1. Introduction

此次 lab 實作 neural network ， data 使用的是自己生成的(inputs, label) pair, code 中只使用 numpy 套件計算矩陣以及 matplotlib 來畫圖, neural network 實作 backpropagation 計算 gradient 拿來更新各個 layer 的 weight, 以使機器學習更接近 ground truth 的知識, neural network 以及 layer 各自透過 class 來實作, 其中因為要計算 gradient, 所以需要使用微分後的 activation function 和 weight 以及 forward 的資訊來計算 backward derivative, 以下為訓練流程

training process:

1. 生成 weight (np.random.normal)
2. 計算 forward (prediction)
3. 計算 loss , loss 的微分
4. 計算 backward
5. learn by gradient ($\text{forward}^T_{(\text{input})} * \text{backward}$)

2. Experiments setups

A. Sigmoid functions

```python
# activation functions
def sigmoid(x):
    return 1.0/(1.0 + np.exp(-x))

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

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

$$\frac{d}{dx}\sigma(x) = \frac{e^{-x}}{(1+e^{-x})^2} = (1-\sigma(x)) * \sigma(x)$$

B. Neural network

```
class NeuralNetwork:
    def __init__(self,epoch, learning_rate, layers, inputs, hidden_units, activation, optimizer):

    def forward(self, inputs):...

    def backward(self, loss_derivative):...

    def learn(self):...

    def train(self, inputs, ground_truth):...
```

如圖, neural network 當中有實作 forward, backward, learn, 以及 train 等等的 functions, 而 train 則會在每個 epoch(掃過一次所有資料)去做 Introduction 中圖片的 training process

```
def train(self, inputs, ground_truth):
    for epoch in range(self.epoch):
        prediction = self.forward(inputs)
        loss = mse_loss(prediction, ground_truth)
        self.backward(mse_loss_derivative(prediction, ground_truth))
        self.learn()

        self.prediction = prediction
        if epoch%100 == 0:
            print(f'Epoch {epoch} loss : {loss}')
        if loss < 0.001:
            break
```

其中最核心的部分其實是實作 layer 的部分

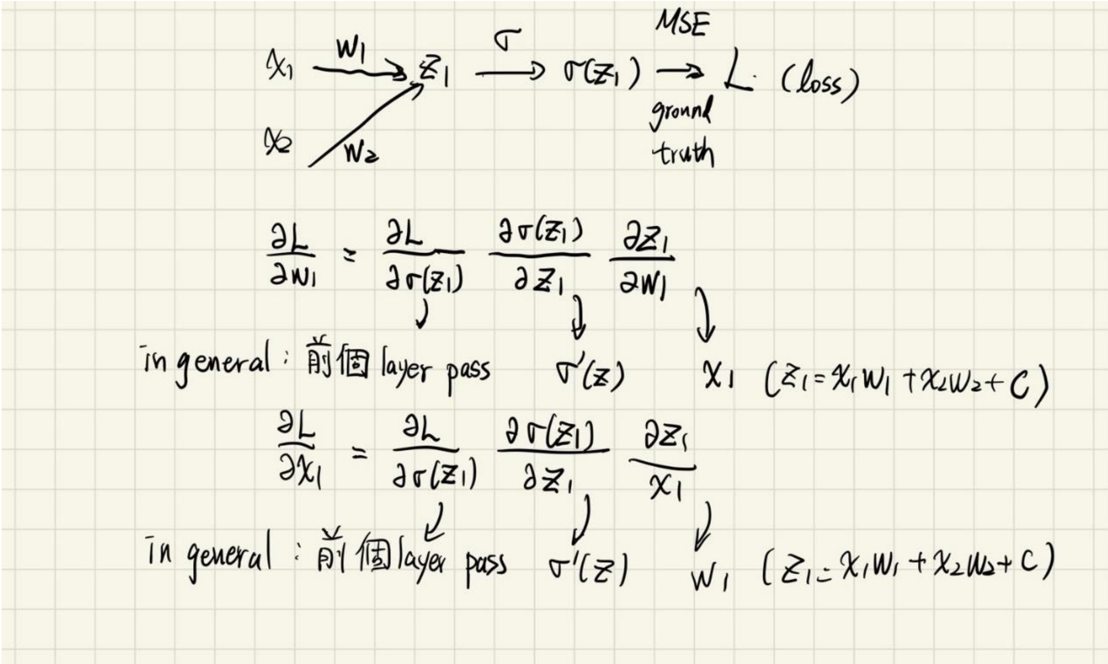
```
class Layer:
    def __init__(self, input_num, output_num, activation , optimizer , learning_rate ):

    def forward_pass(self, inputs):...

    def backward_pass(self, inputs):...

    def learn(self):...
```

其中會根據 input size, output size 生成 random 的 weight, 以及實作 forward pass, backward pass 的計算, learn 的部分則是根據計算出來的 gradient 去進行梯度下降的學習(gradient descent)

C. Backpropagation

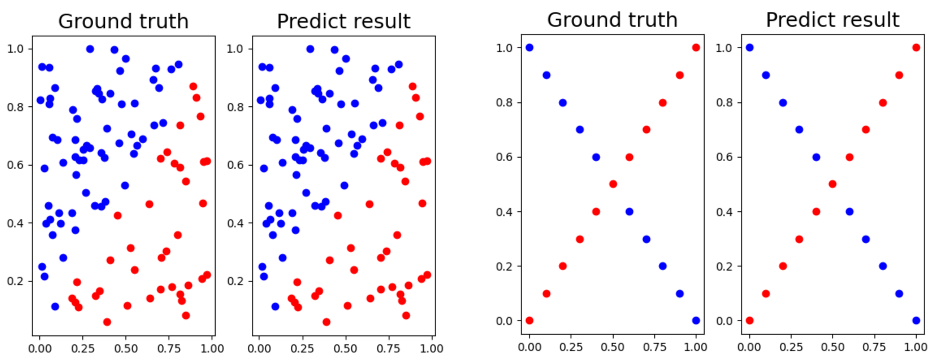
```
def backward_pass(self, inputs):
    self.backward_output = None
    if self.activation == 'sigmoid':
        self.backward_output = np.multiply(derivative_sigmoid(self.forward_output),inputs)

    return np.matmul(self.backward_output, self.weight[:-1].T)
```

Backward pass 的部分會先將 pass 過來的 input(derivative loss)跟 sigmoid 之微分相乘(elementwise), 並且乘上 weight 再往前傳, 以下為推導圖



因此，若不是 input layer，則乘上 weight 的矩陣並將結果往前傳，若是的話則乘上 input 的 vector 就會是 input layer 的 weight

3. Result of your testing
   A. Screenshot and comparison figure (left is linear, right is XOR)



   B. Show the accuracy of your prediction

```python
def main():
    inputs1, label1 = generate_linear()
    inputs2, label2 = generate_XOR_easy()
    # print(inputs2)
    # print(label2)
    network = NeuralNetwork(epoch = 1000000, learning_rate = 0.01, layers = 2, inputs = 2, hidden_units = 4,
                            activation = 'sigmoid', optimizer = 'gd')
    prediction = network.train(inputs1, label1)
    show_result(inputs1, label1, prediction,'linear.png')
    print('Accuracy : ', float(np.sum(prediction == label1)) / len(label1))
    # prediction = network.train(inputs2, label2)
    # show_result(inputs2, label2,prediction,'XOR.png')
    # print('Accuracy : ', float(np.sum(prediction == label2)) / len(label2))
```

Linear:

```
Accuracy :  1.0
```

```python
def main():
    inputs1, label1 = generate_linear()
    inputs2, label2 = generate_XOR_easy()
    # print(inputs2)
    # print(label2)
    network = NeuralNetwork(epoch = 1000000, learning_rate = 0.01, layers = 2, inputs = 2, hidden_units = 4,
                            activation = 'sigmoid', optimizer = 'gd')
    # prediction = network.train(inputs1, label1)
    # show_result(inputs1, label1, prediction,'linear.png')
    # print('Accuracy : ', float(np.sum(prediction == label1)) / len(label1))
    prediction = network.train(inputs2, label2)
    show_result(inputs2, label2,prediction,'XOR.png')
    print('Accuracy : ', float(np.sum(prediction == label2)) / len(label2))
```

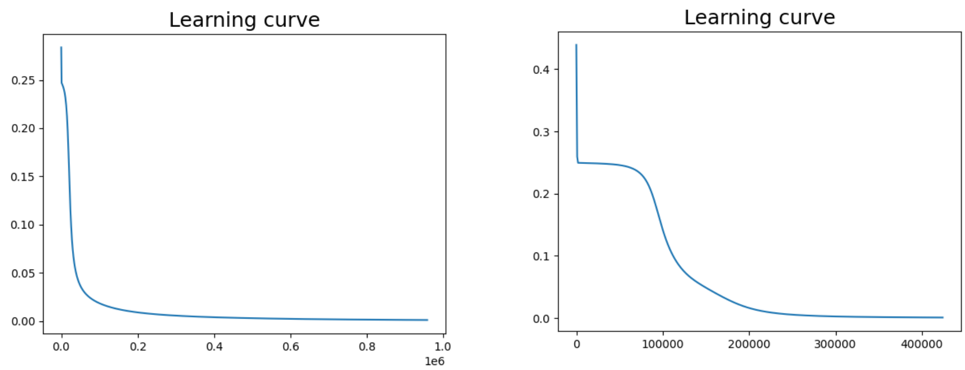
XOR:

```
Accuracy :  1.0
```

兩個預測準確度都是 100%

C. Learning Curve

(left is linear, right is XOR)



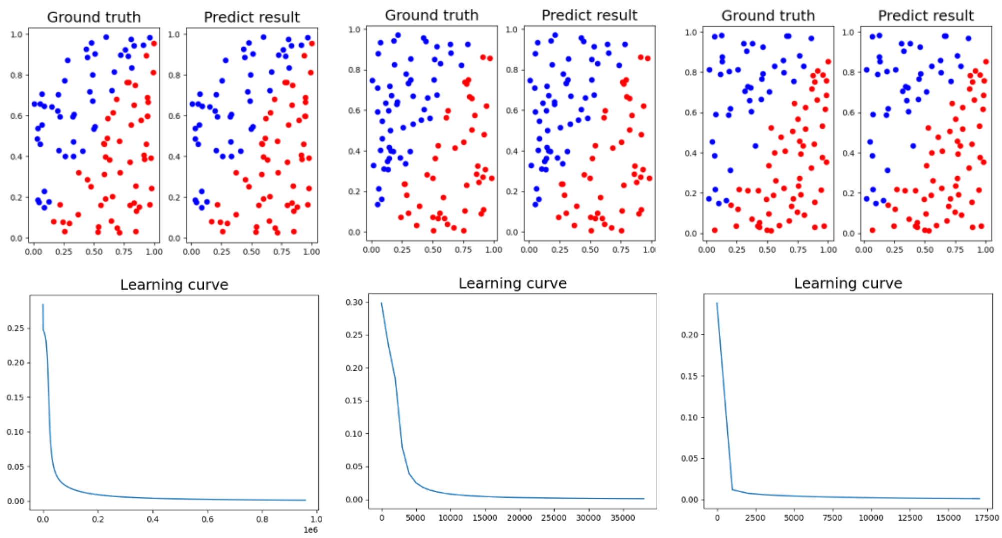
可以觀察到, linear 的 learning curve 是較為平滑的

4. Discussion

A. Try different learning rates

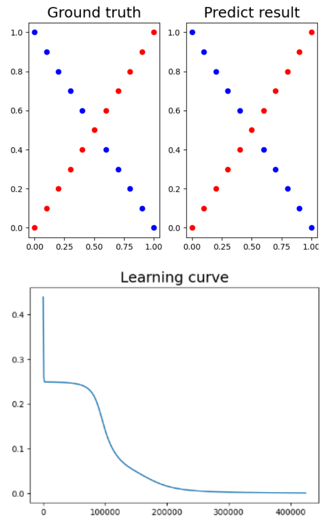
(linear):

Learning rate: 0.01 Learning rate: 0.1 Learning rate: 1

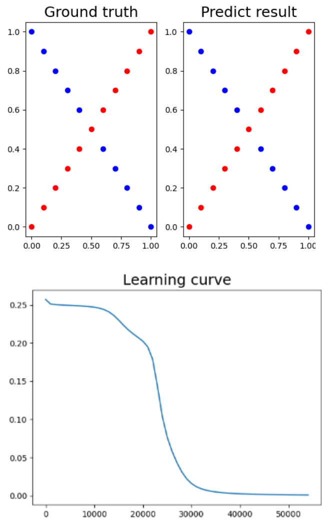


(XOR):

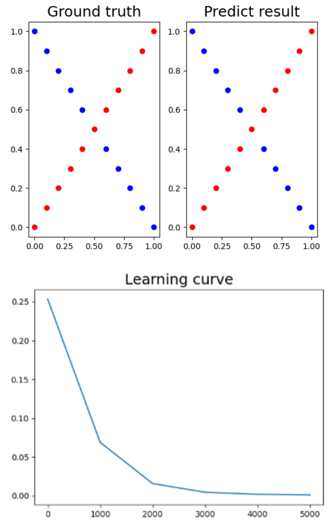
Learning rate: 0.01



Learning rate: 0.1



Learning rate: 1

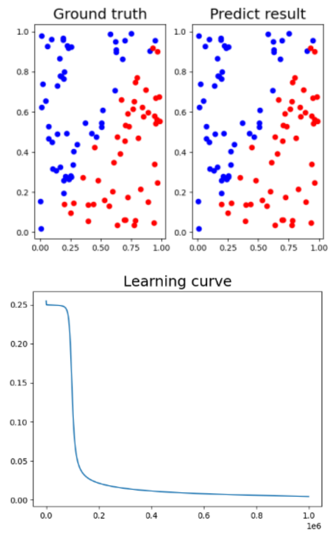


從上面的實驗可以得知, learning rate 越高不一定代表 loss 收斂的越快, XOR 在 0.1 和 0.01 時的表現其實差不多，甚至 0.1 更差一點

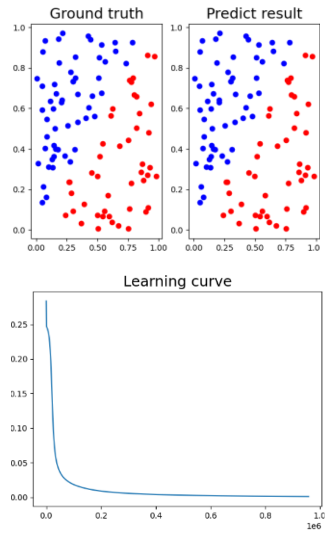
B. Try different numbers of hidden units (learning rate = 0.01)

(linear):

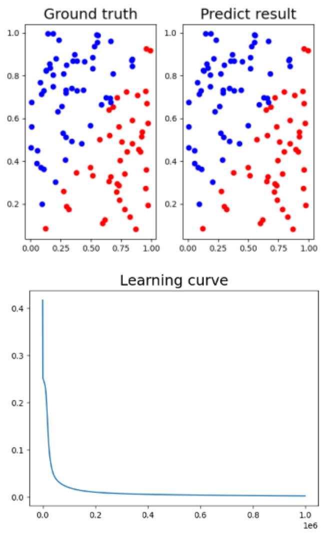
hidden units: 2



hidden units: 4

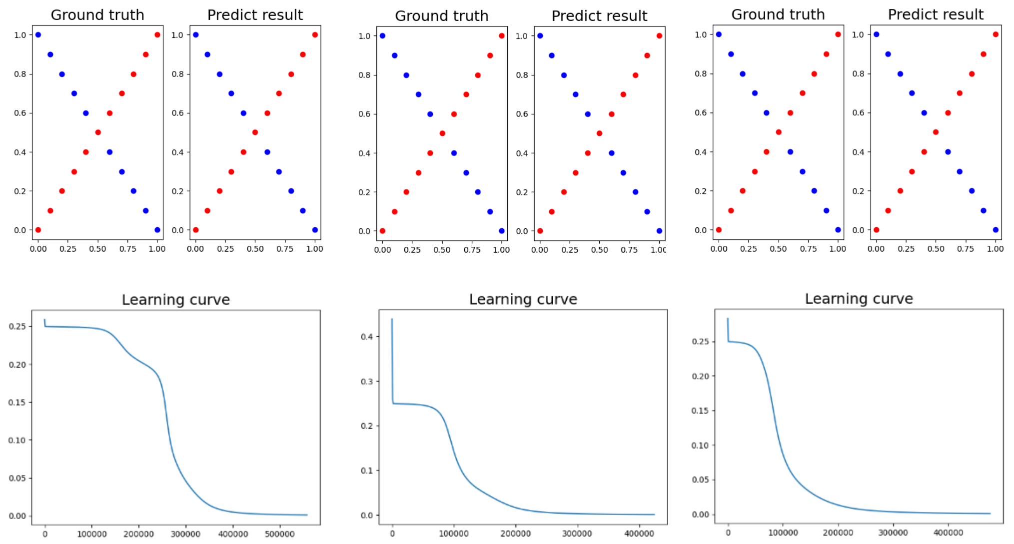


hidden units: 6



(XOR):

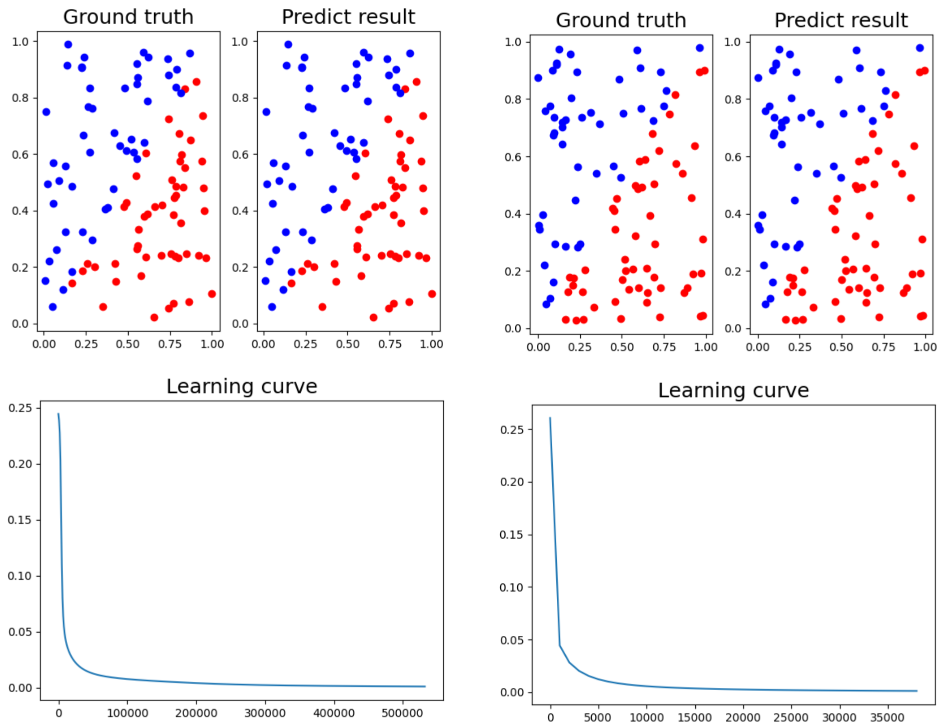
hidden units: 2　　hidden units: 4　　hidden units: 6



由以上實驗可得知, hidden unit 也不一定是越多越好, 反而可能造成模型過於複雜並造成 loss 較高(觀察 linear 的例子)

C. Try without activation functions (learning rate =0.05, hidden unit =4) (linear)

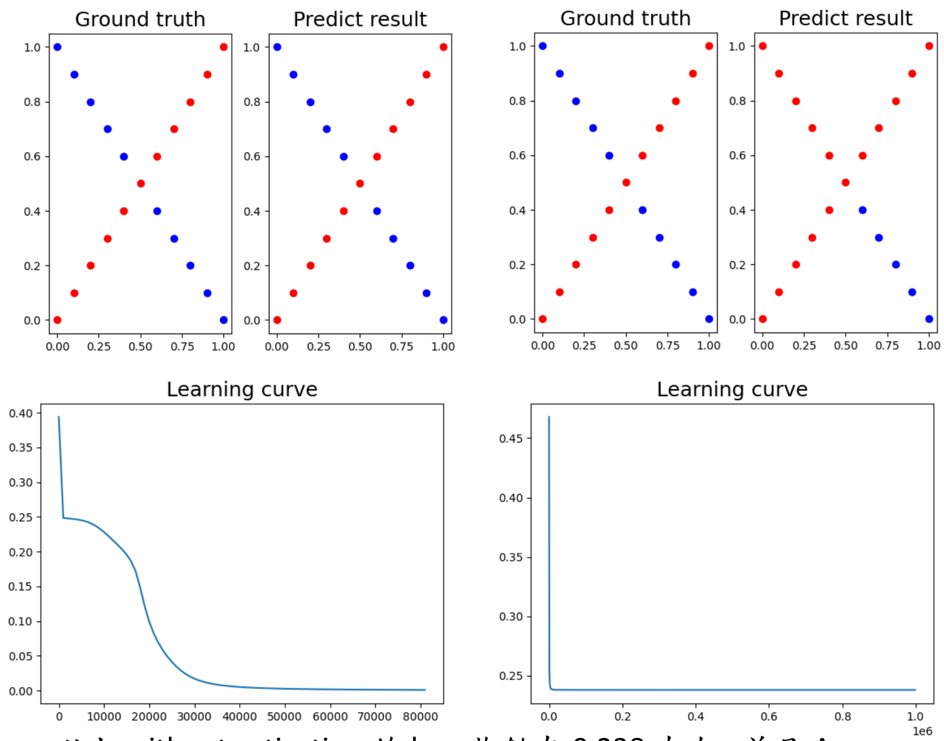
With activation　　Without activation



以上兩者 accuracy 皆為 1.0 但 with activation function 的收斂的比較慢

(XOR)

With activation　　　　　　Without activation



以上 without activation 的 loss 收斂在 0.238 左右，並且 Accuracy 只有 0.761，左上角的資料預測錯誤

5. Extra
   A. Implement different optimizer
      a. Momentum

模擬物理中動量的概念，在同方向上的維度學習的速度會變快，方向改變的時候學習速度會變慢

以下為 momentum optimizer 更新公式

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

$$W_t \leftarrow W_{t-1} + V_t$$

其中，

V_t 為momentum,

β 可以想像成空氣阻力或是地面摩擦力，通常設定成 0.9,

η 為learning rate,

L 為 loss function,

W 為 weight 權重

Implementation:

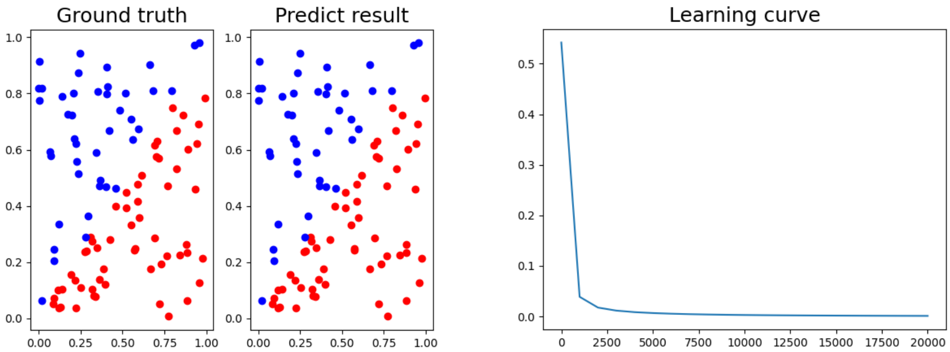
In Layer constructor, set momentum to 0

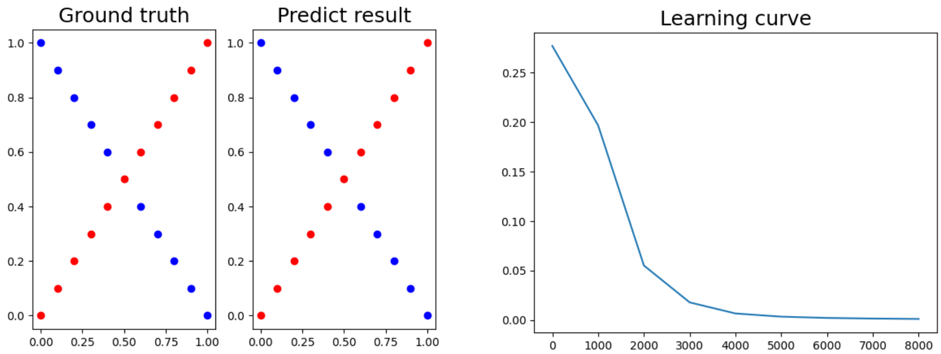
```
self.momentum = 0
```

in Layer.learn(), implement momentum algorithm

```
if self.optimizer == 'momentum':
            self.momentum = beta * self.momentum - self.learning_rate * gradient
            weight_change = self.momentum
```

- Result and Learning curve (learning rate = 0.05, hidden units =4 , layers =2 ,activation = ‘sigmoid’)





b. Adagrad

AdaGrad 就會依照梯度去調整 learning rate, 公式為以下

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

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

ε 為了使分母不為 0 通常設為 1e-8

implementation:

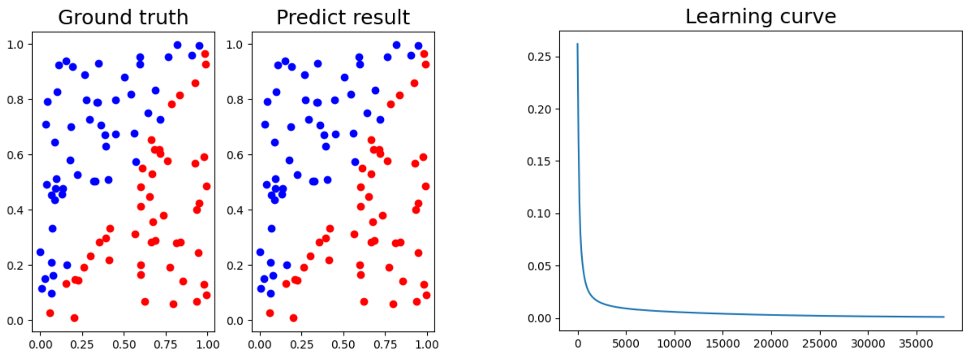
In layer constructor, set n = 0

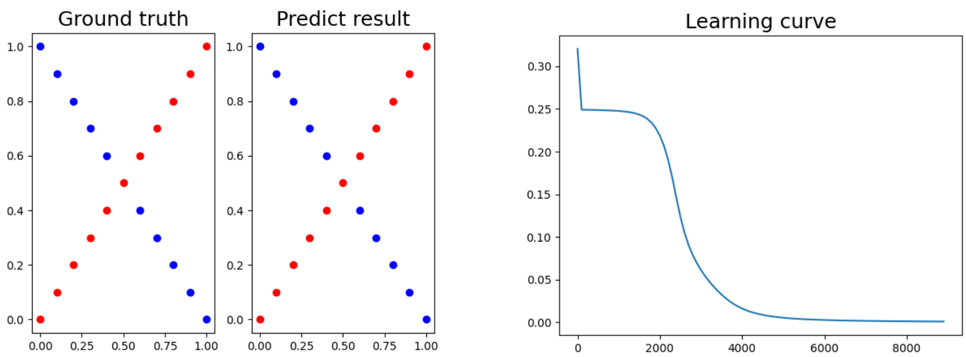
```
self.n = 0
```

in Layer.learn(), implement Adagrad algorithm

```
if self.optimizer == 'adagrad':
            self.n += np.square(gradient)
            weight_change = -self.learning_rate *
np.multiply(1/(np.sqrt(self.n + epsilon)), gradient)
```

Result and Learning curve (learning rate = 0.05, hidden units =4 , layers =2 ,activation = ‘sigmoid’)





B. Implement different activation functions

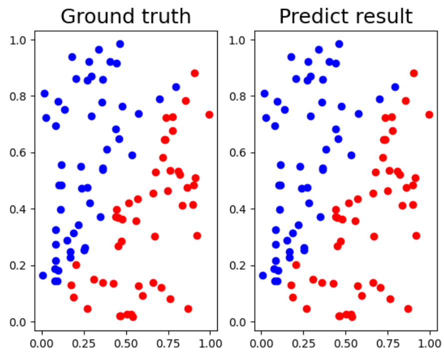
a. implementations

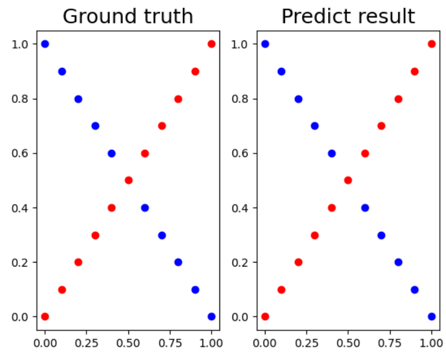
```
def tanh(x):
    return np.tanh(x)

def derivative_tanh(x):
    return 1.0 - x**2

def relu(x):
    return np.maximum(0.0, x)

def derivative_relu(x):
    return np.heaviside(x, 0.0)

def leaky_relu(x):
    alpha = 0.005
    return np.maximum(alpha * x, x)

def derivative_leaky_relu(x):
    alpha = 0.005
    y = copy.deepcopy(x);
    y[y > 0.0] = 1.0
    y[y <= 0.0] = alpha
    return y
```

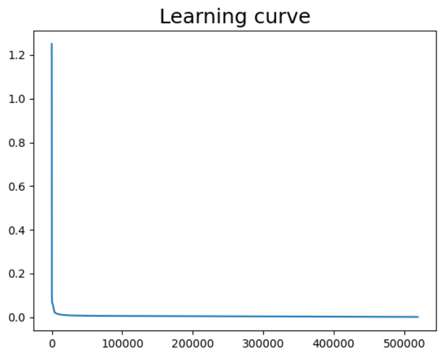
- $\tanh(x) = \frac{e^x - e^x}{e^x + e^x}$
- $\frac{d}{dx}\tanh(x) = \frac{(e^x+e^{-x})(e^x-e^{-x})-(e^x-e^{-x})(e^x-e^{-x})}{(e^x+e^{-x})^2} = 1-\frac{(e^x-e^{-x})^2}{(e^x+e^{-x})^2} = 1\text{-}\tanh^2(x)$
- $\mathrm{ReLu}(x) = \max(x, 0)$
- $\frac{d}{dx}\mathrm{ReLu}(x) = \begin{cases} 0 \text{ if } x < 0 \\ 0 \text{ if } x == 0 \\ 1 \text{ if } x > 0 \end{cases}$, by definition in numpy document, it is equivalent to np.heaviside(x,0)
- $\mathrm{leaky\_ReLu}(x) =$ $\max(\alpha x, x)$, where α is a small number, we set it 0.005 there
- $\frac{d}{dx}\mathrm{leaky\_ReLu}(x) = \begin{cases} \alpha \text{ if } x < 0 \\ \alpha \text{ if } x == 0 \\ 1 \text{ if } x > 0 \end{cases}$, then we can define it as above

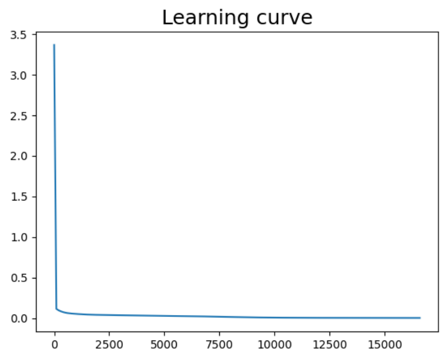
b. 實驗(optimizer = ‘gd’, learning rate = 0.05)

(tanh)

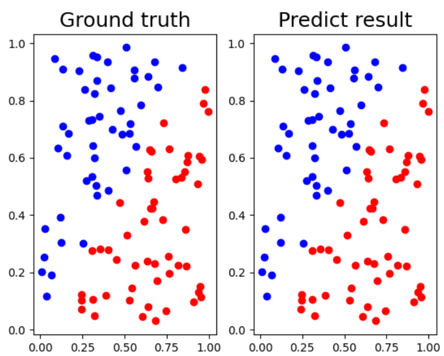


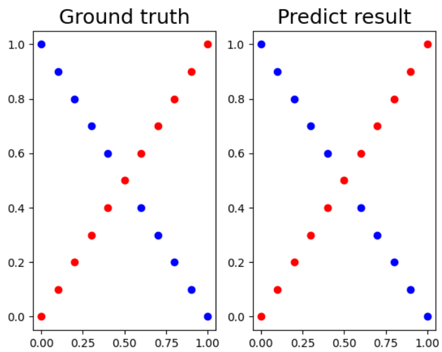


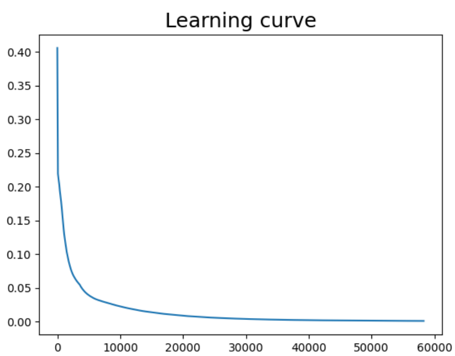


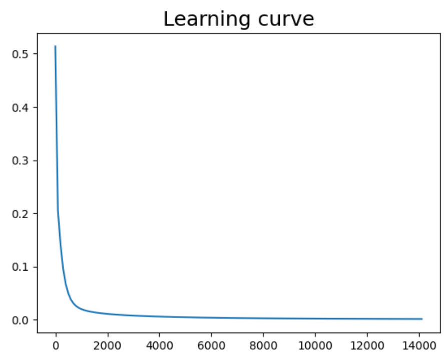


(relu)

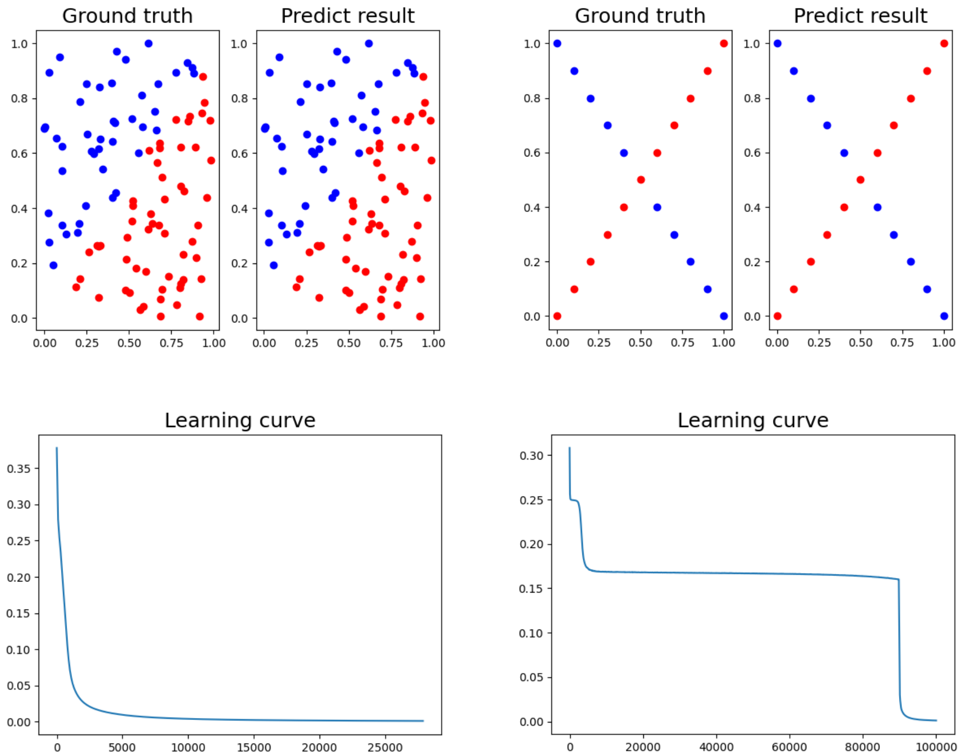








(leaky_relu)


Ground truth
Predict result
Ground truth
Predict result
Learning curve
Learning curve