Regression Using a Neural Network

In this Assignment #4, we will implement a feedforward neural network from scratch to predict cement strength using the given dataset.

Names and IDs:

Abdelrahman Attia Abdelrahman (20206128)

Shrouk Ashraf Ramdan (20206131)

Ahmed Nasr Hassan (20206129)

First: Let's load and preprocess our data

We start by loading the data from the provided excel file, separating the features and target, and normalizing the features.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
import numpy as np

# Loading our dataset
file_path = 'concrete_data.xlsx'
data = pd.read_excel(file_path)

# Separating features and target
features = data.iloc[:, :-1].values  # Geting the first 4 columns
target = data.iloc[:, -1].values    # Geting the last column

# Normalizing features
scaler = MinMaxScaler()
normalized_features = scaler.fit_transform(features)

# Spliting into training and testing sets
X_train, X_test, y_train, y_test =
train_test_split(normalized_features, target, test_size=0.25,
random_state=42)

# Checking shapes of traina n test
print(f"Training features shape: {X_train.shape}")
print(f"Testing features shape: {X_test.shape}")
```

```
Training features shape: (525, 4)
Testing features shape: (175, 4)
```

Second: Let's define our Neural Network

We will create a custom `NeuralNetwork` class that handles initialization, forward propagation, backward propagation, prediction, and error calculation. This way is better than standalone functions.

```
class NeuralNetwork:

    def __init__(self, num_features, num_neurons_in_hidden_layer, num_predictions, learning_rate):
        # Initialize the number of features, neurons in the hidden layer, and output predictions
        self.num_features = num_features
        self.num_neurons_in_hidden_layer = num_neurons_in_hidden_layer
        self.num_predictions = num_predictions
        self.learning_rate = learning_rate

        ## We can use either ways for initialization ##

        # Random initialization of weights and biases
        self.weights_from_input_to_hidden = np.random.rand(self.num_features, self.num_neurons_in_hidden_layer)
        self.bias_for_hidden_layer = np.random.rand(self.num_neurons_in_hidden_layer)
        self.weights_from_hidden_to_output = np.random.rand(self.num_neurons_in_hidden_layer, self.num_predictions)
        self.bias_for_output_layer = np.random.rand(self.num_predictions)

        # Xavier initialization for weights
        #hidden_layer_limit = np.sqrt(6 / (self.num_features + self.num_neurons_in_hidden_layer))
        #self.weights_from_input_to_hidden = np.random.uniform(-hidden_layer_limit, hidden_layer_limit, (self.num_features, self.num_neurons_in_hidden_layer))
        #self.bias_for_hidden_layer = np.zeros(self.num_neurons_in_hidden_layer)

        #output_layer_limit = np.sqrt(6 / (self.num_neurons_in_hidden_layer + self.num_predictions))
        #self.weights_from_hidden_to_output = np.random.uniform(-output_layer_limit, output_layer_limit, (self.num_neurons_in_hidden_layer, self.num_predictions))
        #self.bias_for_output_layer = np.zeros(self.num_predictions)



    def sigmoid(self, x):
```

```
        return 1 / (1 + np.exp(-x))


    def sigmoid_derivative(self, x):
        return x * (1 - x)



    def forward_prop(self, inputs):
        # Input to hidden layer
        self.hidden_layer_input = np.dot(inputs,
self.weights_from_input_to_hidden) + self.bias_for_hidden_layer
        self.hidden_layer_output =
self.sigmoid(self.hidden_layer_input)

        # Hidden to output layer
        self.output_layer_input = np.dot(self.hidden_layer_output,
self.weights_from_hidden_to_output) + self.bias_for_output_layer

        # Store for use in backward propagation
        self.output = self.output_layer_input  # Linear output for
regression

        return self.output



    def backward_prop(self, inputs, actual_output, predicted_output):

        # Calculate output error
        error = actual_output - predicted_output

        # Compute gradients for output layer
        output_layer_gradient = -2 * error  # Derivative of loss with
respect to the output
        hidden_to_output_weight_update =
np.outer(self.hidden_layer_output, output_layer_gradient)
        output_bias_update = output_layer_gradient

        # Compute gradients for hidden layer
        hidden_layer_error = output_layer_gradient @
self.weights_from_hidden_to_output.T
        hidden_layer_gradient = hidden_layer_error *
self.sigmoid_derivative(self.hidden_layer_output)
        input_to_hidden_weight_update = np.outer(inputs,
hidden_layer_gradient)
        hidden_bias_update = hidden_layer_gradient

        # Update weights and biases
        self.weights_from_hidden_to_output -= self.learning_rate *
```

```
hidden_to_output_weight_update
        self.bias_for_output_layer -= self.learning_rate *
output_layer_gradient

        self.weights_from_input_to_hidden -= self.learning_rate *
input_to_hidden_weight_update
        self.bias_for_hidden_layer -= self.learning_rate *
hidden_layer_gradient

    def train(self, training_data, training_labels, epochs):
        for epoch in range(epochs):
            total_loss = 0
            for inputs, actual_output in zip(training_data,
training_labels):
                predicted_output = self.forward_prop(inputs)
                self.backward_prop(inputs, actual_output,
predicted_output)
                total_loss += (actual_output - predicted_output) ** 2

            if (epoch + 1) % 100 == 0:
                print(f"Epoch {epoch + 1}/{epochs}, Loss: {total_loss
/ len(training_data)}")



    def predict(self, new_data_record):
        return self.forward_prop(new_data_record)



    def calc_error(self, X_test, y_test):
        # Make predictions on the test set
        y_pred = np.array([self.predict(x) for x in X_test])

        # Calculate mean squared error (MSE)
        mse = np.mean((y_test - y_pred.flatten()) ** 2)
        return mse
```

Third: Let's train our Neural Network

Now, time to train the network using the training data for a specified number of epochs.

```
# Define the neural network architecture
input_size = 4  # Number of features
hidden_size = 10  # Number of neurons in the hidden layer (can be
adjusted and tuned)
output_size = 1  # Single output (concrete strength)
learning_rate = 0.001
```

```python
# Initialize our neural network
neural_network = NeuralNetwork(input_size, hidden_size, output_size,
learning_rate)

# Train the network
epochs = 1000  # Number of training epochs
neural_network.train(X_train, y_train, epochs)
```

```
Epoch 100/1000, Loss: [65.20626636]
Epoch 200/1000, Loss: [59.67481446]
Epoch 300/1000, Loss: [57.3332437]
Epoch 400/1000, Loss: [55.54652724]
Epoch 500/1000, Loss: [53.42365703]
Epoch 600/1000, Loss: [51.76129296]
Epoch 700/1000, Loss: [50.95980553]
Epoch 800/1000, Loss: [50.15264839]
Epoch 900/1000, Loss: [49.46731329]
Epoch 1000/1000, Loss: [48.97239151]
```

Finally: Let's evaluate and predict

We're going to Use the trained network to make predictions and evaluate its performance on the test set.

```python
# Calculate mean squared error (MSE)
mse = neural_network.calc_error(X_test, y_test)
print(f"Mean Squared Error on Test Set: {mse}")
```

```
Mean Squared Error on Test Set: 49.064507480434074
```