UDACITY

‹ Return to Classroom

Predicting Bike-Sharing Patterns

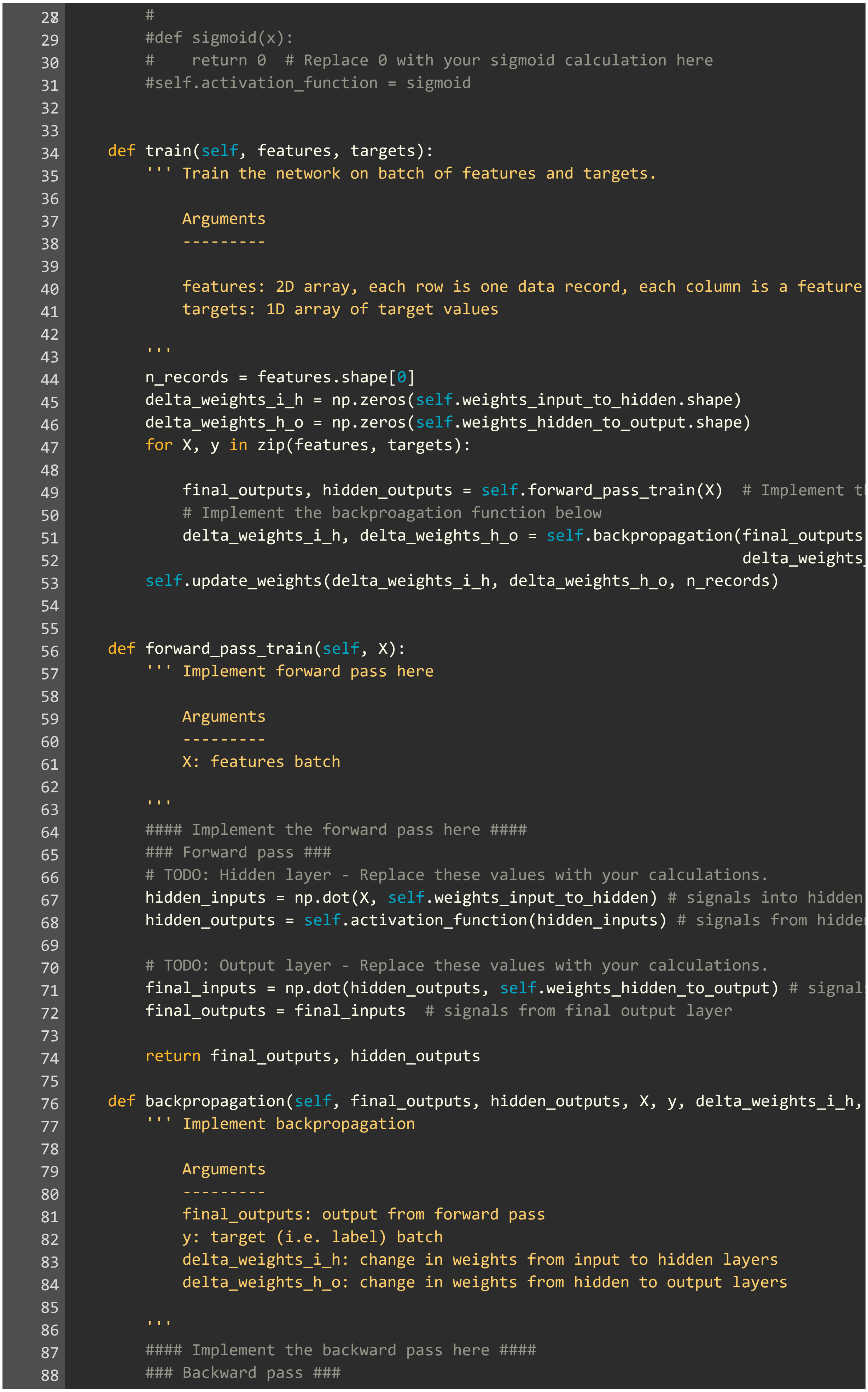
REVIEW

CODE REVIEW

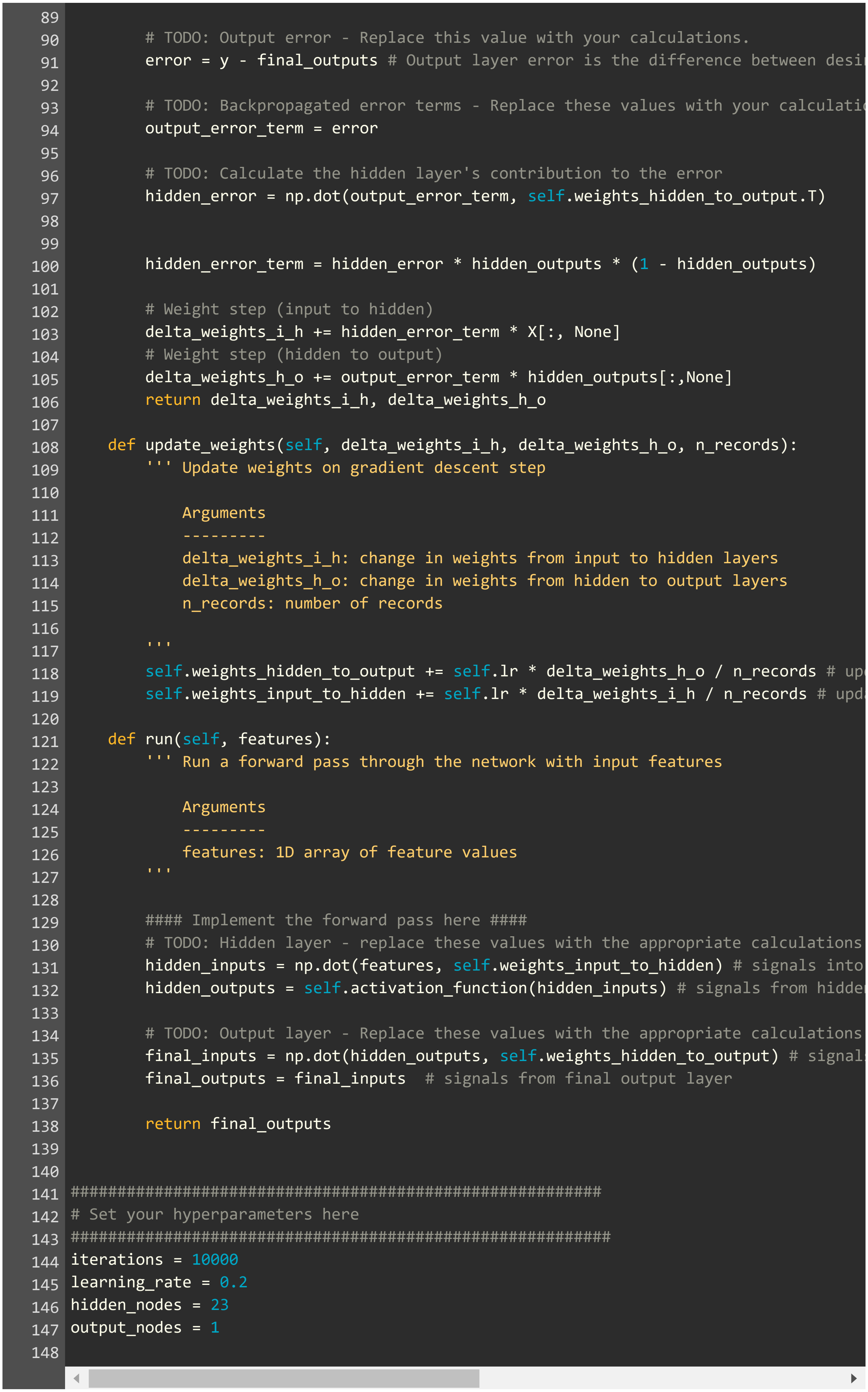
HISTORY

▾ my_answers.py

```
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

        #### TODO: 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 : 1.0/(1+np.exp(-x))  # Replace 0 with yo

        ### 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 t
            # Implement the backproagation function below
            delta_weights_i_h, delta_weights_h_o = self.backpropagation(final_outputs
                                                                        delta_weights_
        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 ###
        # TODO: Hidden layer - Replace these values with your calculations.
        hidden_inputs = np.dot(X, self.weights_input_to_hidden) # signals into hidden
        hidden_outputs = self.activation_function(hidden_inputs) # signals from hidde

        # TODO: Output layer - Replace these values with your calculations.
        final_inputs = np.dot(hidden_outputs, self.weights_hidden_to_output) # signal
        final_outputs = final_inputs  # signals from final output layer

        return final_outputs, hidden_outputs

    def backpropagation(self, final_outputs, hidden_outputs, X, y, delta_weights_i_h,
        ''' 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 ###
```



```

        # TODO: Output error - Replace this value with your calculations.
        error = y - final_outputs # Output layer error is the difference between desi

        # TODO: Backpropagated error terms - Replace these values with your calculati
        output_error_term = error

        # TODO: Calculate the hidden layer's contribution to the error
        hidden_error = np.dot(output_error_term, self.weights_hidden_to_output.T)


        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 # up
        self.weights_input_to_hidden += self.lr * delta_weights_i_h / n_records # upd

    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 ####
        # TODO: Hidden layer - replace these values with the appropriate calculations
        hidden_inputs = np.dot(features, self.weights_input_to_hidden) # signals into
        hidden_outputs = self.activation_function(hidden_inputs) # signals from hidde

        # TODO: Output layer - Replace these values with the appropriate calculations
        final_inputs = np.dot(hidden_outputs, self.weights_hidden_to_output) # signal
        final_outputs = final_inputs  # signals from final output layer

        return final_outputs


#########################################################
# Set your hyperparameters here
##########################################################
iterations = 10000
learning_rate = 0.2
hidden_nodes = 23
output_nodes = 1

```

RETURN TO PATH