```python
import numpy as np


class NeuralNetwork(object):
    def __init__(self, input_nodes, hidden_nodes, output_nodes, learning_rate):
        # 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_nodes**-0.5,
                                       (self.input_nodes, self.hidden_nodes))

        self.weights_hidden_to_output = np.random.normal(0.0, self.hidden_nodes**-0.5,
                                       (self.hidden_nodes, self.output_nodes))
        self.lr = learning_rate

        #### Set self.activation_function to your implemented sigmoid function ####
        #
        # Note: in Python, you can define a function with a lambda expression,
        # as shown below.
        self.activation_function = lambda x : 1/(1+np.exp(-x))  # Replace 0 with your sigmoid calculation.

        ### 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 a feature
            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)  # Implement the forward pass function below
            # Implement the backproagation function below
            delta_weights_i_h, delta_weights_h_o = self.backpropagation(final_outputs, hidden_outputs, X, y,
                                                                        delta_weights_i_h, delta_weights_h_o)
        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

        '''
        #### Implement the forward pass here ####
        ### Forward pass ###
        # Hidden layer - Replace these values with your calculations.
        hidden_inputs = np.dot(X,self.weights_input_to_hidden) # signals into hidden layer
        hidden_outputs = self.activation_function(hidden_inputs) # signals from hidden layer

        # Output layer - Replace these values with your calculations.
        final_inputs = np.dot(hidden_outputs, self.weights_hidden_to_output) # signals into final output layer
        final_outputs = final_inputs #self.activation_function(final_inputs) # signals from final output layer
        #print (final_outputs, hidden_outputs)
        return final_outputs, hidden_outputs

    def backpropagation(self, final_outputs, hidden_outputs, X, y, delta_weights_i_h, delta_weights_h_o):
        ''' 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 layers
            delta_weights_h_o: change in weights from hidden to output layers

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

        # Output error - Replace this value with your calculations.
        # RG: (y-yhat)
        error = y -  final_outputs # Output layer error is the difference between desired target and actual output.

        # Backpropagated error terms - Replace these values with your calculations.
        # RG: d0 = (y - yhat) * f'(w.a)
        # RG: But since the output activation function is f(x) = x, the derivative f'(x) = 1
        output_error_term = error * 1.0

        # Calculate the hidden layer's contribution to the error
```

```python
        # RG: dh = np.dot(W,d0) * f'(h)
        #print ("weights_hidden_to_output.shape", self.weights_hidden_to_output.shape)
        #print ("output_error_term.shape", output_error_term.shape)
        hidden_error = np.dot(self.weights_hidden_to_output, output_error_term )
        #print ("hidden_error.shape", hidden_error.shape)
        #print ("hidden_outputs.shape", hidden_outputs.shape)
        # here the f(x) = sigmoid(x), hence f'(x) = f(x) * (1 - f(x))
        hidden_error_term = hidden_error * hidden_outputs * (1 - hidden_outputs)

        # Weight step (input to hidden)
        delta_weights_i_h += (hidden_error_term * X[:, None])
        # Weight step (hidden to output)
        delta_weights_h_o += (output_error_term * hidden_outputs[:,None])
        return delta_weights_i_h, delta_weights_h_o

    def update_weights(self, delta_weights_i_h, delta_weights_h_o, n_records):
        ''' Update weights on gradient descent step

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

        '''
        self.weights_hidden_to_output += self.lr * delta_weights_h_o/n_records # update hidden-to-output weights with gradient descent step
        self.weights_input_to_hidden += self.lr * delta_weights_i_h/n_records # update input-to-hidden weights with gradient descent step

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

            Arguments
            ---------
            features: 1D array of feature values
        '''

        #### Implement the forward pass here ####
        # Hidden layer - replace these values with the appropriate calculations.
        #hidden_inputs = None # signals into hidden layer
        #hidden_outputs = None # signals from hidden layer

        # Output layer - Replace these values with the appropriate calculations.
        #final_inputs = None # signals into final output layer
        #final_outputs = None # signals from final output layer

        # RG: DRY: since we already have the forward_pass_train function defined, use it.
        final_outputs , _ = self.forward_pass_train(features)
        return final_outputs


#########################################################
# Set your hyperparameters here
##########################################################

# Working
# TL: 0.066 VL: 0.148
iterations = 3000
learning_rate = 1
hidden_nodes = 8
output_nodes = 1
```