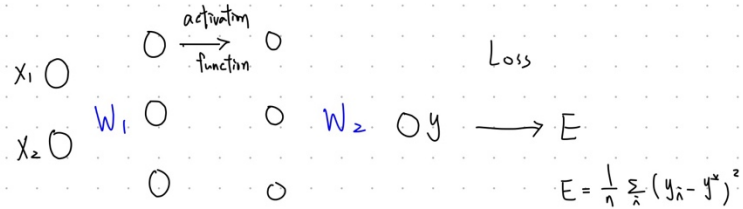


# 311551148 lab1 report

## 1. Introduction

這次 lab 要實作 linear dense layer from scratch，須要實作的有 dense layer, activation layer, loss function，使用 chain rule 找到所有需要的 gradient，以下是各個會用到的 gradient 的推導



① Dense layer: input  $X$ , output  $Y$ , weight  $W, B$

$$Y = WX + B$$

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_j \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1\hat{\lambda}} \\ w_{21} & & & \\ \vdots & & & \\ w_{j1} & & & \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{\hat{\lambda}} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_j \end{bmatrix}$$

$j \times 1 \quad j \times \hat{\lambda} \quad \hat{\lambda} \times 1 \quad j \times 1$

$$X \rightarrow \boxed{\phantom{Y}} \rightarrow Y \xrightarrow{\text{Loss}} E$$

Note:

$$\begin{cases} y_1 = x_1 w_{11} + x_2 w_{12} + \dots + x_{\hat{\lambda}} w_{1\hat{\lambda}} + b_1 \\ y_2 = x_1 w_{21} + x_2 w_{22} + \dots + x_{\hat{\lambda}} w_{2\hat{\lambda}} + b_2 \\ \vdots \\ y_j = x_1 w_{j1} + x_2 w_{j2} + \dots + x_{\hat{\lambda}} w_{j\hat{\lambda}} + b_j \end{cases}$$

$$\frac{\partial E}{\partial Y} = \begin{bmatrix} \frac{\partial E}{\partial y_1} \\ \frac{\partial E}{\partial y_2} \\ \vdots \\ \frac{\partial E}{\partial y_j} \end{bmatrix} \quad \frac{\partial E}{\partial W} = \begin{bmatrix} \frac{\partial E}{\partial w_{11}} & \frac{\partial E}{\partial w_{12}} & \dots & \frac{\partial E}{\partial w_{1\hat{\lambda}}} \\ \frac{\partial E}{\partial w_{21}} & & & \\ \vdots & & & \\ \frac{\partial E}{\partial w_{j1}} & & & \end{bmatrix}$$

$$\frac{\partial E}{\partial w_{12}} = \frac{\partial E}{\partial y_1} \frac{\partial y_1}{\partial w_{12}} + \frac{\partial E}{\partial y_2} \frac{\partial y_2}{\partial w_{12}} + \dots + \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial w_{12}} \Rightarrow \frac{\partial E}{\partial w_{12}} = \frac{\partial E}{\partial y_1} x_2 \Rightarrow \frac{\partial E}{\partial w_{j\hat{\lambda}}} = \frac{\partial E}{\partial y_j} x_{\hat{\lambda}}$$

$$\Rightarrow \frac{\partial E}{\partial W} = \begin{bmatrix} \frac{\partial E}{\partial y_1} x_1 & \frac{\partial E}{\partial y_2} x_2 & \dots & \frac{\partial E}{\partial y_j} x_{\hat{\lambda}} \\ \frac{\partial E}{\partial y_2} x_1 & & & \\ \vdots & & & \\ \frac{\partial E}{\partial y_j} x_1 & & & \end{bmatrix} = \begin{bmatrix} \frac{\partial E}{\partial y_1} \\ \frac{\partial E}{\partial y_2} \\ \vdots \\ \frac{\partial E}{\partial y_j} \end{bmatrix} \begin{bmatrix} x_1 & x_2 & \dots & x_{\hat{\lambda}} \end{bmatrix} = \frac{\partial E}{\partial Y} X^T$$

$j \times \hat{\lambda}$

$$\frac{\partial E}{\partial B} = \begin{bmatrix} \frac{\partial E}{\partial b_1} \\ \frac{\partial E}{\partial b_2} \\ \vdots \\ \frac{\partial E}{\partial b_j} \end{bmatrix} \quad \frac{\partial E}{\partial b_1} = \frac{\partial E}{\partial y_1} \frac{\partial y_1}{\partial b_1} + \frac{\partial E}{\partial y_2} \frac{\partial y_2}{\partial b_1} + \dots + \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial b_1}$$

$$= \frac{\partial E}{\partial y_1} \cdot 1 \Rightarrow \frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial y_j}$$

$$\frac{\partial E}{\partial B} = \frac{\partial E}{\partial Y}$$

$$\frac{\partial E}{\partial X} = \begin{bmatrix} \frac{\partial E}{\partial x_1} \\ \frac{\partial E}{\partial x_2} \\ \vdots \\ \frac{\partial E}{\partial x_i} \end{bmatrix}$$

$$\frac{\partial E}{\partial x_1} = \frac{\partial E}{\partial y_1} \frac{\partial y_1}{\partial x_1} + \frac{\partial E}{\partial y_2} \frac{\partial y_2}{\partial x_1} + \dots + \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial x_1}$$

$$= \frac{\partial E}{\partial y_1} w_{11} + \frac{\partial E}{\partial y_2} w_{21} + \dots + \frac{\partial E}{\partial y_j} w_{j1}$$

$$\frac{\partial E}{\partial x_i} = \sum_{k=1}^J \frac{\partial E}{\partial y_k} w_{ki}$$

$$\frac{\partial E}{\partial X} = \begin{bmatrix} \frac{\partial E}{\partial y_1} w_{11} + \frac{\partial E}{\partial y_2} w_{21} + \dots + \frac{\partial E}{\partial y_j} w_{j1} \\ \frac{\partial E}{\partial y_1} w_{12} + \frac{\partial E}{\partial y_2} w_{22} + \dots + \frac{\partial E}{\partial y_j} w_{j2} \\ \vdots \\ \frac{\partial E}{\partial y_1} w_{1i} + \frac{\partial E}{\partial y_2} w_{2i} + \dots + \frac{\partial E}{\partial y_j} w_{ji} \end{bmatrix} = \underset{1 \times J}{W^T} \underset{J \times 1}{\frac{\partial E}{\partial Y}}$$

last layer output :  $Y$

$$E = \frac{1}{n} \sum_i (y_i^* - y_i)^2$$

$$\frac{\partial E}{\partial Y} = \begin{bmatrix} \frac{\partial E}{\partial y_1} \\ \frac{\partial E}{\partial y_2} \\ \vdots \\ \frac{\partial E}{\partial y_j} \end{bmatrix}$$

$$\frac{\partial E}{\partial y_1} = \frac{\partial}{\partial y_1} \frac{1}{n} \left[ (y_1^* - y_1)^2 + (y_2^* - y_2)^2 + \dots + (y_n^* - y_n)^2 \right]$$

$$= \frac{2}{n} (y_1 - y_1^*)$$

$$\frac{\partial E}{\partial Y} = \frac{2}{n} (Y - Y^*)$$

甲、Dense layer:

input :  $X$ , output :  $Y$ , weight :  $W$ , bias :  $B$

需要計算  $dE/dW$  更新 weight

需要計算  $dE/dX$ ，backprog 時傳回去更上一層的 layer 當作他的  $dE/dY$

乙、Activation lay

$$Y = f(X)$$

$$dE/dX = dE/dY * f'(X)$$

所以除了要實作 forward 的 activation function，也要實作他的 inverse function

丙、Loss function

使用 l2 norm loss function

需要計算  $dE/dY$  當作最後一層的 layer 的 backward 的 input

## 2. Experiment setup

### 甲、Sigmoid function

```
def sigmoid(x):  
    # print(x)  
    return 1.0 / (1.0 + np.exp(-x))  
  
def sigmoid_prime(x):  
    return 1 / (1 + np.exp(-x)) * (1 - (1 / (1 + np.exp(-x))))
```

使用 numpy 實作 sigmoid function 以及他的反函數

```
# inherit from base class Layer  
class ActivationLayer(Layer):  
    def __init__(self, activation, activation_prime):  
        self.activation = activation  
        self.activation_prime = activation_prime  
  
    # returns the activated input  
    def forward_propagation(self, input_data):  
        self.input = input_data  
        self.output = self.activation(self.input)  
        return self.output  
  
    # Returns input_error=dE/dX for a given output_error=dE/dY.  
    # learning_rate is not used because there is no "learnable" parameters.  
    def backward_propagation(self, output_error, learning_rate):  
        return self.activation_prime(self.input) * output_error
```

activation layer 可以直接使用前面實作的 sigmoid function，forward 時就很單純 return  $f(X)$ ，backward 時，根據前面推導的公式，要 return  $dE/dY$  (element-wise mul)  $f'(X)$ ，這邊的 output\_error 就是下一層 layer 回傳的  $dE/dY$

### 乙、Neural Network

因為各種 layer 都有 input output，用一個 base class Layer 裡面有 input, output，dense layer 的部分，一開始 init 時先 random 出 weight  $W$  and bias  $B$  的參數，參考前面推導出來的公式，forward 會是  $Y = WX + B$ ，backward 時要更新 weight and bias 以及計算  $dE/dX$  回傳當作上一層 layer 的  $dE/dY$ ，根據前面推導的公式

$$dE/dW = dE/dY * X^T$$

$$dE/dB = dE/dY$$

$$dE/dX = X^T * dE/dY \text{ 則}$$

$$\text{weight} = \text{weight} - \text{learning\_rate} * dE/dY * X^T$$

$$\text{bias} = \text{bias} - \text{learning\_rate} * dE/dY$$

```

# Base class
class Layer:
    def __init__(self):
        self.input = None
        self.output = None

    # computes the output Y of a layer for a given input X
    def forward_propagation(self, input):
        raise NotImplementedError

    # computes dE/dX for a given dE/dY (and update parameters if any)
    def backward_propagation(self, output_error, learning_rate):
        raise NotImplementedError

# inherit from base class Layer
class FCLayer(Layer):
    # input_size = number of input neurons
    # output_size = number of output neurons
    def __init__(self, input_size, output_size):
        self.weights = np.random.rand(output_size, input_size) #- 0.5
        self.bias = np.random.rand(output_size, 1) #- 0.5

    # returns output for a given input
    def forward_propagation(self, input_data):
        self.input = input_data
        self.output = np.dot(self.weights, self.input) + self.bias
        return self.output

    # computes dE/dW, dE/dB for a given output_error=dE/dY. Returns input_error=dE/dX.
    def backward_propagation(self, output_error, learning_rate):
        input_error = np.dot(self.weights.T, output_error)
        weights_error = np.dot(output_error, self.input.T)
        bias_error = output_error

        # update parameters
        self.weights -= learning_rate * weights_error
        self.bias -= learning_rate * bias_error
        return input_error

```

### 丙、Backpropagation

```

# backward propagation
error = self.loss_prime(y_train[j], output)
for layer in reversed(self.layers):
    error = layer.backward_propagation(error, learning_rate)

```

forward 到底之後利用 mse 的反函數計算出  $dE/dY$ ，把這個  $dE/dY$  傳給前一層 layer，直到最源頭

### 3. Result of your testing

#### 甲、Screenshot and comparison figure

##### i. Training loss

linear data

```
epoch 1000/10000    loss=0.010938227540760815
epoch 2000/10000    loss=0.00718020784667072
epoch 3000/10000    loss=0.0003799850697036738
epoch 4000/10000    loss=0.00015534414276479212
epoch 5000/10000    loss=9.069745237781075e-05
epoch 6000/10000    loss=6.19740777280378e-05
epoch 7000/10000    loss=4.6330982974575995e-05
epoch 8000/10000    loss=3.671077551499476e-05
epoch 9000/10000    loss=3.0291218492234446e-05
epoch 10000/10000   loss=2.5746812919321387e-05
```

XOR data

```
epoch 1000/10000    loss=0.25326492004948625
epoch 2000/10000    loss=0.2304331534884977
epoch 3000/10000    loss=0.037987205180578534
epoch 4000/10000    loss=0.002346365473827735
epoch 5000/10000    loss=0.0007180857821771613
epoch 6000/10000    loss=0.00038492396337189274
epoch 7000/10000    loss=0.0002530210513878264
epoch 8000/10000    loss=0.0001846918202349471
epoch 9000/10000    loss=0.00014366165485306097
epoch 10000/10000   loss=0.00011660435610487455
```

- ii. Prediction and accuracy (with train test split)

linear data (acc = 95%)

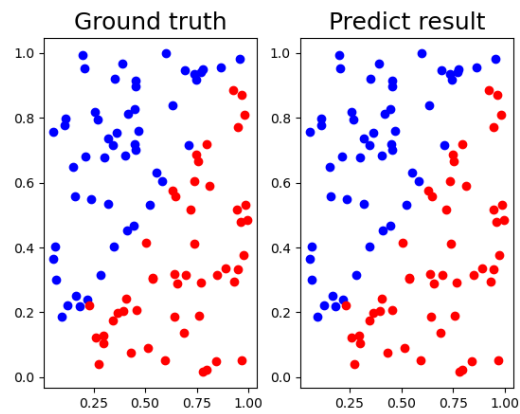
```
Iter 0 | Ground truth: 0 | prediction: 0.00113 |
Iter 1 | Ground truth: 0 | prediction: 0.00118 |
Iter 2 | Ground truth: 0 | prediction: 0.00114 |
Iter 3 | Ground truth: 0 | prediction: 0.00118 |
Iter 4 | Ground truth: 1 | prediction: 0.99995 |
Iter 5 | Ground truth: 0 | prediction: 0.00782 |
Iter 6 | Ground truth: 0 | prediction: 0.00162 |
Iter 7 | Ground truth: 1 | prediction: 0.40745 |
Iter 8 | Ground truth: 0 | prediction: 0.00113 |
Iter 9 | Ground truth: 0 | prediction: 0.00118 |
Iter 10 | Ground truth: 1 | prediction: 0.99995 |
Iter 11 | Ground truth: 1 | prediction: 0.99995 |
Iter 12 | Ground truth: 1 | prediction: 0.99995 |
Iter 13 | Ground truth: 0 | prediction: 0.00135 |
Iter 14 | Ground truth: 0 | prediction: 0.00116 |
Iter 15 | Ground truth: 1 | prediction: 0.99823 |
Iter 16 | Ground truth: 0 | prediction: 0.00113 |
Iter 17 | Ground truth: 1 | prediction: 0.99993 |
Iter 18 | Ground truth: 1 | prediction: 0.99995 |
Iter 19 | Ground truth: 1 | prediction: 0.99994 |
loss= 0.01756 accuracy=95.0%
```

XOR data (acc = 100%)

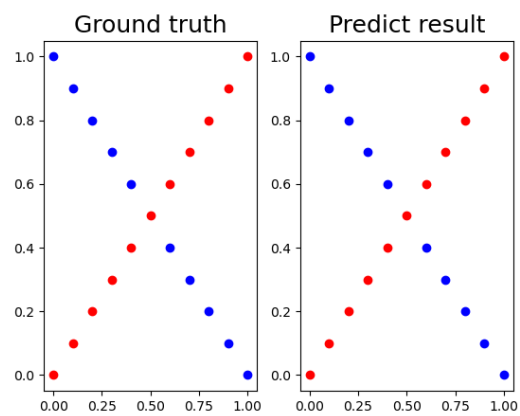
```
Iter 0 | Ground truth: 1 | prediction: 0.99445 |
Iter 1 | Ground truth: 0 | prediction: 0.01048 |
Iter 2 | Ground truth: 1 | prediction: 0.99466 |
Iter 3 | Ground truth: 0 | prediction: 0.01045 |
Iter 4 | Ground truth: 1 | prediction: 0.99473 |
loss= 0.00006 accuracy=100.0%
```

- iii. Comparison figure

linear data

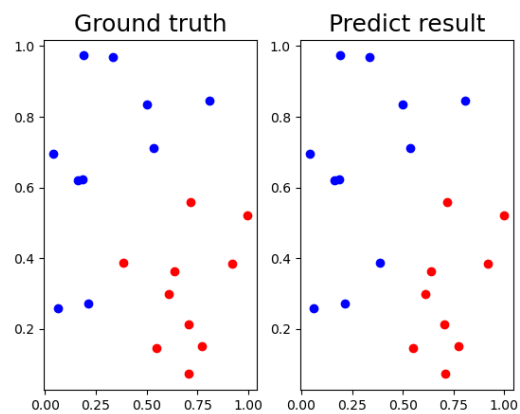


XOR data

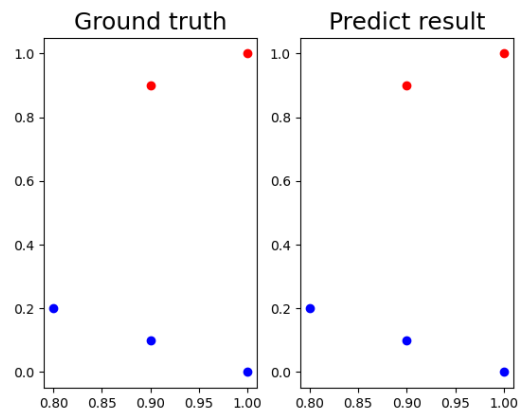


乙、 Show the accuracy of your prediction

- i. Without train test spilt  
linear and XOR data accuracy = 100%
- ii. train test spilt with spilt ratio = 0.8  
linear data acc = 95%

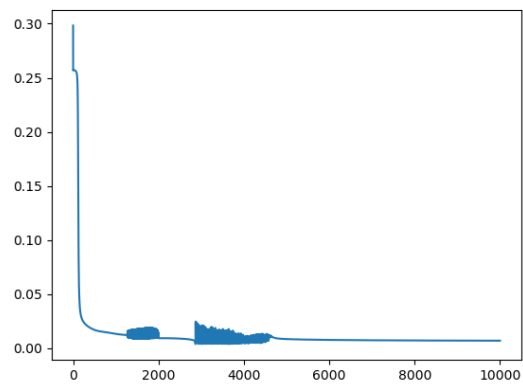


XOR data acc = 100%

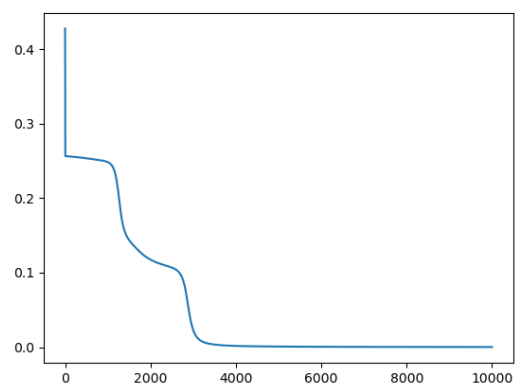


### 丙、Learning curve

#### i. Linear data



#### ii. XOR data



### 丁、Anything you want to share

#### i. Simple train test spilt

```
def train_test_spilt(x, y, train_ratio):
    return x[0:int(x.shape[0]*train_ratio)]:, y[0:int(x.shape[0]*train_ratio)]:, x[int(x.shape[0]*train_ratio):]:, y[int(x.shape[0]*train_ratio):]:
```

依照 train ratio 把資料分成 x\_train, y\_train, x\_test, y\_test，訓練時  
只有使用 training data，testing 時只使用 test data

## ii. Network class

實作一個 network class，因為 layer 都很單純的一層接一層，沒有類似 residual connection 的問題，所以使用一個 list 來紀錄 layer 的順序，增加 layer 就是單純的 append to the list，forward and backward 也都是依據這個 list 的順序去計算，實作 fit function 訓練 layer weight，實作 predict function 用來做 testing，也就是單純的 forward。

```
class Network:
    def __init__(self):
        self.layers = []
        self.loss = None
        self.loss_prime = None

    # add layer to network
    def add(self, layer):
        self.layers.append(layer)

    # set loss to use
    def use(self, loss, loss_prime):
        self.loss = loss
        self.loss_prime = loss_prime

    # predict output for given input
    def predict(self, input_data):
        # sample dimension first
        samples = len(input_data)
        result = []

        # run network over all samples
        for i in range(samples):
            # forward propagation
            output = input_data[i].T
            for layer in self.layers:
                output = layer.forward_propagation(output)
            result.append(list(output[0]))

        return result

    # train the network
    def fit(self, x_train, y_train, epochs, learning_rate):
        # sample dimension first
        samples = len(x_train)
        loss_record = []
        # training loop
        for i in range(epochs):
            err = 0
            for j in range(samples):
                # forward propagation
                output = x_train[j].T
                for layer in self.layers:
                    output = layer.forward_propagation(output)

                # compute loss (for display purpose only)
                err += self.loss(y_train[j], output)

                # backward propagation
                error = self.loss_prime(y_train[j], output)
                for layer in reversed(self.layers):
                    error = layer.backward_propagation(error, learning_rate)

            # calculate average error on all samples
            err = err / samples
            loss_record.append(err)
            if (i+1) % 1000 == 0:
                print(f'epoch {i+1}/{epochs}    loss={err}')
        return loss_record
```



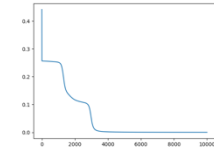
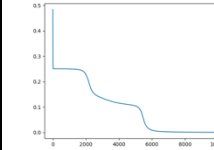
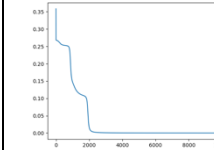
```
# network
net = Network()
net.add(FCLayer(2, 5))
net.add(ActivationLayer(sigmoid, sigmoid_prime))
net.add(FCLayer(5, 3))
net.add(ActivationLayer(sigmoid, sigmoid_prime))
net.add(FCLayer(3, 1))
net.add(ActivationLayer(sigmoid, sigmoid_prime))

# train
net.use(mse, mse_prime)
loss_record = net.fit(np.expand_dims(x_train, axis=1), np.expand_dims(y_train, axis=1), epochs=10000, learning_rate=0.1)
plot_loss(loss_record)
# test
out = net.predict(np.expand_dims(x_test, axis=1))
```

#### 4. Discussion

##### 甲、Try different learning rate

觀察 learning rate 與 loss curve 的關係，learning rate 越高會越快收斂

Learning rate	0.1	0.05	0.2
Loss curve			

##### 乙、Try different number of hidden units

觀察 hidden unit 多寡對訓練的影響，要是太多會不會造成 overfitting (layer input dimension, layer out dimension)

Layer parameter	(2,5)(5,3)(3,1)	(2,10)(10,5)(5,1)	(2,5)(5,10)(10,5)(5,3)(3,1)
Testing acc(with train test spilt)	100%	100%	40%

訓練設定都是使用 XOR data with train 80% test 20%，訓練 10000 epoch with learning rate = 0.2，發現當網路深度太深，同樣的訓練次數下，testing acc 大幅下降，觀察 training loss，發現其實是有在收斂的，但是速度奇慢無比，我認為是因為網路深度太深，有可能會造成類似 gradient vanish 的問題，導致模型無法正常收斂。

##### 丙、Try without activation function

少了 activation function，不只是無法解決 XOR problem，在 forward 時就很有可能會遇到 overflow 的問題，因為 forward 過程中有大量的乘法加法運算，要是沒有 activation function 把數值控制在 0~1 之間，一直連乘下去非常有可能會造成 overflow，程式直接 crash 掉

```
[[1.03148704]]
[[-9.20044659]]
[[3622.73502759]]
[[-4.53424071e+16]]
[[1.47191981e+82]]
[[-inf]]
/eva_data2/shlu2240/NCTU_DLP/lab1/layer.py:31: RuntimeWarning: invalid value encountered in add
  self.output = np.dot(self.weights, self.input) + self.bias
[[nan]]
[[nan]]
```

為了解決在算 mse 時會 overflow 的問題，在最後要算 mse 前放一層 activate layer，把 output 控制在 0~1 之間，但模型的中間都不要使用 activate layer，發現他無法解決 XOR 的問題，prediction accuracy 只有 40%。

```
# network
net = Network()
net.add(FCLayer(2, 4, 'sgd'))
# net.add(ActivationLayer(sigmoid, sigmoid_prime))
net.add(FCLayer(4, 4, 'sgd'))
# net.add(ActivationLayer(sigmoid, sigmoid_prime))
net.add(FCLayer(4, 1, 'sgd'))
net.add(ActivationLayer(sigmoid, sigmoid_prime))
```

```
Iter 0 | Ground truth: 1 | prediction: 0.00000 |
Iter 1 | Ground truth: 0 | prediction: 0.00161 |
Iter 2 | Ground truth: 1 | prediction: 0.00000 |
Iter 3 | Ground truth: 0 | prediction: 0.00173 |
Iter 4 | Ground truth: 1 | prediction: 0.00000 |
loss= 0.60000 accuracy=40.0%
```

#### 丁、Anything you want to share

有嘗試依據 epoch 動態變更 learning rate，輸入的 learning rate 從原本的 scalar 改成一個 list，裡面是從大到小的 learning rate

```
net.fit(np.expand_dims(x_train, axis=1), np.expand_dims(y_train, axis=1), epochs=10000, learning_rate=[0.5, 0.2, 0.1])

# backward propagation
error = self.loss_prime(y_train[j], output)
for layer in reversed(self.layers):
    error = layer.backward_propagation(error, learning_rate[int(i/(int(epochs/len(learning_rate))+1))])
```

learning\_rate[int(i/(int(epochs/len(learning\_rate))+1))] 可以把 epoch 平均分給從大到小各個 learning rate。

#### 5. Extra

##### 甲、Implement different optimizers

除了最原始的 sgd optimizer，還有實作 momentum optimizer，帶入的公式為下圖，基本上就是把上一個 delta 存下來，乘上 alpha 加上 learning rate \* gradient 當作這次學習的 delta

$$\Delta W_1^N = -\eta * \frac{\partial Error}{\partial W_1} + \alpha * \Delta W_1^{N-1}$$

$$W_1 += \Delta W_1^N$$

```

if self.optimizer=='momentum':
    # update parameters
    delta = learning_rate * weights_error + self.alpha * self.delta_prev
    self.delta_prev = delta
    self.weights -= delta
    self.bias -= learning_rate * bias_error

```

## 乙、Implement different activation function

有實作 tanh 以及他的反函數

```

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

def tanh_prime(x):
    return 1 - np.tanh(x)**2

```

```

net = Network()
net.add(FCLayer(2, 5, 'momentum'))
net.add(ActivationLayer(tanh, tanh_prime))
net.add(FCLayer(5, 3, 'momentum'))
net.add(ActivationLayer(tanh, tanh_prime))
net.add(FCLayer(3, 1, 'momentum'))
net.add(ActivationLayer(tanh, tanh_prime))

```

也有實作 ReLU 以及他的反函數，但因為他函數的性質，在使用的時候最後一層的 activate function 是使用 sigmoid 來把 output 控制在 0~1 之間

```

def ReLU(x):
    return x * (x > 0)

def ReLU_prime(x):
    return 1. * (x > 0)

```

```

# network
net = Network()
net.add(FCLayer(2, 4, 'sgd'))
net.add(ActivationLayer(ReLU, ReLU_prime))
net.add(FCLayer(4, 4, 'sgd'))
net.add(ActivationLayer(ReLU, ReLU_prime))
net.add(FCLayer(4, 1, 'sgd'))
net.add(ActivationLayer(sigmoid, sigmoid_prime))

```

## 丙、Implement convolutional layers

假設 input 的 channel 數=1，kernel 的數量也=1

給定  $dE/dY$  去計算  $dE/dK$ ,  $dE/dB$ ,  $dE/dX$  分別會是多少，在 implement 的時候 convolution operation call scipy.signal library，作用上只有單純計算 convolution operation， $dE/dK$ ,  $dE/dB$ ,  $dE/dX$  的計算是依照推導出來的公式去計算的，下面附上推導的公式

# Convolutional layer

let input depth = 1, number of kernel = 1

$$X = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}$$

$$K = \begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix}$$

$$B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$Y \text{ (valid)} = \begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{bmatrix}$$

forward:

$$Y = B + X * K \quad \text{cross correlation}$$

$$\text{展開} \Rightarrow \begin{cases} y_{11} = b_{11} + x_{11} \cdot k_{11} + x_{12} \cdot k_{12} + x_{21} \cdot k_{21} + x_{22} \cdot k_{22} \\ y_{12} = b_{12} + x_{12} \cdot k_{11} + x_{13} \cdot k_{12} + x_{22} \cdot k_{21} + x_{23} \cdot k_{22} \\ \vdots \end{cases}$$

$$Y \text{ (full)} = \begin{bmatrix} y_{11} & y_{12} & y_{13} & y_{14} \\ y_{21} & y_{22} & y_{23} & y_{24} \\ y_{31} & y_{32} & y_{33} & y_{34} \\ y_{41} & y_{42} & y_{43} & y_{44} \end{bmatrix}$$

backward:  $\frac{dE}{dK}, \frac{dE}{dB}, \frac{dE}{dX}$  given  $\frac{dE}{dY}$

$$\textcircled{1} \quad \frac{dE}{dK} = \begin{bmatrix} \frac{dE}{dk_{11}} & \frac{dE}{dk_{12}} \\ \frac{dE}{dk_{21}} & \frac{dE}{dk_{22}} \end{bmatrix}$$

$$\frac{dE}{dk_{11}} = \frac{dE}{dy_{11}} \frac{dy_{11}}{dk_{11}} + \frac{dE}{dy_{12}} \frac{dy_{12}}{dk_{11}} + \frac{dE}{dy_{21}} \frac{dy_{21}}{dk_{11}} + \frac{dE}{dy_{22}} \frac{dy_{22}}{dk_{11}}$$

$$= \frac{dE}{dy_{11}} x_{11} + \frac{dE}{dy_{12}} x_{12} + \frac{dE}{dy_{21}} x_{21} + \frac{dE}{dy_{22}} x_{22}$$

$$\frac{dE}{dk_{12}} = \frac{dE}{dy_{11}} x_{12} + \frac{dE}{dy_{12}} x_{13} + \frac{dE}{dy_{21}} x_{22} + \frac{dE}{dy_{22}} x_{23}$$

$$\frac{dE}{dk_{21}} = \frac{dE}{dy_{11}} x_{21} + \frac{dE}{dy_{12}} x_{22} + \frac{dE}{dy_{21}} x_{31} + \frac{dE}{dy_{22}} x_{32}$$

$$\frac{dE}{dk_{22}} = \frac{dE}{dy_{11}} x_{22} + \frac{dE}{dy_{12}} x_{23} + \frac{dE}{dy_{21}} x_{32} + \frac{dE}{dy_{22}} x_{33}$$

$$\Rightarrow \frac{dE}{dK} = X * \frac{dE}{dY}$$

$$\textcircled{2} \quad \frac{dE}{dB} = \begin{bmatrix} \frac{dE}{db_{11}} & \frac{dE}{db_{12}} \\ \frac{dE}{db_{21}} & \frac{dE}{db_{22}} \end{bmatrix} = \frac{dE}{dY}$$

$$\frac{dE}{db_{11}} = \frac{dE}{dy_{11}}, \quad \frac{dE}{db_{12}} = \frac{dE}{dy_{12}}, \quad \frac{dE}{db_{21}} = \frac{dE}{dy_{21}}, \quad \frac{dE}{db_{22}} = \frac{dE}{dy_{22}}$$

$$\textcircled{3} \quad \frac{dE}{dX} = \begin{bmatrix} \frac{dE}{dx_{11}} & \frac{dE}{dx_{12}} & \frac{dE}{dx_{13}} \\ \frac{dE}{dx_{21}} & \frac{dE}{dx_{22}} & \frac{dE}{dx_{23}} \\ \frac{dE}{dx_{31}} & \frac{dE}{dx_{32}} & \frac{dE}{dx_{33}} \end{bmatrix}$$

$$\frac{dE}{dx_{11}} = \frac{dE}{dy_{11}} \frac{dy_{11}}{dx_{11}} + \frac{dE}{dy_{12}} \frac{dy_{12}}{dx_{11}} + \dots = \frac{dE}{dy_{11}} k_{11}$$

$$\begin{aligned}
 \frac{dE}{dX_{12}} &= \frac{dE}{dy_{11}} \frac{dy_{11}}{dX_{12}} + \frac{dE}{dy_{12}} \frac{dy_{12}}{dX_{12}} + \dots = 0 \quad \left( \begin{matrix} \text{''} \\ k_{12} \end{matrix} \right) \quad \left( \begin{matrix} \text{''} \\ k_{11} \end{matrix} \right) \quad \left( \begin{matrix} \text{''} \\ k_{12} \end{matrix} \right) + \frac{dE}{dy_{12}} k_{11} \\
 \frac{dE}{dX_{13}} &= \frac{dE}{dy_{12}} k_{12} \\
 \frac{dE}{dX_{21}} &= \frac{dE}{dy_{11}} k_{21} + \frac{dE}{dy_{21}} k_{11} \\
 \frac{dE}{dX_{22}} &= \frac{dE}{dy_{11}} k_{21} + \frac{dE}{dy_{12}} k_{21} + \frac{dE}{dy_{21}} k_{12} + \frac{dE}{dy_{22}} k_{11} \\
 \frac{dE}{dX_{23}} &= \frac{dE}{dy_{12}} k_{22} + \frac{dE}{dy_{22}} k_{12} \\
 \frac{dE}{dX_{31}} &= \frac{dE}{dy_{21}} k_{21} \\
 \frac{dE}{dX_{32}} &= \frac{dE}{dy_{21}} k_{22} + \frac{dE}{dy_{22}} k_{21} \\
 \frac{dE}{dX_{33}} &= \frac{dE}{dy_{22}} k_{22} \\
 \Rightarrow \frac{dE}{dX} &= \frac{dE}{dY} \star_{full} \text{rot}_{180}(K) \\
 &= \frac{dE}{dY} \star_{full}^{\text{convolve}} K
 \end{aligned}$$

如果 input depth=1 and # of kernel=1，那公式會變得簡單很多

$dE/dK = X \text{ (valid correlation) } dE/dY$

$dE/dB = dE/dY$

$dE/dX = dE/dY \text{ (full convolution) } K$

```

class Convolutional(Layer):
    def __init__(self, input_shape, kernel_size): # output shape = (input_height - kernel_size + 1, input_width - kernel_size + 1)
        input_height, input_width = input_shape # w * h
        self.kernels = np.random.randn(kernel_size, kernel_size)
        self.biases = np.random.randn(input_height - kernel_size + 1, input_width - kernel_size + 1)

    def forward_propagation(self, input):
        self.input = input
        self.output = np.copy(self.biases)
        self.output += signal.correlate2d(self.input, self.kernels, "valid")
        return self.output

    def backward_propagation(self, output_gradient, learning_rate):
        kernels_gradient = np.zeros(self.kernels_shape)
        input_gradient = np.zeros(self.input_shape)

        kernels_gradient = signal.correlate2d(self.input, output_gradient, "valid")
        input_gradient += signal.convolve2d(output_gradient, self.kernels, "full")

        self.kernels -= learning_rate * kernels_gradient
        self.biases -= learning_rate * output_gradient
        return input_gradient

```

bias shape = output shape = (input height – kernel size + 1, input width – kernel size + 1)

kernel shape = kernel size \* kernel size

那因為 convolution 是做在 2d 的資料上，但是 dense layer 只會 output

1d 的資料，所以在實作一個 reshape layer 來把 1d 的資料 reshape 成 2d 的，reshape layer 的 forward, backward 就很單純，就是直接 reshape 成想要的 w\*h 就好。

```
class Reshape(Layer):
    def __init__(self, input_shape, output_shape):
        self.input_shape = input_shape
        self.output_shape = output_shape

    def forward_propagation(self, input):
        return np.reshape(input, self.output_shape)

    def backward_propagation(self, output_gradient, learning_rate):
        return np.reshape(output_gradient, self.input_shape)
```

要在模型中使用 convolution, reshape 也很簡單，就直接 add layer 就好，不過要注意的是 output shape 要先自己計算好，因為兩層相鄰的 layer 的維度要對應到不然會出錯。

```
# network
net = Network()
net.add(FCLayer(2, 9, 'sgd'))
net.add(ActivationLayer(sigmoid, sigmoid_prime))
#add convolutional layer with reshape layer
net.add(Reshape((9,1), (3,3)))
net.add(Convolutional((3,3), 2)) # output 2 * 2
net.add(Reshape((2,2), (4,1)))
# net.add(FCLayer(4, 4, 'sgd'))
net.add(ActivationLayer(sigmoid, sigmoid_prime))
net.add(FCLayer(4, 1, 'sgd'))
net.add(ActivationLayer(sigmoid, sigmoid_prime))
```

result

```
epoch 1000/10000    loss=0.11075497959248756
epoch 2000/10000    loss=0.0034141110656873567
epoch 3000/10000    loss=0.0006722718718597601
epoch 4000/10000    loss=0.0003592604459355404
epoch 5000/10000    loss=0.00024263720268396437
epoch 6000/10000    loss=0.0001823553740900765
epoch 7000/10000    loss=0.0001457126762861189
epoch 8000/10000    loss=0.00012115288770234527
epoch 9000/10000    loss=0.00010357673138465645
epoch 10000/10000   loss=9.039224541774561e-05
Iter   0 |      Ground truth: 1 |      prediction: 0.99738 |
Iter   1 |      Ground truth: 0 |      prediction: 0.01034 |
Iter   2 |      Ground truth: 1 |      prediction: 0.99742 |
Iter   3 |      Ground truth: 0 |      prediction: 0.01082 |
Iter   4 |      Ground truth: 1 |      prediction: 0.99743 |
loss= 0.00005 accuracy=100.0%
```