In [1]:

```
import numpy as np


class NeuralNetwork(object):
    def __init__(self, input_nodes, hidden_nodes, output_nodes, learning_ra
        # Set number of nodes in input, hidden and output layers.
        self.input_nodes = input_nodes
        self.hidden_nodes = hidden_nodes
        self.output_nodes = output_nodes

        # Initialize weights
        self.weights_input_to_hidden = np.random.normal(0.0, self.input_noc
                                       (self.input_nodes, self.hidden_nodes

        self.weights_hidden_to_output = np.random.normal(0.0, self.hidden_n
                                       (self.hidden_nodes, self.output_node
        self.lr = learning_rate

        #### TODO: Set self.activation_function to your implemented sigmoi
        #
        # Note: in Python, you can define a function with a lambda expressi
        # as shown below.
        self.activation_function = lambda x : 1.0 / (1.0 + np.exp(-1.0*x))

        ### If the lambda code above is not something you're familiar with,
        # You can uncomment out the following three lines and put your
        # implementation there instead.
        #
        #def sigmoid(x):
        #    return 0  # Replace 0 with your sigmoid calculation here
        #self.activation_function = sigmoid


    def train(self, features, targets):
        ''' Train the network on batch of features and targets.

            Arguments
            ---------

            features: 2D array, each row is one data record, each column is
            targets: 1D array of target values

        '''
        n_records = features.shape[0]
        delta_weights_i_h = np.zeros(self.weights_input_to_hidden.shape)
        delta_weights_h_o = np.zeros(self.weights_hidden_to_output.shape)
        for X, y in zip(features, targets):

            final_outputs, hidden_outputs = self.forward_pass_train(X)  # I
            # Implement the backpropagation function below
            delta_weights_i_h, delta_weights_h_o = self.backpropagation(fin
                                                                        del
        self.update_weights(delta_weights_i_h, delta_weights_h_o, n_records

    def forward_pass_train(self, X):
        ''' Implement forward pass here

            Arguments
            ---------
            X: features batch

        '''
```

```
        final_inputs = np.dot(hidden_outputs, self.weights_hidden_to_output
        final_outputs = final_inputs # signals from final output layer

        return final_outputs, hidden_outputs

    def backpropagation(self, final_outputs, hidden_outputs, X, y, delta_we
        ''' Implement backpropagation

            Arguments
            ---------
            final_outputs: output from forward pass
            y: target (i.e. label) batch
            delta_weights_i_h: change in weights from input to hidden layer
            delta_weights_h_o: change in weights from hidden to output laye

        '''
        #### Implement the backward pass here ####
        ### Backward pass ###

        ##NOTA: en el excel unit_tests_model.xlsx mostramos el cálculo de l
        # del error respecto a cada uno de los parámetros, justificando las

        # TODO: Output error - Replace this value with your calculations.
        # Output layer error is the difference between desired target and a
        error = np.transpose(y - final_outputs)

        # TODO: Calculate the hidden layer's contribution to the error
        hidden_error = np.outer(X,(hidden_outputs*(1-hidden_outputs)))

        # TODO: Backpropagated error terms - Replace these values with your
        output_error_term = -1.0*error*hidden_outputs # Para que esto sea a
        # ser E(y_hat) = (1/2)*(y-y_hat)**2, de modo que la derivada respec
        # una función diferente, este término cambiaría

        hidden_error_term = -1.0*error*hidden_error*np.transpose(self.weigh

        # Weight step (input to hidden)
        delta_weights_i_h += hidden_error_term*-1.0
        # Weight step (hidden to output)
        delta_weights_h_o += np.expand_dims(output_error_term*-1.0, axis=1
        return delta_weights_i_h, delta_weights_h_o

    def update_weights(self, delta_weights_i_h, delta_weights_h_o, n_record
        ''' Update weights on gradient descent step

            Arguments
            ---------
            delta_weights_i_h: change in weights from input to hidden layer
            delta_weights_h_o: change in weights from hidden to output laye
            n_records: number of records

        '''
        self.weights_hidden_to_output += (1 / n_records) * self.lr * delta_
        self.weights_input_to_hidden += (1 / n_records) * self.lr * delta_w

    def run(self, features):
        ''' Run a forward pass through the network with input features
```

```
        return final_outputs


##############################################################
# Set your hyperparameters here
##############################################################
iterations = 100
learning_rate = 0.1
hidden_nodes = 2
output_nodes = 1
```