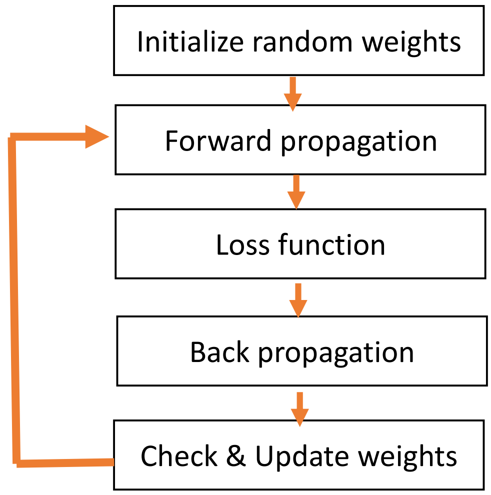
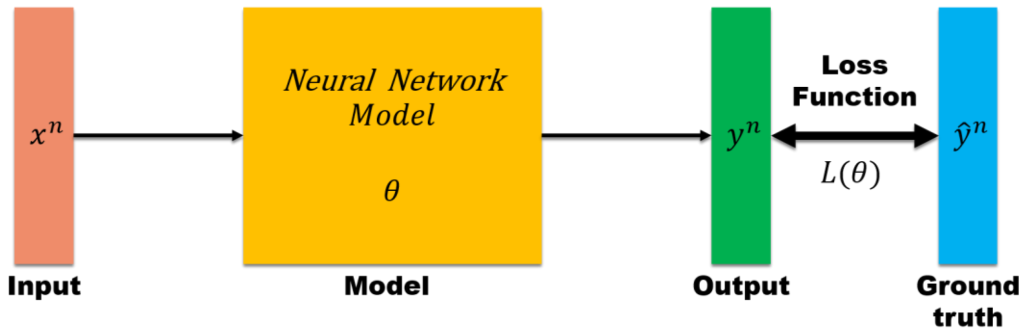
# Lab1 : back-propagation

1. Introduction (20%)




$$\theta = \{w_1, w_2, w_3, w_4, \cdots\}$$

$$\nabla \boldsymbol{L}(\boldsymbol{\theta}) = \begin{bmatrix} \partial L(\theta)/\partial w_1 \\ \partial L(\theta)/\partial w_2 \\ \partial L(\theta)/\partial w_3 \\ \vdots \\ \vdots \end{bmatrix}$$

Compute $\nabla L(\theta^0)$ $\quad \theta^1 = \theta^0 - \rho \nabla L(\theta^0)$

Compute $\nabla L(\theta^1)$ $\quad \theta^2 = \theta^0 - \rho \nabla L(\theta^1)$

Compute $\nabla L(\theta^2)$ $\quad \theta^3 = \theta^0 - \rho \nabla L(\theta^2)$

$\rho$ : Learning rate

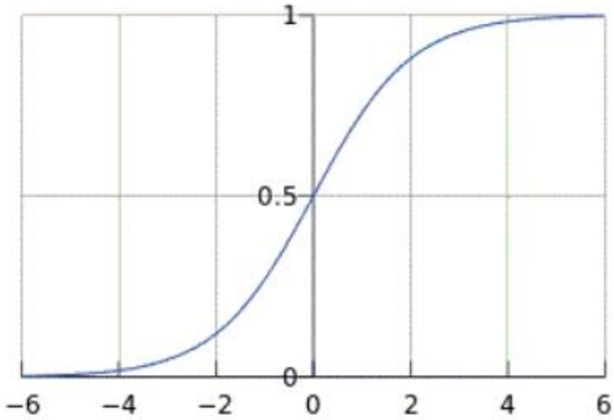
**Input:** generate from ***generate_linear()*** function or ***generate_XOR_easy*()** function

**Model:** simple neuron network with two hidden layers contains forward and backward propagation

**Loss function:** we use MSE to calculate the error between output and ground truth in this homework.

2. Experiment setups (30%):

    A. Sigmoid functions




```python
def __init__(self, input_size, hidden1_size, hidden2_size, output_size, lr):
    self.inodes = input_size
    self.h1nodes = hidden1_size
    self.h2nodes = hidden2_size
    self.onodes = output_size
    self.wih = np.random.randn(self.inodes, self.h1nodes)
    self.whh = np.random.randn(self.h1nodes, self.h2nodes)
    self.who = np.random.randn(self.h2nodes, self.onodes)
    self.lr = lr
    self.sigmoid = lambda x: 1/(1+np.exp(-x))
```

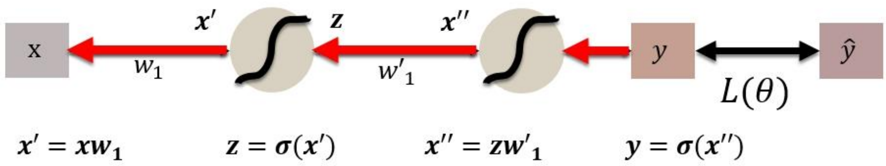
B. Neural network

Class Net() need the parameter of input_size, hidden_size, output_size and learning rate. For the ***__init__() function*** we will random generate the weight matrix of network layers and save the variable of learning rate. The ***train()*** function will run the forward propagation and ***sigmoid*** function to get the ***final_outputs***.

```python
class Net():
    def __init__(self, input_size, hidden1_size, hidden2_size, output_size, lr):
        self.inodes = input_size
        self.h1nodes = hidden1_size
        self.h2nodes = hidden2_size
        self.onodes = output_size
        self.wih = np.random.randn(self.inodes, self.h1nodes)
        self.whh = np.random.randn(self.h1nodes, self.h2nodes)
        self.who = np.random.randn(self.h2nodes, self.onodes)
        self.lr = lr
        self.sigmoid = lambda x: 1/(1+np.exp(-x))

    def train(self, inputs, targets):
        for epoch in range(1, 1001):
            hidden_inputs = np.dot(inputs, self.wih)
            hidden_outputs = self.sigmoid(hidden_inputs)

            hidden2_inputs = np.dot(hidden_outputs, self.whh)
            hidden2_outputs = self.sigmoid(hidden2_inputs)

            final_inputs = np.dot(hidden2_outputs, self.who)
            final_outputs = self.sigmoid(final_inputs)

            error = (targets - final_outputs) ** 2
            derr = -2 * (targets - final_outputs)
            hidden2_error = np.dot(derr, self.who.T)
            hidden_error = np.dot(hidden2_error, self.whh.T)

            self.who -= self.lr * np.dot(hidden2_outputs.T, (derr * final_outputs * (1.0 - final_outputs)))
            self.whh -= self.lr * np.dot(hidden_outputs.T, (hidden2_error * hidden2_outputs * (1.0 - hidden2_outputs)))
            self.wih -= self.lr * np.dot(inputs.T, (hidden_error * hidden_outputs * (1.0 - hidden_outputs)))

            if epoch % 100 == 0:
                acc = len(np.where(final_outputs.round(0) == targets)[0]) / len(final_outputs)
                print('epoch {:4d} loss : {:13s} Acc : {:3.2f}'.format(epoch, str(np.sum(error, axis=0)), acc))

    def query(self, inputs):
        hidden_inputs = np.dot(inputs, self.wih)
        hidden_outputs = self.sigmoid(hidden_inputs)

        hidden2_inputs = np.dot(hidden_outputs, self.whh)
        hidden2_outputs = self.sigmoid(hidden2_inputs)

        final_inputs = np.dot(hidden2_outputs, self.who)
        final_outputs = self.sigmoid(final_inputs)
        return final_outputs
```

C. Backpropagation

After that we will calculate the MSE for our loss function and do back propagation followed by chain rule..




**Chain rule**

$$y = g(x) \quad z = h(y)$$

$$x \xrightarrow{g()} y \xrightarrow{h()} z \qquad \frac{dz}{dx} = \frac{dz}{\cancel{dy}}\frac{\cancel{dy}}{dx}$$

$$\begin{aligned}\frac{\partial L(\theta)}{\partial w_1} &= \frac{\partial y}{\partial w_1}\frac{\partial L(\theta)}{\partial y} \\ &= \frac{\partial x''}{\partial w_1}\frac{\partial y}{\partial x''}\frac{\partial L(\theta)}{\partial y} \\ &= \frac{\partial z}{\partial w_1}\frac{\partial x''}{\partial z}\frac{\partial y}{\partial x''}\frac{\partial L(\theta)}{\partial y} \\ &= \frac{\partial x'}{\partial w_1}\frac{\partial z}{\partial x'}\frac{\partial x''}{\partial z}\frac{\partial y}{\partial x''}\frac{\partial L(\theta)}{\partial y}\end{aligned}$$

```
def train(self, inputs, targets):
    for epoch in range(1, 1001):
        hidden_inputs = np.dot(inputs, self.wih)
        hidden_outputs = self.sigmoid(hidden_inputs)

        hidden2_inputs = np.dot(hidden_outputs, self.whh)
        hidden2_outputs = self.sigmoid(hidden2_inputs)

        final_inputs = np.dot(hidden2_outputs, self.who)
        final_outputs = self.sigmoid(final_inputs)

        error = (targets - final_outputs) ** 2
        derr = -2 * (targets - final_outputs)
        hidden2_error = np.dot(derr, self.who.T)
        hidden_error = np.dot(hidden2_error, self.whh.T)

        self.who -= self.lr * np.dot(hidden2_outputs.T, (derr * final_outputs * (1.0 - final_outputs)))
        self.whh -= self.lr * np.dot(hidden_outputs.T, (hidden2_error * hidden2_outputs * (1.0 - hidden2_outputs)))
        self.wih -= self.lr * np.dot(inputs.T, (hidden_error * hidden_outputs * (1.0 - hidden_outputs)))
```

3. Results of your testing (30%)

A. Screenshot and comparison figure

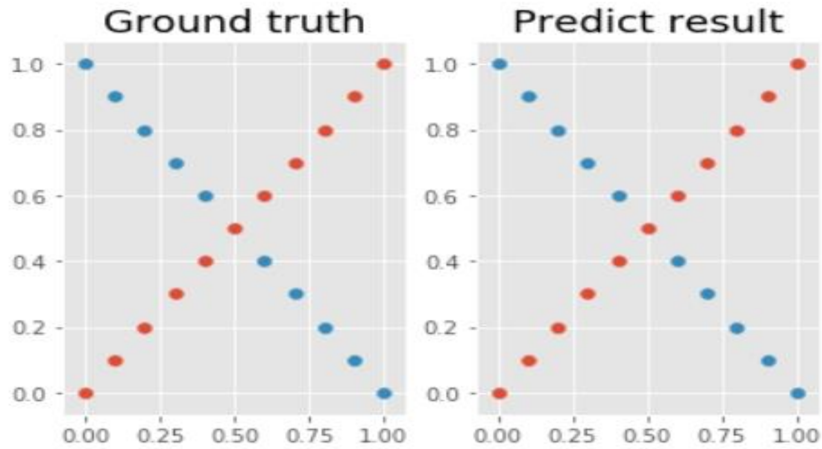
```
x, y = generate_XOR_easy()
```

```
net = Net(2, 4, 4, 1, 0.1)
```

```
net.train(x, y)
```

```
epoch  100 loss : [4.73827861]  Acc : 0.76
epoch  200 loss : [1.70296119]  Acc : 0.95
epoch  300 loss : [0.48552442]  Acc : 1.00
epoch  400 loss : [0.26396059]  Acc : 1.00
epoch  500 loss : [0.40333414]  Acc : 1.00
epoch  600 loss : [0.18786989]  Acc : 1.00
epoch  700 loss : [0.09860397]  Acc : 1.00
epoch  800 loss : [0.0647426]   Acc : 1.00
epoch  900 loss : [0.04512707]  Acc : 1.00
epoch 1000 loss : [0.03229713]  Acc : 1.00
```

```
show_result(x, y, net.query(x).round(0))
```




```
Groud truth vs Prediction
[0] [0.02793106]
[1] [0.99547693]
[0] [0.01345034]
[1] [0.99552912]
[0] [0.01002922]
[1] [0.99557822]
[0] [0.00919904]
[1] [0.99557769]
[0] [0.00921852]
[1] [0.98204262]
[0] [0.00961064]
[0] [0.01017071]
[1] [0.98024552]
[0] [0.01076668]
[1] [0.99478846]
[0] [0.01131061]
[1] [0.99489855]
[0] [0.01175802]
[1] [0.9949578]
[0] [0.01209958]
[1] [0.99501388]
```

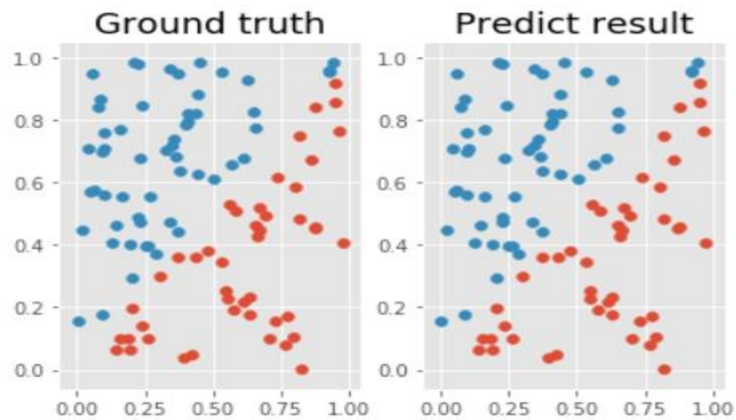
```
x, y = generate_linear()
```

```
net = Net(2, 4, 4, 1, 0.01)
```

```
net.train(x, y)
```

```
epoch  100 loss : [0.00877257]  Acc : 1.00
epoch  200 loss : [0.00844115]  Acc : 1.00
epoch  300 loss : [0.00813474]  Acc : 1.00
epoch  400 loss : [0.00785079]  Acc : 1.00
epoch  500 loss : [0.00758704]  Acc : 1.00
epoch  600 loss : [0.0073415]   Acc : 1.00
epoch  700 loss : [0.00711241]  Acc : 1.00
epoch  800 loss : [0.00689823]  Acc : 1.00
epoch  900 loss : [0.00669756]  Acc : 1.00
epoch 1000 loss : [0.00650918]  Acc : 1.00
```

```
show_result(x, y, net.query(x).round(0))
```




```
Groud truth vs Prediction
[0] [0.01811116]
[0] [0.01810475]
[1] [0.98939093]
[1] [0.98938949]
[1] [0.98935015]
[0] [0.01814245]
[1] [0.88519055]
[1] [0.94878686]
[1] [0.98937117]
[0] [0.01811056]
[1] [0.98930003]
[0] [0.02595461]
[0] [0.01816231]
[0] [0.0181088]
[0] [0.01810676]
[0] [0.01851029]
[1] [0.98938818]
[1] [0.98939123]
[1] [0.98938504]
[1] [0.98938959]
[1] [0.98937703]
```

4. Discussion (20%)

A) For the generate_linear function if we generate the more data, we need to give the learning rate smaller. Otherwise we can not converge the loss function.

B) The first hidden layer is more important the the second layer, we try to compare two experiments, we found that if we put more neurons on first hidden layer, the network will converage earlier than the hidden layer with few neurons.