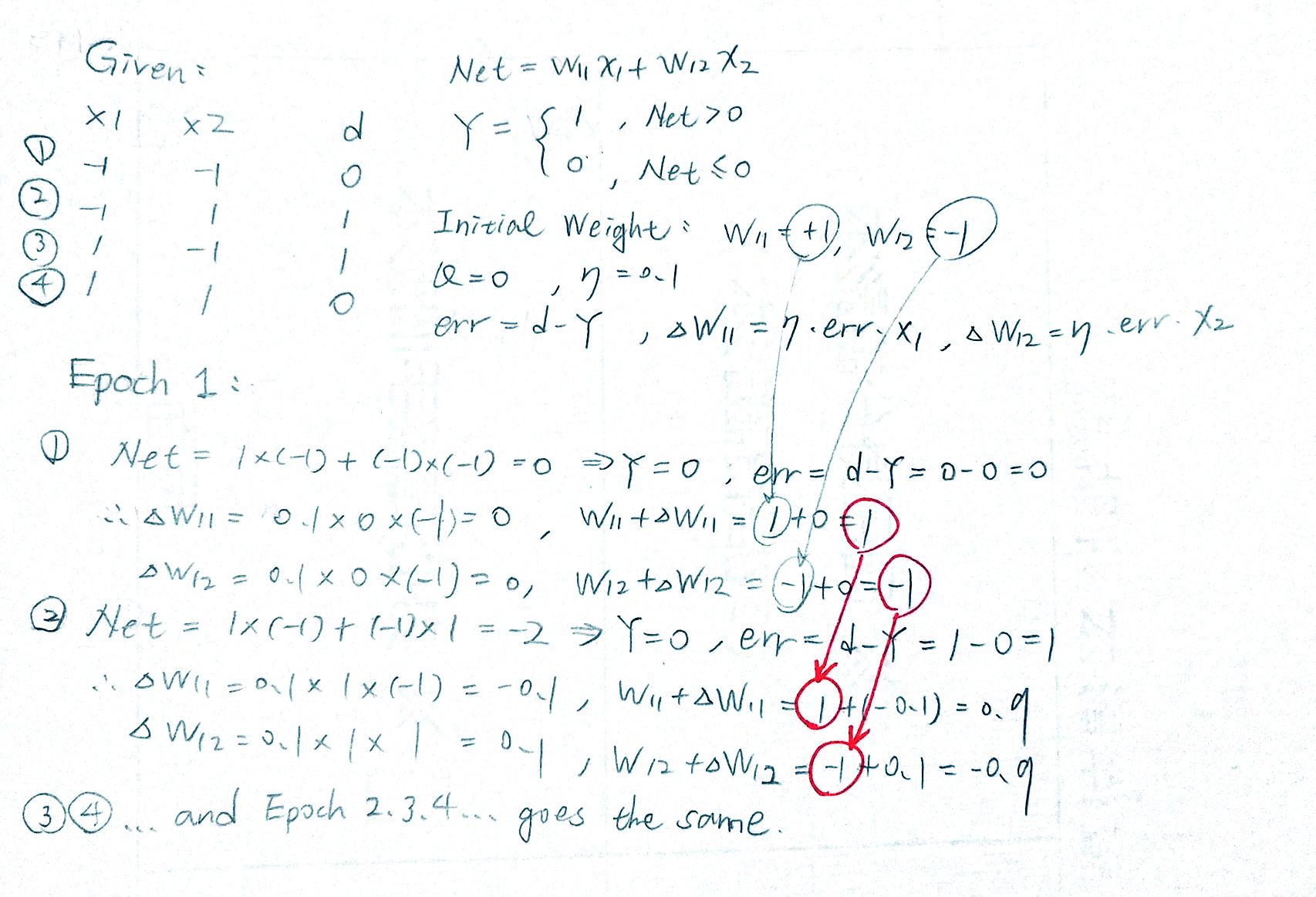
# (1)

### Concept

This problem is a simple version of problem **(2-1)**.There is only two major change. One is that the hidden layer is removed from the structure of problem **(2-1)**. The other is that the problem gives “Y=1 when net>0, Y=0 when net≦0”, which means that the activation function is a step function. Other concepts are the same. For further explanation please refer to problem **(2-1)**.

### Hand Writing



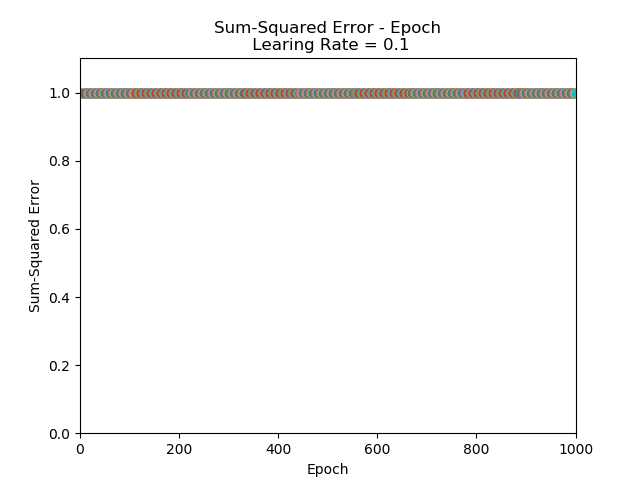
### Code (Python)

import numpy as np  
  
# scipy.special for the sigmoid function expit()  
import scipy.special  
  
import matplotlib.pyplot as plt  
  
class neuralNetwork:  
      
    def \_\_init\_\_(self, learningrate):  
          
        # w\_i\_j, from node i to node j in the next layer  
        # w11 w21  
        # w12 w22 etc   
        # theta 3 is the threshold with input -1  
        # [W11, W12, W13=0], [W21, W22, W23=0], [theta 31, theta 32, theta 33=1]  
        self.wio = np.array([1.0, -1.0, 0.0])   
  
        # learning rate  
        self.lr = learningrate  
          
        # activation function: sigmoid function  
        self.activation\_function = lambda x: 1 if x>0 else 0  
          
        pass  
  
    def train(self, inputs\_list, target):   
        # convert inputs list to 2d array  
        inputs = np.array(inputs\_list, ndmin=1).T  
        targets = np.array(target, ndmin=1).T #  ndmin=1 changed from the standard   
          
        # calculate signals into hidden layer  
        final\_inputs = np.dot(self.wio, inputs)  
        # calculate the signals emerging from hidden layer  
        final\_outputs = self.activation\_function(final\_inputs)  
          
        # output layer error is the (target - actual)  
        final\_errors = targets - final\_outputs  
          
        # update the weights for the links between the input and hidden layers  
        self.wio += self.lr \* final\_errors \* np.transpose(inputs)  
          
        pass  
      
    # query the neural network  
    def query(self, inputs\_list):  
        # convert inputs list to 2d array  
        inputs = np.array(inputs\_list, ndmin=2).T  
          
        # calculate signals into final output layer  
        final\_inputs = np.dot(self.wio, inputs)  
        # calculate the signals emerging from final output layer  
        final\_outputs = self.activation\_function(final\_inputs)  
          
        return final\_outputs  
  
def main():  
  
    input\_list = []  
    target\_list = []  
  
    # Number of Epoch  
    epoch = 1000  
  
    # learning rate  
    learing\_rate = 0.1  
  
    # Inputs & Targets  
      
    input\_list.append([-1, -1]); target\_list.append(0)  
    input\_list.append([-1, 1]); target\_list.append(1)  
    input\_list.append([1, -1]); target\_list.append(1)  
    input\_list.append([1, 1]); target\_list.append(0)  
  
    # Create an instance of neuralNetwork with the learning rate specified  
    nn = neuralNetwork(learing\_rate)  
      
    # Add the threshold input  
    for i in range(len(input\_list)):  
        input\_list[i].append(-1)   
  
    # Plot the Sum-Squared Error - Epoch  
    plt.axis([0, epoch+1, 0, 1.1])  
    plt.title('Sum-Squared Error - Epoch\n Learing Rate = 0.1')  
    plt.xlabel('Epoch')  
    plt.ylabel('Sum-Squared Error')  
  
    # Train & Plot  
    for x in range(0, epoch):  
        for i in range(len(input\_list)):  
            nn.train(input\_list[i], target\_list[i])    
          
        sum\_squared\_errors = 0  
  
        for i in range(len(input\_list)):  
            sum\_squared\_errors += (nn.query(input\_list[i])-target\_list[i])\*\*2  
  
        plt.scatter(x+1, sum\_squared\_errors)  
  
    plt.show()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
    main()

### Results

The result shown below is not surprise because the XOR logic problem with no hidden layer and activation function being step function.

Figure 1



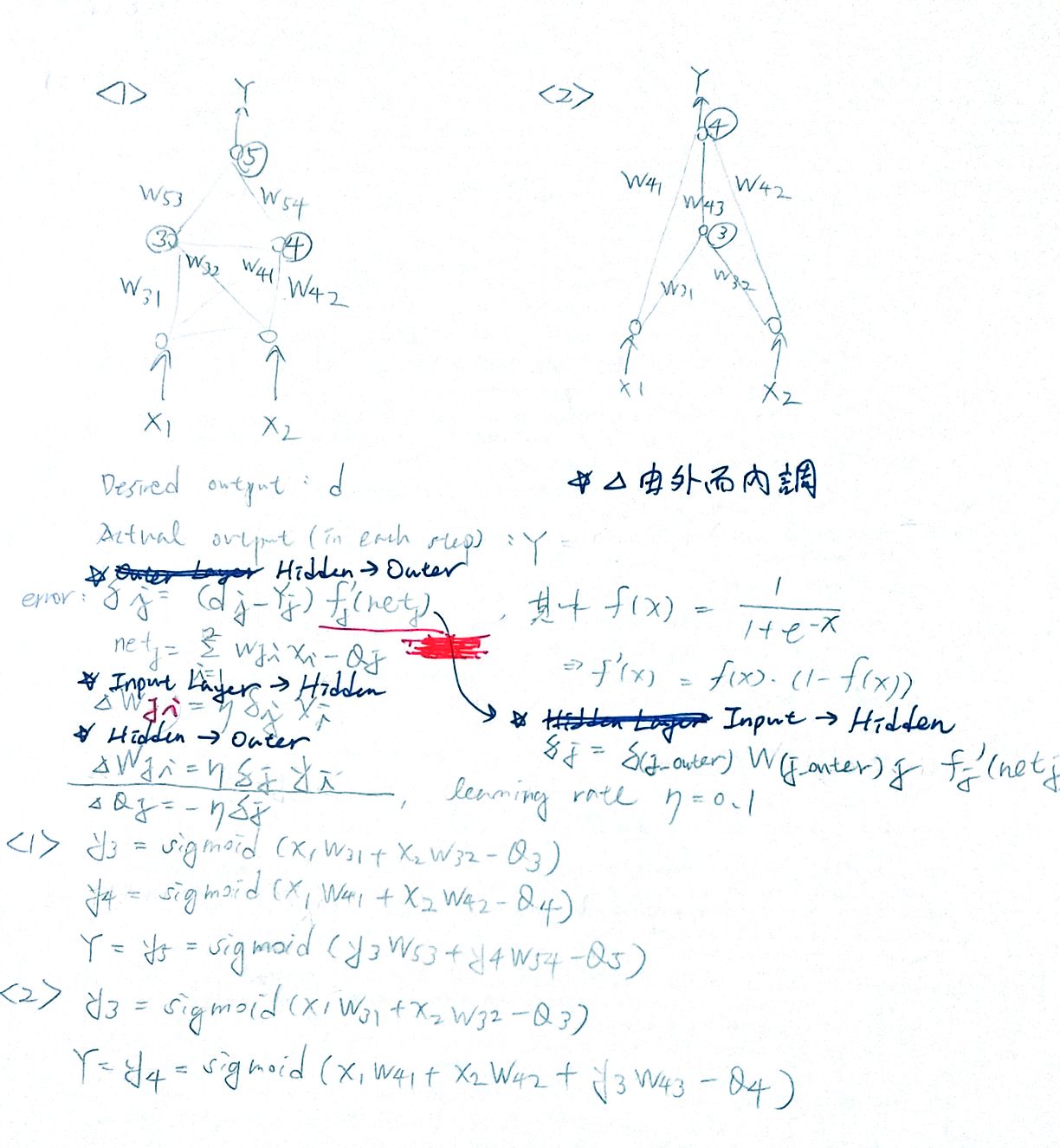
Step by step check with excel explains the same as shown below.

Table 1



# (2)

### Hand Writing

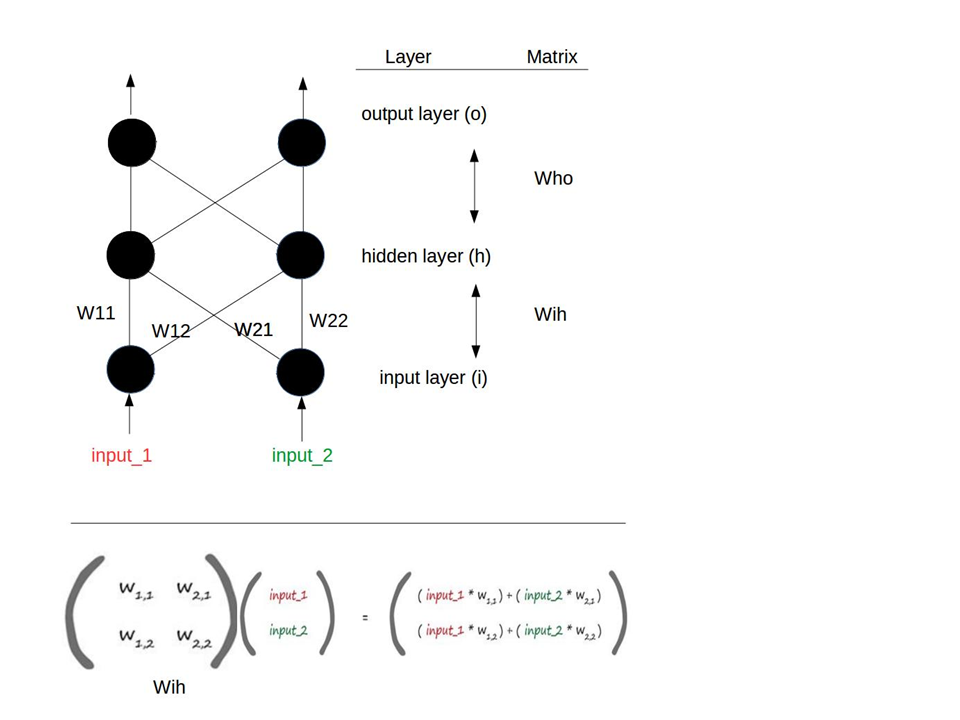


## (2-1) Structure 1

### Concept

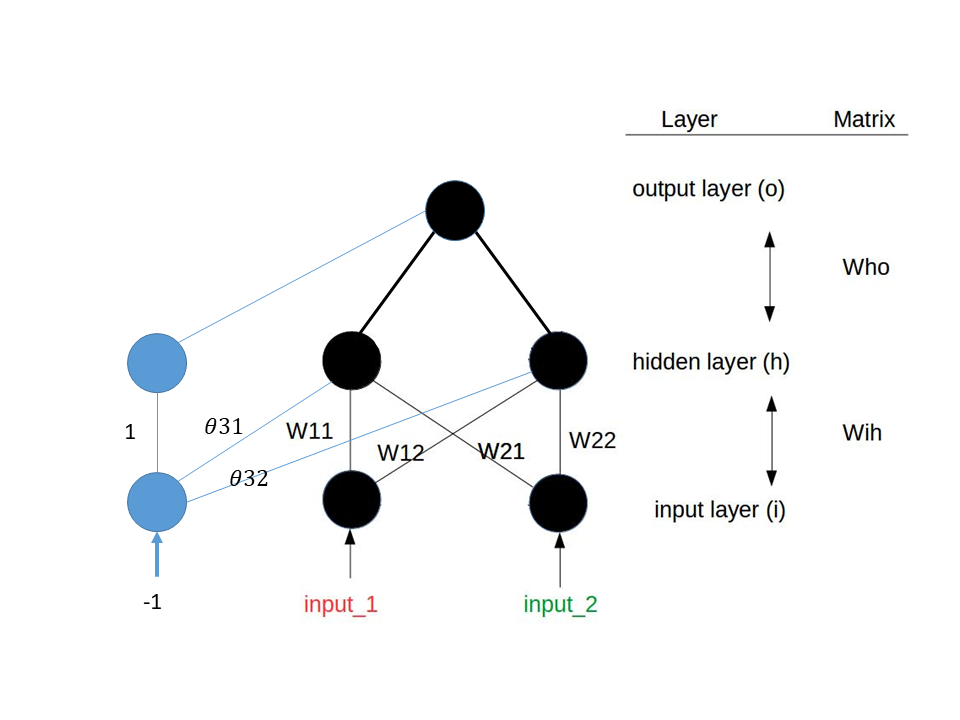
The basic concept of the code is to build a 3-layer back-propagation structure shown below. This is an expandable basic structure mentioned in the “Make Your Own Neural Network”. Note that the indexing is different from which the problem gives.

Figure 2



However, modification has to be made to satisfy this problem.

Figure 3



First, make the output layer only one node (one output).

Next, realize the thresholds in activation functions with input -1 and thresholds as weights.

Given δj=(dj-Yj)•f’(net) and f(x)= 1/(1+e-x)

The activation function is a sigmoid function.

### Code (Python)

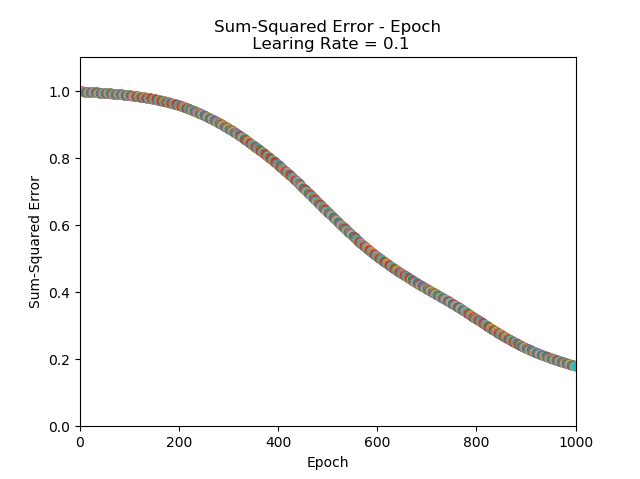
The code utilizes the concept mentioned above to build an expandable 3-layer neural network structure.

import numpy as np  
  
# scipy.special for the sigmoid function expit()  
import scipy.special  
  
import matplotlib.pyplot as plt  
  
class neuralNetwork:  
      
    def \_\_init\_\_(self, learningrate):  
          
        # w\_i\_j, from node i to node j in the next layer  
        # theta 3 is the threshold with input -1  
        # [W11, W12, W13=0], [W21, W22, W23=0], [theta 31, theta 32, theta 33=1]  
        self.wih = np.array([[0.2, -0.4, 0], [0.2, -0.2, 0], [0.8, -0.1, 1]])   
        self.who = np.array([[0.1, -0.4, 0.3]])  
  
        # learning rate  
        self.lr = learningrate  
          
        # activation function: sigmoid function  
        self.activation\_function = lambda x: scipy.special.expit(x)  
          
        pass  
  
    def train(self, inputs\_list, target):   
        # convert inputs list to 2d array  
        inputs = np.array(inputs\_list, ndmin=2).T  
        targets = np.array(target, ndmin=1).T #  ndmin=1 changed from the standard   
          
        # calculate signals into hidden layer  
        hidden\_inputs = np.dot(self.wih, inputs)  
        # calculate the signals emerging from hidden layer  
        hidden\_outputs = self.activation\_function(hidden\_inputs)  
          
        # calculate signals into final output layer  
        final\_inputs = np.dot(self.who, hidden\_outputs)  
        # calculate the signals emerging from final output layer  
        final\_outputs = self.activation\_function(final\_inputs)  
          
        # output layer error is the (target - actual)  
        output\_errors = targets - final\_outputs  
        # hidden layer error is the output\_errors, split by weights, recombined at hidden nodes  
        hidden\_errors = np.dot(self.who.T, output\_errors)   
          
        # update the weights for the links between the hidden and output layers  
        self.who += self.lr \* np.dot((output\_errors \* final\_outputs \* (1.0 - final\_outputs)), np.transpose(hidden\_outputs))  
          
        # update the weights for the links between the input and hidden layers  
        self.wih += self.lr \* np.dot((hidden\_errors \* hidden\_outputs \* (1.0 - hidden\_outputs)), np.transpose(inputs))  
          
        pass  
  
    # query the neural network  
    def query(self, inputs\_list):  
        # convert inputs list to 2d array  
        inputs = np.array(inputs\_list, ndmin=2).T  
          
        # calculate signals into hidden layer  
        hidden\_inputs = np.dot(self.wih, inputs)  
        # calculate the signals emerging from hidden layer  
        hidden\_outputs = self.activation\_function(hidden\_inputs)  
          
        # calculate signals into final output layer  
        final\_inputs = np.dot(self.who, hidden\_outputs)  
        # calculate the signals emerging from final output layer  
        final\_outputs = self.activation\_function(final\_inputs)  
          
        return final\_outputs  
  
def main():  
    input\_list = []  
    target\_list = []  
  
    # Number of Epoch  
    epoch = 1000  
  
    # learning rate  
    learing\_rate = 1  
  
    # Inputs & Targets  
    input\_list.append([-1, -1]); target\_list.append(0)  
    input\_list.append([-1, 1]); target\_list.append(1)  
    input\_list.append([1, -1]); target\_list.append(1)  
    input\_list.append([1, 1]); target\_list.append(0)  
  
    # Create an instance of neuralNetwork with the learning rate specified  
    nn = neuralNetwork(learing\_rate)  
      
    # Add the threshold input  
    for i in range(len(input\_list)):  
        input\_list[i].append(-1)   
  
    # Plot the Sum-Squared Error - Epoch  
    plt.axis([0, epoch+1, 0, 1.1])  
    plt.title('Sum-Squared Error - Epoch\n Learing Rate = 0.1')  
    plt.xlabel('Epoch')  
    plt.ylabel('Sum-Squared Error')  
  
    # Train & Plot  
    for x in range(0, epoch):  
        for i in range(len(input\_list)):  
            nn.train(input\_list[i], target\_list[i])    
          
        sum\_squared\_errors = 0  
  
        for i in range(len(input\_list)):  
            sum\_squared\_errors += (nn.query(input\_list[i])-target\_list[i])\*\*2  
  
        plt.scatter(x+1, sum\_squared\_errors)  
  
    plt.show()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
    main()

### Results

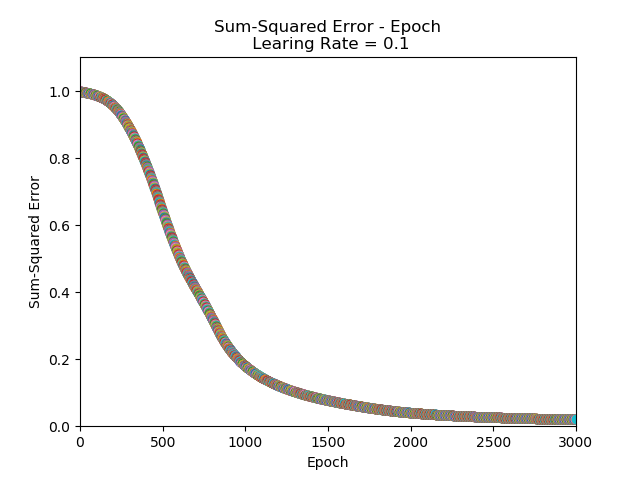
For the learning rate 0.1 specified by the problem, as shown below, the sum-squared error is still 0.2 even when epoch goes to 1000.

Figure 4



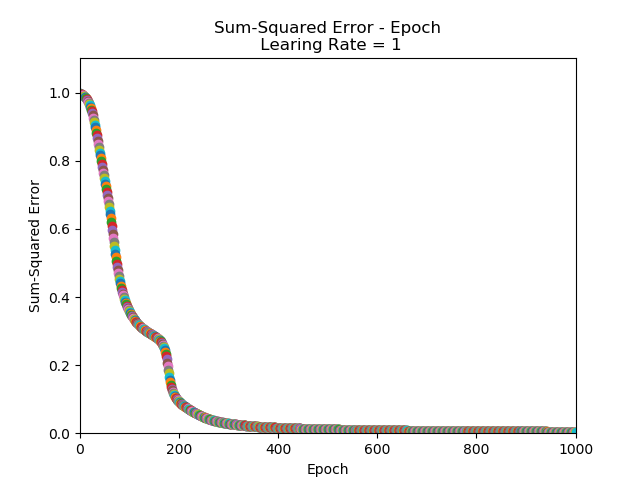
To show the convergence, increase the epoch to 3000 as shown below.

Figure 5



Now increase the learning rate to 1. The sum-squared error converges much more rapidly as shown below compared to Figure 5.

Figure 6



## (2-2) Structure 2

### Concept

The Structure is not a basic type defined in Structure 1. The code will not build an expandable structure for simplicity.

### Code (Python)

import numpy as np  
  
# scipy.special for the sigmoid function expit()  
import scipy.special  
  
import matplotlib.pyplot as plt  
  
class neuralNetwork:  
      
    def \_\_init\_\_(self):  
          
        self.w31 = 0.2  
        self.w32 = -0.4  
        self.w41 = 0.2  
        self.w42 = -0.2  
        self.w43 = -0.4  
        self.w43 = -0.4  
  
        self.theta3 = 0.8  
        self.theta4 = 0.3  
  
        self.learningRate = 0.1  
        pass  
  
  
    def train(self, x1, x2, target):   
        net3 = x1\*self.w31+x2\*self.w32-self.theta3  
        y3 = scipy.special.expit(net3)  
  
        net4 = x1\*self.w41+x2\*self.w42+y3\*self.w43-self.theta4  
        Y = scipy.special.expit(net4)  
  
        error4 = (target - Y)\* scipy.special.expit(net4)\* (1 - scipy.special.expit(net4))  
        error3 = error4\* self.w43\* scipy.special.expit(net3)\* (1 - scipy.special.expit(net3))  
  
        self.w43 += self.learningRate\* error4\* y3  
  
        self.w41 += self.learningRate\* error4\* x1  
        self.w42 += self.learningRate\* error4\* x2  
  
        self.w31 += self.learningRate\* error3\* x1  
        self.w32 += self.learningRate\* error3\* x2  
  
        self.theta3 += -self.learningRate\* error3  
        self.theta4 += -self.learningRate\* error4  
        pass  
  
    def query(self, x1, x2):  
        net3 = x1\*self.w31+x2\*self.w32-self.theta3  
        y3 = scipy.special.expit(net3)  
  
        net4 = x1\*self.w41+x2\*self.w42+y3\*self.w43-self.theta4  
        Y = scipy.special.expit(net4)  
  
        return Y  
  
def main():  
  
    epoch = 10000  
  
    plt.axis([0, epoch+1, 0, 1.5])  
    plt.title('Sum-Squared Error - Epoch\n Learing Rate = 0.1')  
    plt.xlabel('Epoch')  
    plt.ylabel('Sum-Squared Error')  
  
    for x in range(0, epoch):  
          
        nn = neuralNetwork()  
        nn.train(-1, -1, 0)  
        nn.train(-1, 1, 1)  
        nn.train(1, -1, 1)  
        nn.train(1, 1, 0)  
  
        sum\_squared\_errors = (0 - nn.query(-1,-1))\*\*2+(1 - nn.query(-1,1))\*\*2+(1 - nn.query(1,-1))\*\*2+(0 - nn.query(1,1))\*\*2  
  
        plt.scatter(x+1, sum\_squared\_errors)  
  
    plt.show()  
  
if \_\_name\_\_ == '\_\_main\_\_':  
    main()

### Result

The sum-squared error will not converge as shown below.

Figure 7

