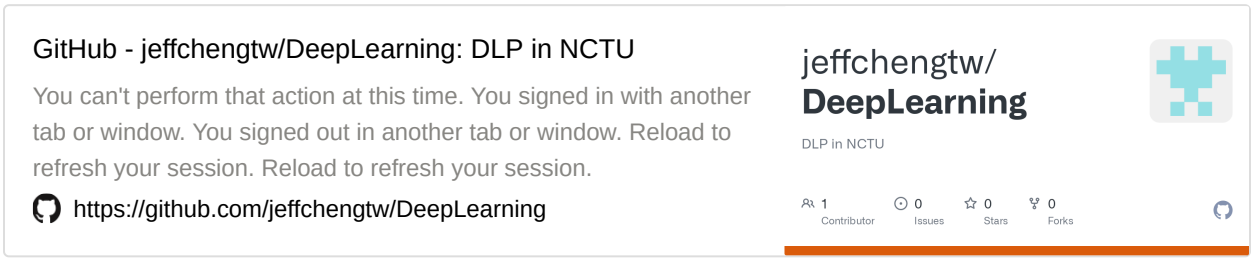
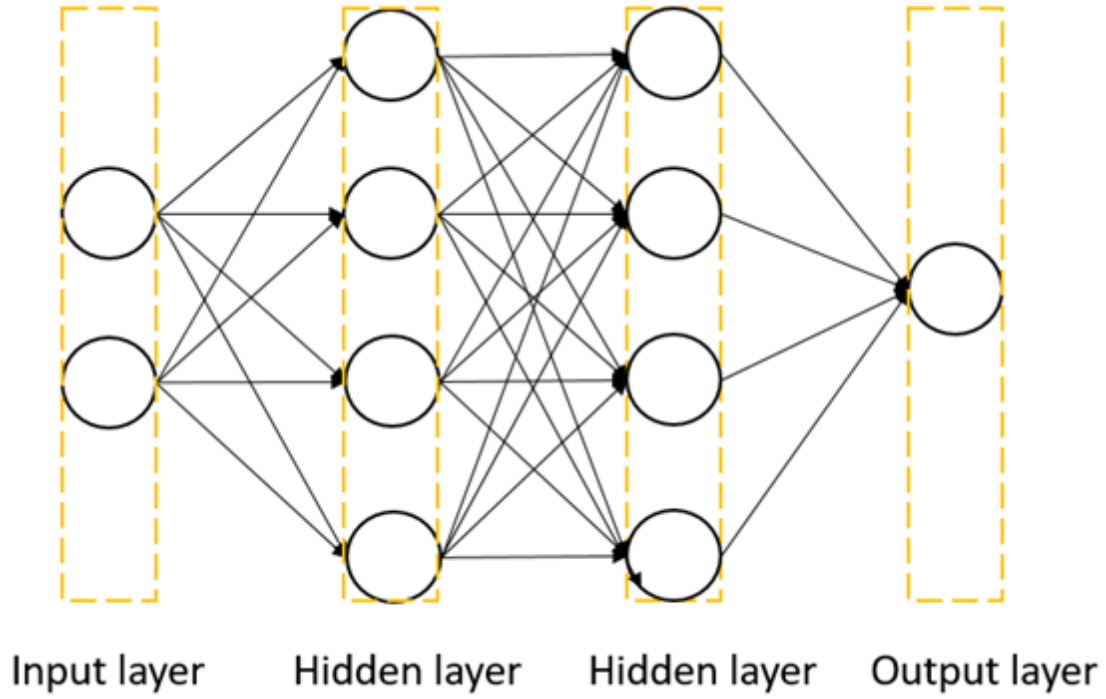
Lab01 - Back-propagation

Jeff Cheng 410551012



Lab Objective

In this lab, you will need to understand and implement simple neural networks with forwarding pass and backpropagation using two hidden layers. Notice that you can only use **Numpy** and the python standard libraries, any other frameworks (ex : Tensorflow、PyTorch) are not allowed in this lab.



Basic Idea

We are trying to find the best network parameters $\theta\{w_1, w_2, w_3, ..., b_1, b_2\}$ that minimize the loss L . When we get the gradient ∇L , update the network parameter w by **Gradient Descent.**

Gradient Descent

Actually we have a lot of ∇L, it is a vector :

$$\begin{bmatrix} \frac{\partial L(\theta)}{\partial w_1} \\ \frac{\partial L(\theta)}{\partial w_2} \\ \cdot \\ \cdot \\ \cdot \\ \frac{\partial L(\theta)}{\partial b_1} \end{bmatrix} \tag{1}$$

Then compute $\nabla L(\theta^0), \nabla L(\theta^1), \nabla L(\theta^2)...$

Update the θ , $\theta^1 = \theta^0 - \eta \nabla L(\theta^0)$, we get new θ^1.

Compute ∇L

$L(\theta) = \sum_{n=1}^{N} C^n(\theta)$, where C^n is the distance between y_{pred}, y_{gt}

Then do the partial to all parameters, we get

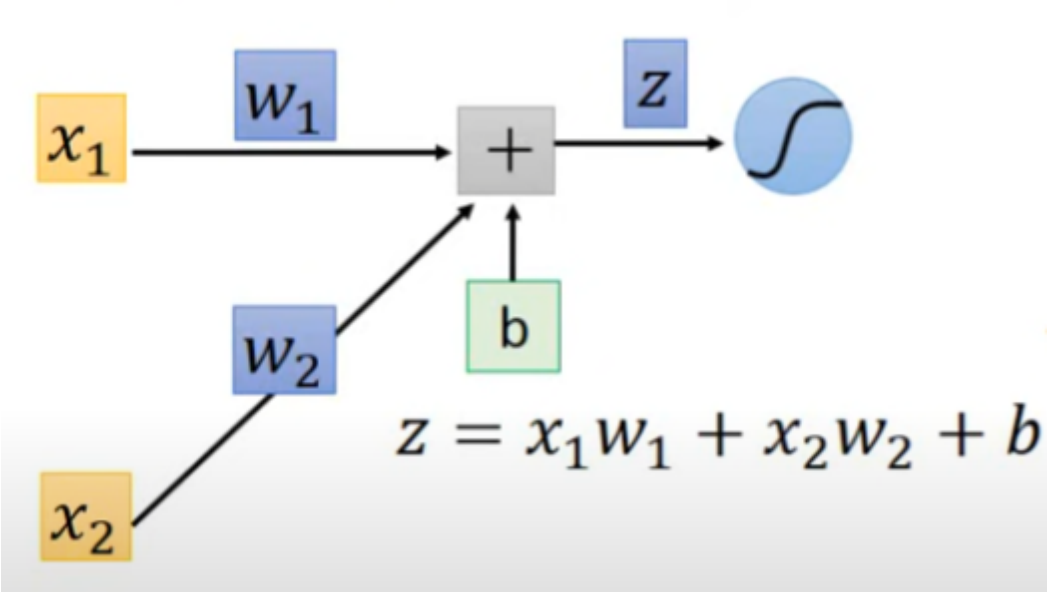
$$\frac{\partial L(\theta)}{\partial w} = \sum_{n=1}^{N} \frac{\partial C^n(\theta)}{\partial w}$$

Next, we will focus on how to compute $\frac{\partial C^n(\theta)}{\partial w}$.

Chain Rule

$$\frac{\partial C}{\partial w} = \frac{\partial z}{\partial w}\frac{\partial C}{\partial z}$$

Forward pass :



Compute $\frac{\partial z}{\partial w}$ for all parameters.

$$\frac{\partial z}{\partial w} = x_1$$

Backward pass :

Compute $\frac{\partial C}{\partial z}$ for all activation function inputs z.

Use Chain rule again :

$$\frac{\partial C}{\partial z} = \frac{\partial a}{\partial z}\frac{\partial C}{\partial a}$$

It is easy to compute the first term :

$$\frac{\partial a}{\partial z} = \sigma'(z)$$

The difficult part is the second term that we have to concern about two cases :

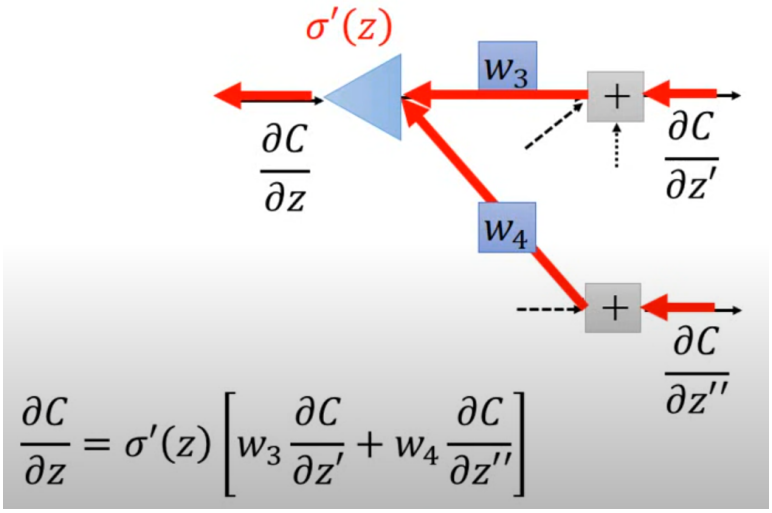
- output → hidden
- hidden → hidden

In this part, we have to start from the "known parameters". So we will do the **"output → hidden"** case first.

$$\frac{\partial C}{\partial z} = L'(y, \hat{y})$$

Now that we know the gradient of "output → hidden", we can continue to derive the previous layer **"hidden → hidden"**.
As the figure shown, we can now compute the output of previous node, then we get the backward gradient :

$$\frac{\partial C}{\partial z} = \sigma'(z)[w_3 \frac{\partial C}{\partial z'} + w_4 \frac{\partial C}{\partial z''}]$$



Implement

Neural Network Architecture

- **Input layer** with input data $\{(x_{11}, x_{12}), (x_{21}, x_{22}), ...(x_{n1}, x_{n2})\}$
- **First hidden layer** with 4 neurons and input 2 values for x_1, x_2

- **Second hidden layer** with 4 neurons and input 4 values for previous 4 output a, where $a = \sigma(wx)$, and 1 output for classification.

Forward pass

As the code shown in **“def forward()”**, there will two value output from two hidden layer which is so called **“forward gradient ”** $\frac{\partial z}{\partial w}$, the direct of **forward pass is : input → hidden_1st → hidden_2nd**

Backward pass

As the code shown in “def backward()”, there will be two output value which is so called **“backward gradient ”** $\frac{\partial C}{\partial z}$, the direct of backward pass is : **output (loss computation) → hidden_2nd → hidden_1st .**

Update (Gradient Descent)

After finishing forward pass and backward pass, we can compute the ∇L by matrix **multiplication of forward gradient and backward gradient :** $\frac{\partial C}{\partial w} = \frac{\partial z}{\partial w}\frac{\partial C}{\partial z}$**, then do gradient descent.**

```
class Model():
    def __init__(self) -> None:
        self.hidden_layer_1 = layer(2, 4)
        self.hidden_layer_2 = layer(4, 1)
        pass

    def forward(self, X):
        a1 = self.hidden_layer_1.forward(X)
        a2 = self.hidden_layer_2.forward(a1)
        return a2

    def backward(self, d_loss):
        g1 = self.hidden_layer_2.backward(d_loss)
        g2 = self.hidden_layer_1.backward(g1)
        return g1, g2

    def update(self):
        self.hidden_layer_2.update_param()
        self.hidden_layer_1.update_param()
```

Layer

More detail are defined in “class layer”.

Initial model parameter $\theta = \{w_1, w_2, w_3, ...\}$

w is generated in $\{0, 1\}$ randomly.

Forward

$z = w^T x + b$, but there is no bias in my implement.

$a = \sigma(z)$, where σ is the activation function. (**function output**)

We can easily compute the **forward gradient : $\frac{\partial z}{\partial w} = x'$.**

Backward

The key point of backward is to compute from the last node. The loss computation must be the last process. And in the backward pass, the first known parameter is “loss”. It means that we need to derivative of loss. Then multiply with derivative of output of previous node.

Gradient Descent

$\theta^1 = \theta^0 - \eta \nabla L(\theta^0)$, where $\nabla L(\theta^0)$ is $\frac{\partial C}{\partial w} = \frac{\partial z}{\partial w} \frac{\partial C}{\partial z}$

```
class layer():
    def __init__(self, input_size, output_size) -> None:
        # neuron
        self.w = np.random.normal(0, 1, (input_size, output_size))
        self.gradient = None

    def forward(self, X):
        self.forward_gradient = X
        z = X@self.w
        self.a = sigmoid(z)
        return self.a

    def backward(self, derivative_C):
        self.backward_gradient = np.multiply(
            derivative_sigmoid(self.a),
            derivative_C
        )
        return np.matmul(self.backward_gradient, self.w.T)

    def update_param(self):
        self.gradient = np.matmul(
            self.forward_gradient.T,
            self.backward_gradient
        )
        self.w += LEARNING_RATE*self.gradient
```

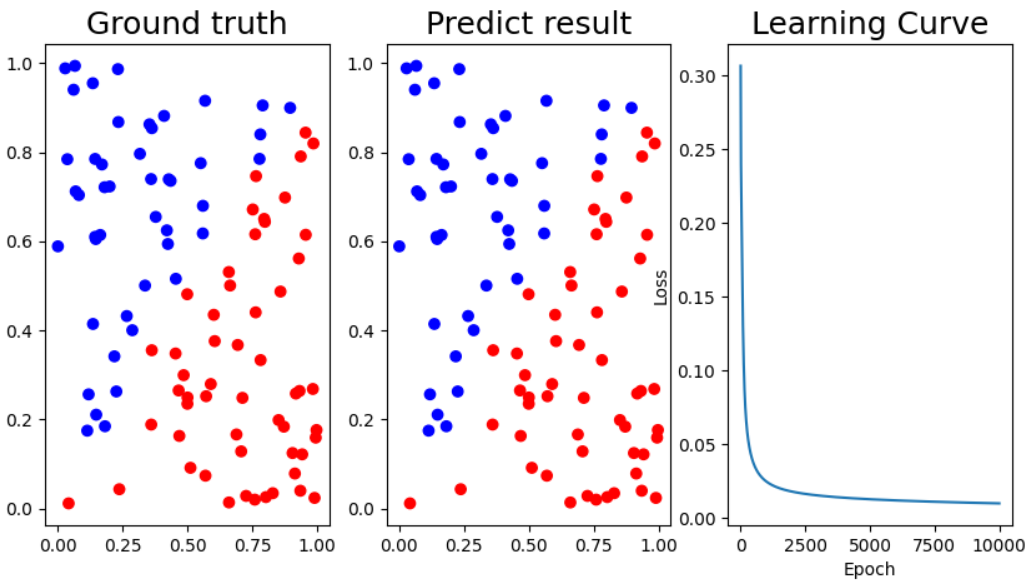
Utils

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

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

def MSE(y, y_hat):
    return np.mean((y - y_hat)**2)

def derivative_loss(y, y_hat):
    return (y - y_hat)*(2/y.shape[0])
```

Result of testing

Result of linear data (num=100)

- Learning rate = 1

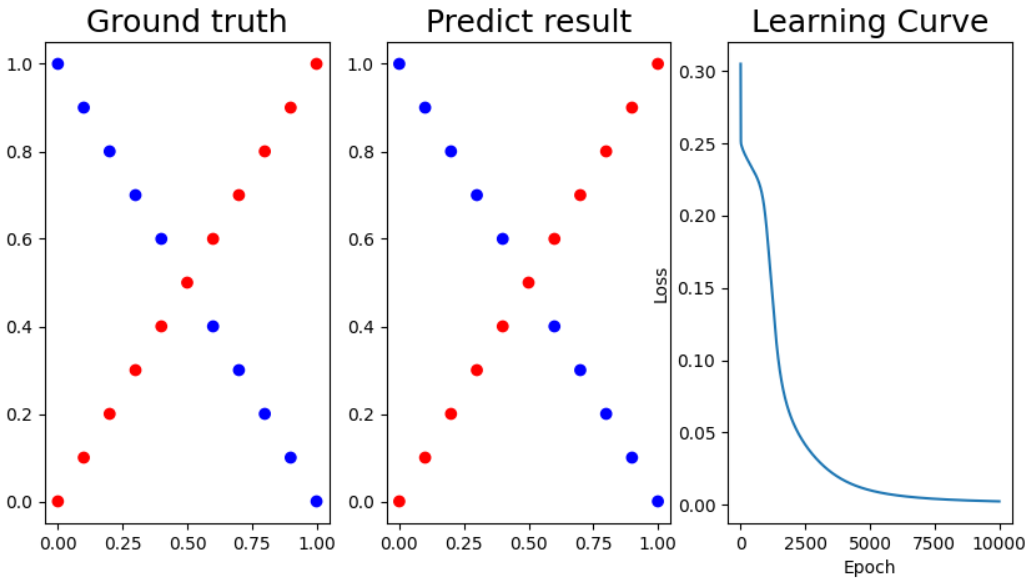



```
epoch : 0 loss : 0.32138588248299527
epoch : 500 loss : 0.04979340691714258
```

```
epoch : 1000 loss : 0.03309575531973608
epoch : 1500 loss : 0.02600506476746792
epoch : 2000 loss : 0.021652521579702256
epoch : 2500 loss : 0.018632668072133225
epoch : 3000 loss : 0.016399251371669848
epoch : 3500 loss : 0.014677134145623024
epoch : 4000 loss : 0.013307704081058573
epoch : 4500 loss : 0.012191875629055332
epoch : 5000 loss : 0.011264344852054327
epoch : 5500 loss : 0.010480342319845022
epoch : 6000 loss : 0.009808200847012118
epoch : 6500 loss : 0.00922492746147165
epoch : 7000 loss : 0.008713446065144482
epoch : 7500 loss : 0.008260820157937263
epoch : 8000 loss : 0.007857074283190829
epoch : 8500 loss : 0.0074943932574085634
epoch : 9000 loss : 0.007166566293786909
epoch : 9500 loss : 0.00686859361882803
```

Result of XOR data

- Learning rate = 1



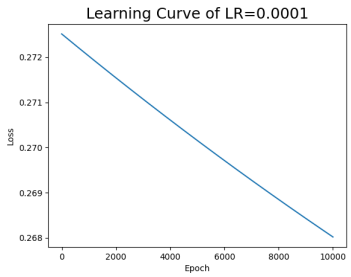

```
epoch : 0 loss : 0.45013681053040233
epoch : 500 loss : 0.24814994649149305
epoch : 1000 loss : 0.2358035259431213
epoch : 1500 loss : 0.2180768318764983
epoch : 2000 loss : 0.1791621350035321
```

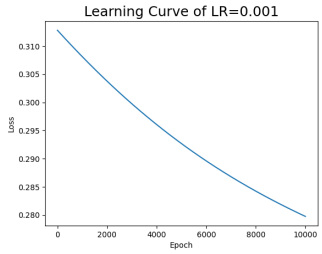
```
epoch : 2500 loss : 0.09971261235449831
epoch : 3000 loss : 0.0666243049679618
epoch : 3500 loss : 0.0488387724783186
epoch : 4000 loss : 0.036797403541448524
epoch : 4500 loss : 0.027910348502011722
epoch : 5000 loss : 0.021311195887315112
epoch : 5500 loss : 0.016480875118326035
epoch : 6000 loss : 0.012970614853408577
epoch : 6500 loss : 0.010407751819488002
epoch : 7000 loss : 0.008512287745964029
epoch : 7500 loss : 0.007086557412801751
epoch : 8000 loss : 0.005994676770088541
epoch : 8500 loss : 0.005143631560155898
epoch : 9000 loss : 0.0044692554314716875
epoch : 9500 loss : 0.003926680222047044
```

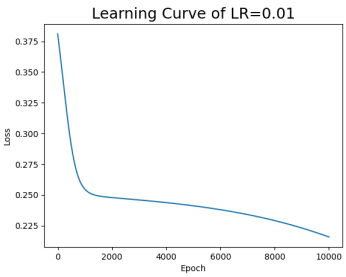
Discussion

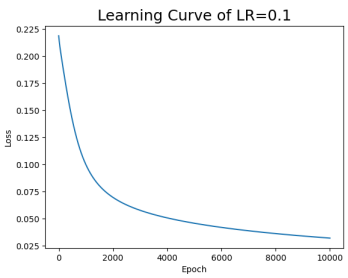
A. Try different learning rate

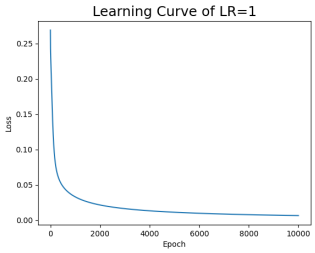
We can clearly observe that the speed of convergence is different, and the training process with a larger learning rate has a faster convergence speed. Learning rate is the parameter of gradient descent. $\theta^1 = \theta^0 - \eta \nabla L(\theta^0)$. It can control the size of each step. If it is too large, it may reach the very best point, and the next step will be over, so it will not converge. If it is too slow, it will iterate many times before it converges.

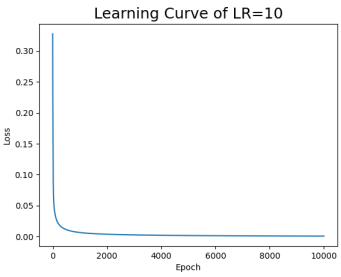








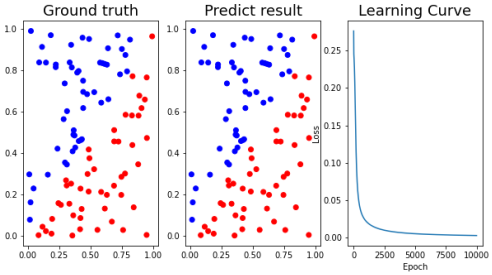




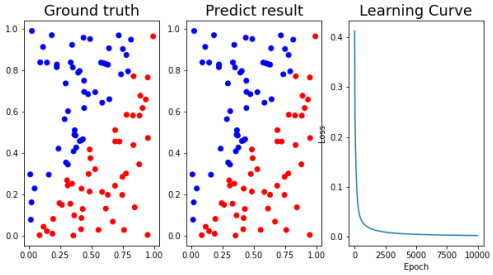
B. Try different numbers of hidden units

I think it is most clearly felt that the training time increases with the number of neurons due to $z = w^T x + b$, it means that the more neurons we create the more matrix computation we need to do. So I think the calculation time of forward will increase, but because the hidden design is too simple, and the precision of the timestamp is not enough, so I did not list the forward time

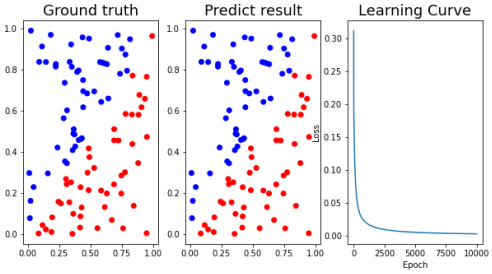
number of neurons : 4
training time : 0.5953 s



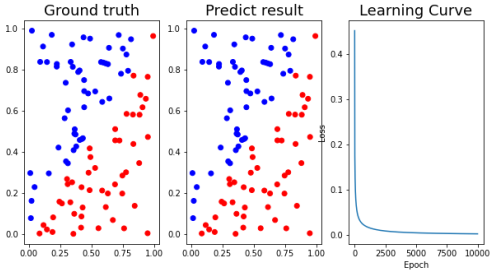
number of neurons : 8
training time : 0.6680 s



number of neurons : 16
training time : 0.7725 s



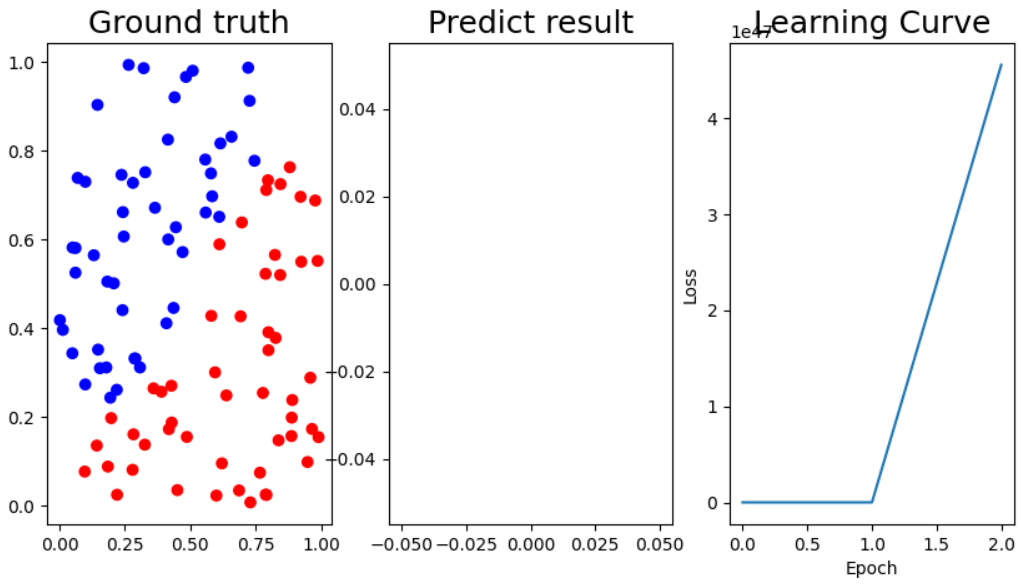
number of neurons : 64
training time : 3.7952 s



C. Try without activation functions

Remove the sigmoid function in every layer.

```
def forward(self, X):
    self.forward_gradient = X
    z = X@self.w
    self.a = z#sigmoid(z)
    return self.a
```



Runtime warning

```
e:\jeff\NCTU\DLP\lab01.py:65: RuntimeWarning: overflow encountered in square
  return np.mean((y - y_hat)**2)
e:\jeff\NCTU\DLP\lab01.py:62: RuntimeWarning: overflow encountered in multiply
  return np.multiply(x, 1.0 - x)
e:\jeff\NCTU\DLP\lab01.py:99: RuntimeWarning: invalid value encountered in matmul
  return np.matmul(self.backward_gradient, self.w.T)
e:\jeff\NCTU\DLP\lab01.py:90: RuntimeWarning: invalid value encountered in matmul
  z = X@self.w
```

```
epoch : 500 loss : nan
epoch : 1000 loss : nan
epoch : 1500 loss : nan
epoch : 2000 loss : nan
```

```
epoch : 2500 loss : nan
epoch : 3000 loss : nan
epoch : 3500 loss : nan
epoch : 4000 loss : nan
epoch : 4500 loss : nan
epoch : 5000 loss : nan
epoch : 5500 loss : nan
epoch : 6000 loss : nan
epoch : 6500 loss : nan
epoch : 7000 loss : nan
epoch : 7500 loss : nan
epoch : 8000 loss : nan
epoch : 8500 loss : nan
epoch : 9000 loss : nan
epoch : 9500 loss : nan
```

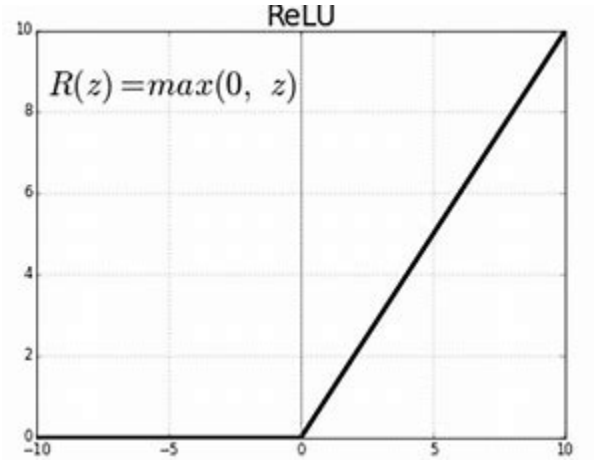
Extra

B. Implement different activation functions

ReLU

Not centered at 0, **there is no gradient when x<0, backpropagation will not update**, it will be dead.
For example: a very large gradient passes through a ReLU neuron. After updating the parameters, this neuron will no longer activate any data. If this happens, then all gradients flowing through this neuron will become 0 from now on.
That is, this ReLU unit will irreversibly die during training, resulting in a loss of data diversification. In practice, if the learning rate is set too high, it is possible to find that 40% of the neurons in the network will die (they will not fire in the entire training set). Reasonable setting of the learning rate will reduce the probability of this happening.

$R(z) = max(0, z)$

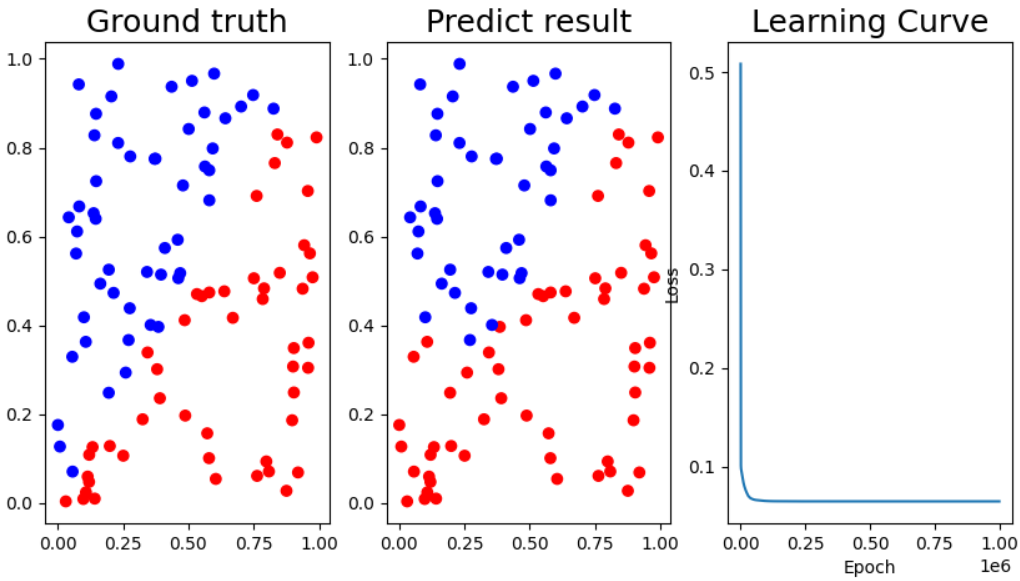


Implement

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

def derivative_ReLU(x):
    return 1.0 * (x > 0)
```

Result

LEARNING_RATE = 0.001



LEARNING_RATE = 1