optimizer = SGD, Activate function: sigmoid

Linear:

Hidden Unit = 2	Hidden Unit = 100
	

對於 Lineaer Data 而言，與 part B 十個 hidden unit 相比，不論是減小或增大 hidden unit 數量對模型表現皆無明顯影響，但過多 hidden unit 會造成訓練時間大幅增加。

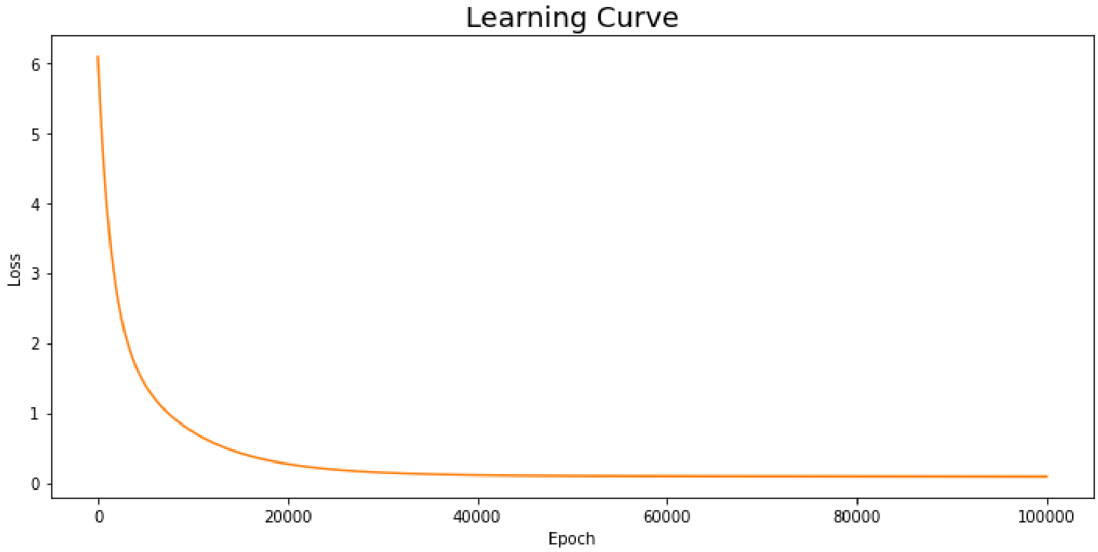
XOR:

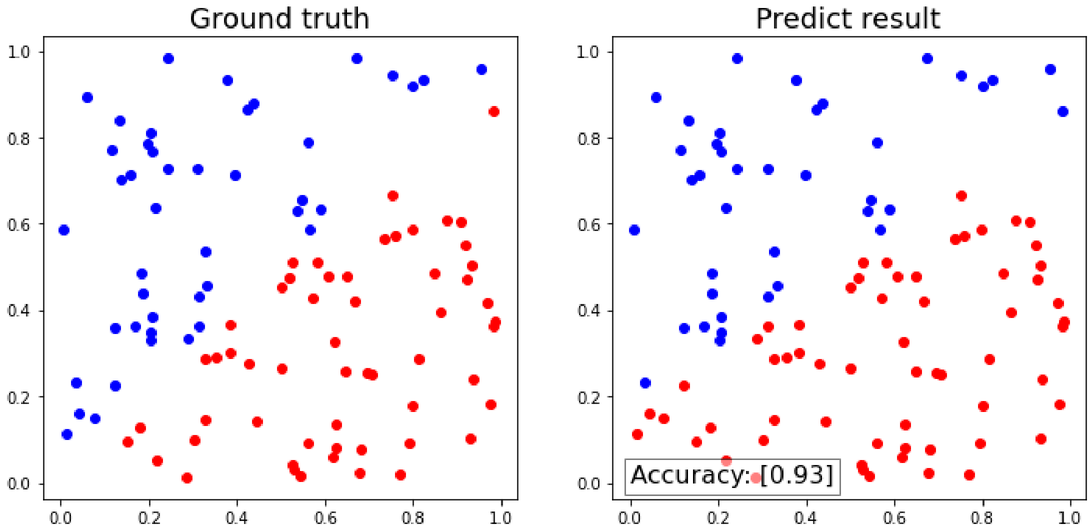
Hidden Unit = 2	Hidden Unit = 100
	

由於 XOR 的任務對模型來說較為複雜，減小 hidden size 會使模型訓練不太穩定，多次測試下來有時可以維持 95% accuracy，有時卻連 70%都無法達到；而在 hidden size 增加的情況下除了維持表現外，loss 也下降得更為快速。

C. Try without activation functions

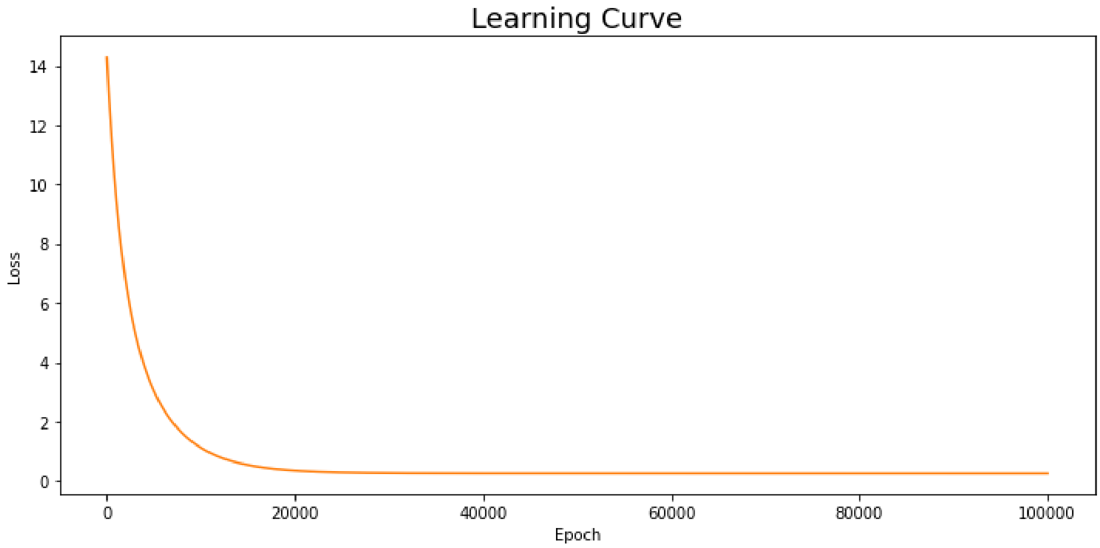
Linear:

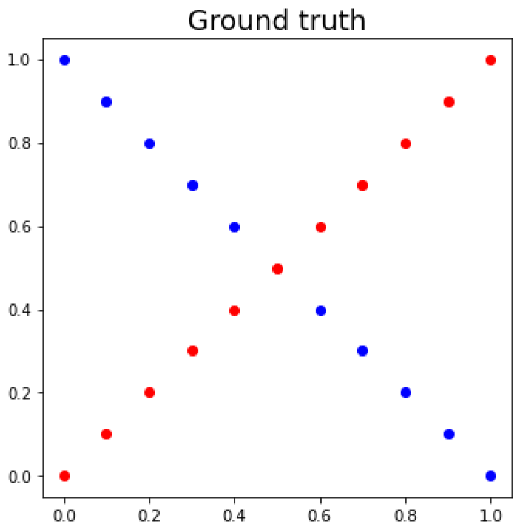


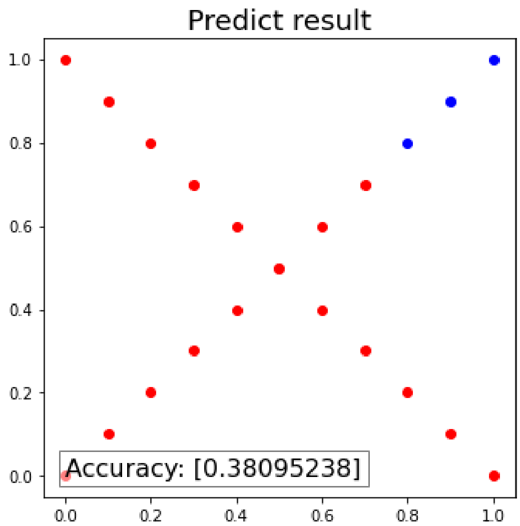


與 part B 中使用 sigmoid 相比並無明顯差異，剛開始訓練時的 loss 值變很高，且模型收斂得更快，表現也稍微降低。

XOR:







在 XOR data 中，移除 activation function 會使模型不足以學習 XOR 的資料分布。

5. Extra

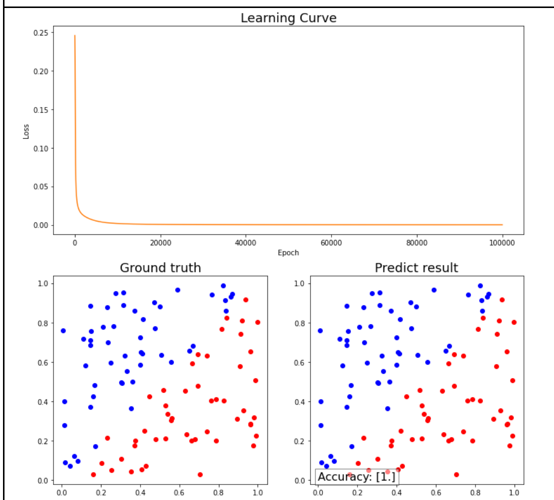
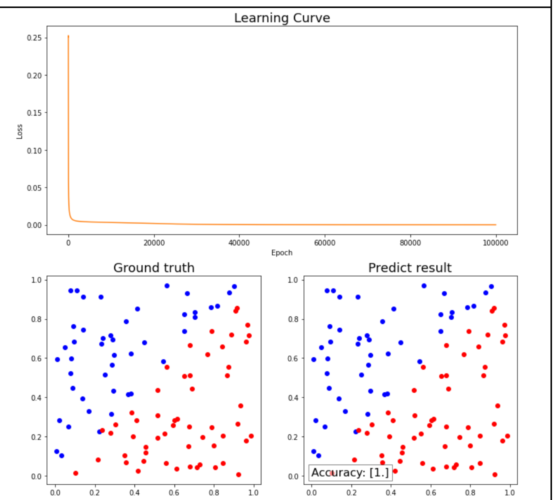
A. Implement different optimizers

除了 SGD 以外，我實做了 momentum 以及 AdaGrad 的 optimizer

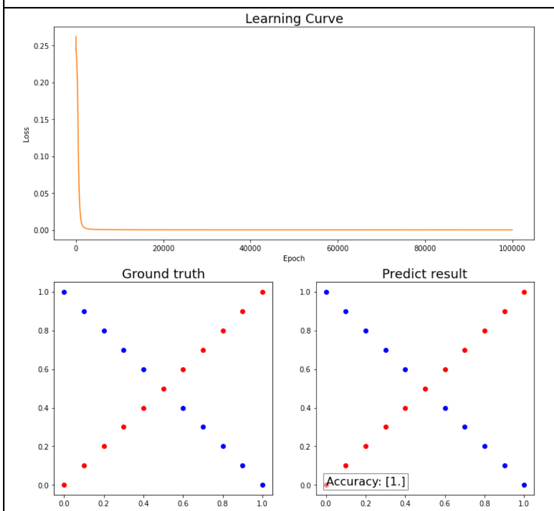
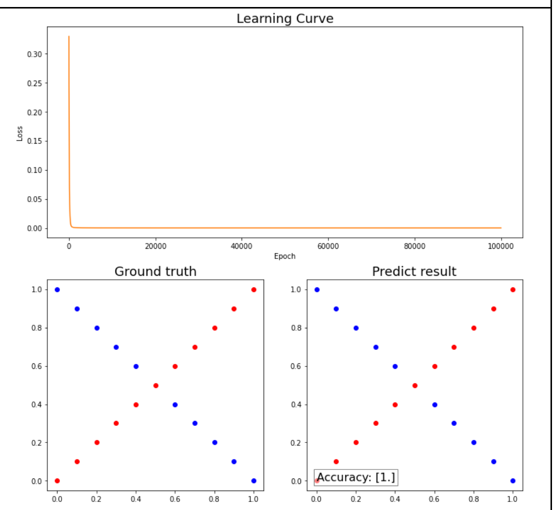
```
if optim == "SGD":
    self.b -= np.sum(grad) * lr
    self.w -= np.dot(self.x.T, grad) *lr
elif optim == "momentum":
    self.v_w = decay * self.v_w + lr * np.dot(self.x.T, grad)
    self.v_b = decay * self.v_b + lr * np.sum(grad)
    self.b -= self.v_b
    self.w -= self.v_w
elif optim == "AdaGrad":
    self.total_w += np.dot(self.x.T, grad) ** 2
    self.total_b += np.sum(grad) ** 2
    self.b -= np.sum(grad) * lr / (np.sqrt(self.total_b) + 1e-8)
    self.w -= np.dot(self.x.T, grad) * lr / (np.sqrt(self.total_w) + 1e-8)
```

Lr = 0.1, hidden unit = 10, activate function: Sigmoid

Linear:

momentum	AdaGrad
	

XOR:

momentum	AdaGrad
	

由於訓練梯度方向一致，使用 momentum 後模型的訓練速度變得更加快速，而使用 AdaGrad 則使 Lr 在初期較大，並隨時間遞減。

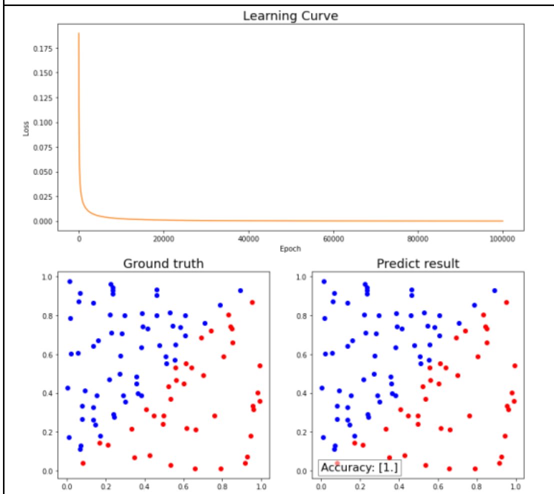
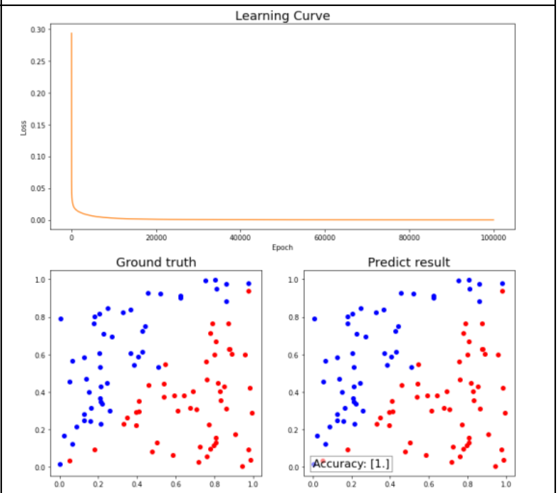
B. Implement different activation functions

除了 Sigmoid 外，我實作了 ReLU 以及 tanh。

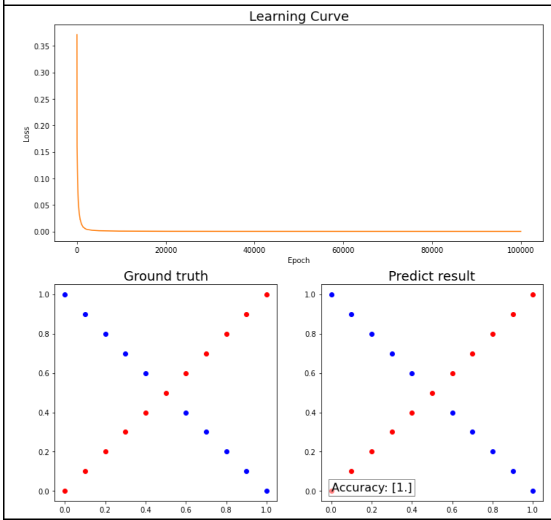
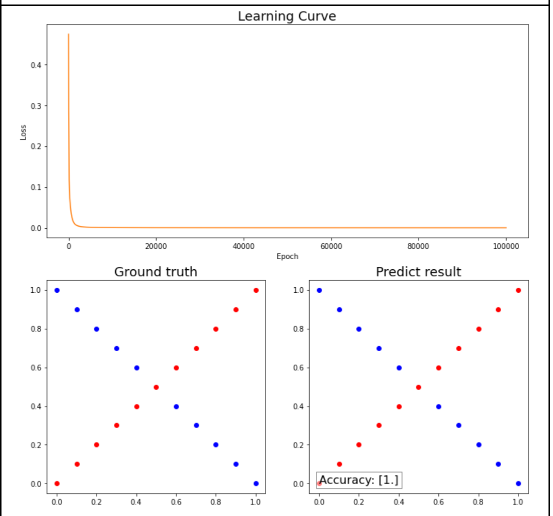
```
def ReLU(x):
    return np.where(x > 0, x, 0)

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

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

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

Lr = 0.1, hidden unit = 10, Optimizer = SGD

Linear:

ReLU	tanh
	

XOR:

ReLU	tanh
	

改變 activation function 使的 loss 的下降更為快速，且對 XOR 而言 loss 的曲線也更加圓滑。