Lab 12 (Neural Network Regressor)

Submitted by: Ahmed Khalid 200148

Submitted to: Sir Awais

Done Simply without libraries like lab 9

```
import numpy as np

# Generate synthetic data
np.random.seed(42)  # for reproducibility

# Number of samples and features
num_samples = 1000
num_features = 10

# Generate random feature vectors
X_train = np.random.randn(num_samples, num_features)

# Generate continuous targets for regression
y_train = np.random.randn(num_samples, 1)

# Print shapes for verification
print("X_train shape:", X_train.shape)
print("y_train shape:", y_train.shape)

# Define neural network architecture
layer_sizes = [num_features, 2, 4, 8, 1]  # Including input and output layers
num_layers = len(layer_sizes) - 1  # Excluding input layer

# Initialize parameters
parameters = {}
for l in range(1, num_layers + 1):
    parameters['W' + str(l)] = np.random.randn(layer_sizes[l], layer_sizes[l-1]) * 0.01
    parameters['b' + str(l)] = np.zeros((layer_sizes[l], 1))

# Activation functions
def relu(z):
    return np.maximum(0, z)

def relu_derivative(z):
    return np.where(z > 0, 1, 0)

def linear(z):
    return z

def linear_derivative(z):
    return np.ones_like(z)

# Forward propagation
def forward_propagation(X, parameters):
    cache = {'A0': X.T}

    for l in range(1, num_layers):
        cache['Z' + str(l)] = np.dot(parameters['W' + str(l)], cache['A' + str(l-1)]) + parameters['b' + str(l)]
        cache['A' + str(l)] = relu(cache['Z' + str(l)])

    # Output layer (no activation for regression)
    cache['Z' + str(num_layers)] = np.dot(parameters['W' + str(num_layers)], cache['A' + str(num_layers-1)]) + parameters['b' + str(num_l
    cache['A' + str(num_layers)] = linear(cache['Z' + str(num_layers)])

    return cache

# Mean squared error loss
def mean_squared_error_loss(A, Y):
    m = Y.shape[0]
    loss = np.sum((A - Y)**2) / (2 * m)
    return loss

# Backpropagation
def backward_propagation(cache, parameters, X, Y):
    m = Y.shape[0]
    grads = {}
    dZ = cache['A' + str(num_layers)] - Y.T
```

```
    for l in range(num_layers, 0, -1):
        grads['dW' + str(l)] = 1/m * np.dot(dZ, cache['A' + str(l-1)].T)
        grads['db' + str(l)] = 1/m * np.sum(dZ, axis=1, keepdims=True)
        if l > 1:
            dA_prev = np.dot(parameters['W' + str(l)].T, dZ)
            dZ = dA_prev * relu_derivative(cache['Z' + str(l-1)])

    return grads

# Update parameters using gradient descent
def update_parameters(parameters, grads, learning_rate):
    for l in range(1, num_layers + 1):
        parameters['W' + str(l)] -= learning_rate * grads['dW' + str(l)]
        parameters['b' + str(l)] -= learning_rate * grads['db' + str(l)]

# Training hyperparameters
learning_rate = 0.01
num_epochs = 1000

# Training loop
for epoch in range(num_epochs):
    # Forward propagation
    cache = forward_propagation(X_train, parameters)

    # Calculate loss
    loss = mean_squared_error_loss(cache['A' + str(num_layers)].T, y_train)

    # Backpropagation
    grads = backward_propagation(cache, parameters, X_train, y_train)

    # Update parameters
    update_parameters(parameters, grads, learning_rate)

    # Print loss every 100 epochs
    if epoch % 100 == 0:
        print(f"Epoch {epoch}, Loss: {loss}")

# Training loop with outputs
for epoch in range(num_epochs):
    # Forward propagation
    cache = forward_propagation(X_train, parameters)

    # Calculate loss
    loss = mean_squared_error_loss(cache['A' + str(num_layers)].T, y_train)

    # Backpropagation
    grads = backward_propagation(cache, parameters, X_train, y_train)

    # Update parameters
    update_parameters(parameters, grads, learning_rate)

    # Print outputs every 100 epochs
    if epoch % 100 == 0:
        print(f"Epoch {epoch}, Loss: {loss}")
        print("Forward Propagation Outputs:")
        for key, value in cache.items():
            print(key, ":", value)
        print("Backward Propagation Gradients:")
        for key, value in grads.items():
            print(key, ":", value)
        print("\n")
```

```
X_train shape: (1000, 10)
y_train shape: (1000, 1)
Epoch 0, Loss: 0.47116707613691494
Epoch 100, Loss: 0.47047731231425055
Epoch 200, Loss: 0.4703849947765276
Epoch 300, Loss: 0.47037263906330634
Epoch 400, Loss: 0.4703709853816273
Epoch 500, Loss: 0.47037076405181166
Epoch 600, Loss: 0.4703707344282649
Epoch 700, Loss: 0.47037073046156397
Epoch 800, Loss: 0.47037072992835277
Epoch 900, Loss: 0.4703707298570996
Epoch 0, Loss: 0.47037072984785266
Forward Propagation Outputs:
A0 : [[ 0.49671415 -0.46341769  1.46564877 ... -0.9125882  -0.44579531
   1.43362502]
 [-0.1382643  -0.46572975 -0.2257763  ...  0.70138989 -0.50372234
   0.19145072]
 [ 0.64768854  0.24196227  0.0675282  ...  0.8452733   0.52593728
   0.66216875]
 ...
 [ 0.76743473  0.31424733  0.37569802 ... -0.90092112 -1.77598225
```

```
  -0.70531672]
 [-0.46947439 -0.90802408 -0.60063869 ... -1.01268556 -0.98094673
   0.49576557]
 [ 0.54256004 -1.4123037  -0.29169375 ... -1.75995888 -0.77081363
   0.64438845]]
Z1 : [[-0.00298598 -0.0218029  -0.03122495 ...  0.04609311  0.01462545
   0.00390985]
 [-0.00901669 -0.00945001  0.01264058 ... -0.00047607  0.00250494
   0.03629605]]
A1 : [[0.         0.         0.         ... 0.04609311 0.01462545 0.00390985]
 [0.         0.         0.01264058 ... 0.         0.00250494 0.03629605]]
Z2 : [[-2.14049835e-06 -2.14049835e-06 -2.07412839e-04 ...  2.53513940e-04
   3.83011704e-05 -5.69871908e-04]
 [ 9.37876400e-06  9.37876400e-06  1.25995821e-04 ...  3.61164692e-04
   1.44110804e-04  3.74072311e-04]
 [-5.19070750e-06 -5.19070750e-06  1.96819222e-04 ...  5.21381815e-05
   5.30314396e-05  5.79721887e-04]
 [-1.24588575e-05 -1.24588575e-05 -1.91553646e-04 ...  8.74346062e-05
  -1.62529886e-05 -5.18236700e-04]]
A2 : [[0.00000000e+00 0.00000000e+00 0.00000000e+00 ... 2.53513940e-04
  3.83011704e-05 0.00000000e+00]
 [9.37876400e-06 9.37876400e-06 1.25995821e-04 ... 3.61164692e-04
  1.44110804e-04 3.74072311e-04]
 [0.00000000e+00 0.00000000e+00 1.96819222e-04 ... 5.21381815e-05
  5.30314396e-05 5.79721887e-04]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 ... 8.74346062e-05
  0.00000000e+00 0.00000000e+00]]
Z3 : [[ 4.66764238e-04  4.66764238e-04  4.65862621e-04 ...  4.68738678e-04
   4.66914232e-04  4.64160506e-04]
 [ 2.46695413e-04  2.46695413e-04  2.46122373e-04 ...  2.44633424e-04
   2.46925667e-04  2.45217592e-04]
 [ 7.14636689e-04  7.14636689e-04  7.15464539e-04 ...  7.15307740e-04
   7.15263547e-04  7.17166908e-04]
 ...
 [ 1.30859400e-04  1.30859400e-04  1.36234181e-04 ...  1.37315220e-04
```

Done using libraries

Loading the data for regression

```
# Reading the cleaned numeric car prices data
import pandas as pd
import numpy as np

# To remove the scientific notation from numpy arrays
np.set_printoptions(suppress=True)

CarPricesDataNumeric=pd.read_pickle('CarPricesData.pkl')
CarPricesDataNumeric.head()
```

	Age	KM	Weight	HP	MetColor	CC	Doors	Price
0	23.0	46986	1165.0	90	1	2000.0	3	13500
1	23.0	72937	1165.0	90	1	2000.0	3	13750
2	24.0	41711	1165.0	90	1	2000.0	3	13950
3	26.0	48000	1165.0	90	0	2000.0	3	14950
4	30.0	38500	1170.0	90	0	2000.0	3	13750

Splitting the Data into Training and Testing

```
# Separate Target Variable and Predictor Variables
TargetVariable=['Price']
Predictors=['Age', 'KM', 'Weight', 'HP', 'MetColor', 'CC', 'Doors']

X=CarPricesDataNumeric[Predictors].values
y=CarPricesDataNumeric[TargetVariable].values

### Sandardization of data ###
from sklearn.preprocessing import StandardScaler
PredictorScaler=StandardScaler()
TargetVarScaler=StandardScaler()

# Storing the fit object for later reference
PredictorScalerFit=PredictorScaler.fit(X)
TargetVarScalerFit=TargetVarScaler.fit(y)

# Generating the standardized values of X and y
X=PredictorScalerFit.transform(X)
y=TargetVarScalerFit.transform(y)

# Split the data into training and testing set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Quick sanity check with the shapes of Training and testing datasets
print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

```
(1004, 7)
(1004, 1)
(431, 7)
(431, 1)
```

Installing the required libraries

```
# Installing required libraries
!pip install tensorflow
!pip install keras
```

```
Requirement already satisfied: tensorflow in /usr/local/lib/python3.10/dist-packages (2.15.0)
Requirement already satisfied: absl-py>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (1.4.0)
Requirement already satisfied: astunparse>=1.6.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (1.6.3)
Requirement already satisfied: flatbuffers>=23.5.26 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (24.3.25)
Requirement already satisfied: gast!=0.5.0,!=0.5.1,!=0.5.2,>=0.2.1 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (0.5
Requirement already satisfied: google-pasta>=0.1.1 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (0.2.0)
Requirement already satisfied: h5py>=2.9.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (3.9.0)
Requirement already satisfied: libclang>=13.0.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (18.1.1)
Requirement already satisfied: ml-dtypes~=0.2.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (0.2.0)
Requirement already satisfied: numpy<2.0.0,>=1.23.5 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (1.25.2)
Requirement already satisfied: opt-einsum>=2.3.2 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (3.3.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages (from tensorflow) (24.0)
Requirement already satisfied: protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<5.0.0dev,>=3.20.3 in /usr/local/lib/py
Requirement already satisfied: setuptools in /usr/local/lib/python3.10/dist-packages (from tensorflow) (67.7.2)
Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (1.16.0)
Requirement already satisfied: termcolor>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (2.4.0)
Requirement already satisfied: typing-extensions>=3.6.6 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (4.11.0)
Requirement already satisfied: wrapt<1.15,>=1.11.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (1.14.1)
Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.23.1 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (0
Requirement already satisfied: grpcio<2.0,>=1.24.3 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (1.63.0)
Requirement already satisfied: tensorboard<2.16,>=2.15 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (2.15.2)
Requirement already satisfied: tensorflow-estimator<2.16,>=2.15.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (2.15
Requirement already satisfied: keras<2.16,>=2.15.0 in /usr/local/lib/python3.10/dist-packages (from tensorflow) (2.15.0)
Requirement already satisfied: wheel<1.0,>=0.23.0 in /usr/local/lib/python3.10/dist-packages (from astunparse>=1.6.0->tensorflow) (
Requirement already satisfied: google-auth<3,>=1.6.3 in /usr/local/lib/python3.10/dist-packages (from tensorboard<2.16,>=2.15->tenso
Requirement already satisfied: google-auth-oauthlib<2,>=0.5 in /usr/local/lib/python3.10/dist-packages (from tensorboard<2.16,>=2.15
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.10/dist-packages (from tensorboard<2.16,>=2.15->tensorflow
Requirement already satisfied: requests<3,>=2.21.0 in /usr/local/lib/python3.10/dist-packages (from tensorboard<2.16,>=2.15->tensor
Requirement already satisfied: tensorboard-data-server<0.8.0,>=0.7.0 in /usr/local/lib/python3.10/dist-packages (from tensorboard<2
Requirement already satisfied: werkzeug>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from tensorboard<2.16,>=2.15->tensorflow
Requirement already satisfied: cachetools<6.0,>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from google-auth<3,>=1.6.3->tensor
Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/lib/python3.10/dist-packages (from google-auth<3,>=1.6.3->tensorb
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.10/dist-packages (from google-auth<3,>=1.6.3->tensorboard<2.
Requirement already satisfied: requests-oauthlib>=0.7.0 in /usr/local/lib/python3.10/dist-packages (from google-auth-oauthlib<2,>=0
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests<3,>=2.21.0->tensor
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests<3,>=2.21.0->tensorboard<2.16,
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests<3,>=2.21.0->tensorboard
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests<3,>=2.21.0->tensorboard
Requirement already satisfied: MarkupSafe>=2.1.1 in /usr/local/lib/python3.10/dist-packages (from werkzeug>=1.0.1->tensorboard<2.16
Requirement already satisfied: pyasn1<0.7.0,>=0.4.6 in /usr/local/lib/python3.10/dist-packages (from pyasn1-modules>=0.2.1->google-a
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.10/dist-packages (from requests-oauthlib>=0.7.0->google-aut
Requirement already satisfied: keras in /usr/local/lib/python3.10/dist-packages (2.15.0)
```

Creating Deep Learning- Artificial Neural Networks(ANN) model

```
# importing the libraries
from keras.models import Sequential
from keras.layers import Dense

# create ANN model
model = Sequential()

# Defining the Input layer and FIRST hidden layer, both are same!
model.add(Dense(units=5, input_dim=7, kernel_initializer='normal', activation='relu'))

# Defining the Second layer of the model
# after the first layer we don't have to specify input_dim as keras configure it automatically
model.add(Dense(units=5, kernel_initializer='normal', activation='tanh'))

# The output neuron is a single fully connected node
# Since we will be predicting a single number
model.add(Dense(1, kernel_initializer='normal'))

# Compiling the model
model.compile(loss='mean_squared_error', optimizer='adam')

# Fitting the ANN to the Training set
model.fit(X_train, y_train ,batch_size = 20, epochs = 50, verbose=1)
```

```
Epoch 1/50
51/51 [==============================] - 1s 2ms/step - loss: 0.9823
Epoch 2/50
51/51 [==============================] - 0s 2ms/step - loss: 0.9245
Epoch 3/50
51/51 [==============================] - 0s 2ms/step - loss: 0.6910
Epoch 4/50
51/51 [==============================] - 0s 2ms/step - loss: 0.4168
Epoch 5/50
51/51 [==============================] - 0s 2ms/step - loss: 0.2844
Epoch 6/50
51/51 [==============================] - 0s 2ms/step - loss: 0.2373
Epoch 7/50
51/51 [==============================] - 0s 2ms/step - loss: 0.2120
Epoch 8/50
51/51 [==============================] - 0s 2ms/step - loss: 0.1928
Epoch 9/50
51/51 [==============================] - 0s 2ms/step - loss: 0.1802
Epoch 10/50
51/51 [==============================] - 0s 2ms/step - loss: 0.1706
Epoch 11/50
51/51 [==============================] - 0s 2ms/step - loss: 0.1630
Epoch 12/50
51/51 [==============================] - 0s 2ms/step - loss: 0.1568
Epoch 13/50
51/51 [==============================] - 0s 2ms/step - loss: 0.1525
Epoch 14/50
51/51 [==============================] - 0s 2ms/step - loss: 0.1489
Epoch 15/50
51/51 [==============================] - 0s 2ms/step - loss: 0.1449
Epoch 16/50
51/51 [==============================] - 0s 2ms/step - loss: 0.1417
Epoch 17/50
51/51 [==============================] - 0s 2ms/step - loss: 0.1383
Epoch 18/50
51/51 [==============================] - 0s 2ms/step - loss: 0.1363
Epoch 19/50
51/51 [==============================] - 0s 2ms/step - loss: 0.1340
Epoch 20/50
51/51 [==============================] - 0s 2ms/step - loss: 0.1316
Epoch 21/50
51/51 [==============================] - 0s 2ms/step - loss: 0.1293
Epoch 22/50
51/51 [==============================] - 0s 2ms/step - loss: 0.1278
Epoch 23/50
51/51 [==============================] - 0s 2ms/step - loss: 0.1258
Epoch 24/50
51/51 [==============================] - 0s 2ms/step - loss: 0.1242
Epoch 25/50
51/51 [==============================] - 0s 2ms/step - loss: 0.1227
Epoch 26/50
51/51 [==============================] - 0s 2ms/step - loss: 0.1217
Epoch 27/50
51/51 [==============================] - 0s 2ms/step - loss: 0.1205
Epoch 28/50
51/51 [==============================] - 0s 2ms/step - loss: 0.1188
Epoch 29/50
51/51 [==============================] - 0s 2ms/step - loss: 0.1183
```

Hyperparameter tuning of ANN Finding best set of parameters using manual grid search

```
def FunctionFindBestParams(X_train, y_train, X_test, y_test):

    # Defining the list of hyper parameters to try
    batch_size_list=[5, 10, 15, 20]
    epoch_list = [5, 10, 50, 100]

    SearchResultsData = pd.DataFrame(columns=['TrialNumber', 'Parameters', 'Accuracy'])

    # initializing the trials
    TrialNumber = 0
    for batch_size_trial in batch_size_list:
        for epochs_trial in epoch_list:
            TrialNumber += 1
            # create ANN model
            model = Sequential()
            # Defining the first layer of the model
            model.add(Dense(units=5, input_dim=X_train.shape[1], kernel_initializer='normal', activation='relu'))

            # Defining the Second layer of the model
            model.add(Dense(units=5, kernel_initializer='normal', activation='relu'))

            # The output neuron is a single fully connected node
            # Since we will be predicting a single number
            model.add(Dense(1, kernel_initializer='normal'))

            # Compiling the model
            model.compile(loss='mean_squared_error', optimizer='adam')

            # Fitting the ANN to the Training set
            model.fit(X_train, y_train, batch_size=batch_size_trial, epochs=epochs_trial, verbose=0)

            MAPE = np.mean(100 * (np.abs(y_test-model.predict(X_test))/y_test))

            # printing the results of the current iteration
            print(TrialNumber, 'Parameters:', 'batch_size:', batch_size_trial, '-', 'epochs:', epochs_trial, 'Accuracy:', 100-MAPE)

            SearchResultsData = pd.concat([SearchResultsData, pd.DataFrame(data=[[TrialNumber, str(batch_size_trial)+'-'+str(epochs_tria
    return SearchResultsData

# Calling the function
ResultsData = FunctionFindBestParams(X_train, y_train, X_test, y_test)


# Calculate and print the accuracy on the test set
test_predictions = best_model.predict(X_test)
MAPE_test = np.mean(100 * (np.abs(y_test - test_predictions) / y_test))
accuracy_test = 100 - MAPE_test
print("Test Accuracy:", accuracy_test)
```

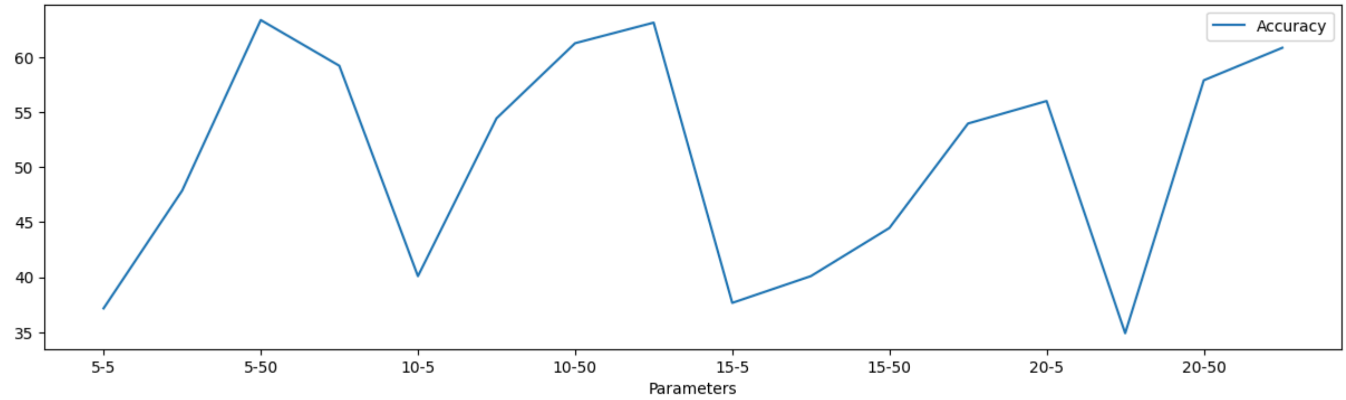
```
14/14 [==============================] - 0s 2ms/step
1 Parameters: batch_size: 5 - epochs: 5 Accuracy: 37.17182836377365
14/14 [==============================] - 0s 1ms/step
2 Parameters: batch_size: 5 - epochs: 10 Accuracy: 47.87825730247413
14/14 [==============================] - 0s 3ms/step
3 Parameters: batch_size: 5 - epochs: 50 Accuracy: 63.380615125903674
14/14 [==============================] - 0s 2ms/step
4 Parameters: batch_size: 5 - epochs: 100 Accuracy: 59.22339572534605
14/14 [==============================] - 0s 2ms/step
5 Parameters: batch_size: 10 - epochs: 5 Accuracy: 40.09922615333969
14/14 [==============================] - 0s 2ms/step
6 Parameters: batch_size: 10 - epochs: 10 Accuracy: 54.444331306398915
14/14 [==============================] - 0s 2ms/step
7 Parameters: batch_size: 10 - epochs: 50 Accuracy: 61.27449583691738
14/14 [==============================] - 0s 2ms/step
8 Parameters: batch_size: 10 - epochs: 100 Accuracy: 63.14488860261723
14/14 [==============================] - 0s 2ms/step
9 Parameters: batch_size: 15 - epochs: 5 Accuracy: 37.67121015954872
14/14 [==============================] - 0s 2ms/step
10 Parameters: batch_size: 15 - epochs: 10 Accuracy: 40.10039040027554
14/14 [==============================] - 0s 2ms/step
11 Parameters: batch_size: 15 - epochs: 50 Accuracy: 44.482419461064026
14/14 [==============================] - 0s 3ms/step
12 Parameters: batch_size: 15 - epochs: 100 Accuracy: 53.973784924620745
14/14 [==============================] - 0s 2ms/step
13 Parameters: batch_size: 20 - epochs: 5 Accuracy: 56.01810215234351
14/14 [==============================] - 0s 2ms/step
14 Parameters: batch_size: 20 - epochs: 10 Accuracy: 34.91105683047358
14/14 [==============================] - 0s 1ms/step
15 Parameters: batch_size: 20 - epochs: 50 Accuracy: 57.903263350950816
```

```
14/14 [==============================] - 0s 2ms/step
16 Parameters: batch_size: 20 - epochs: 100 Accuracy: 60.86011217045063
14/14 [==============================] - 0s 2ms/step
Test Accuracy: 57.176199258315364
```

Plotting the parameter trial results

```
%matplotlib inline
ResultsData.plot(x='Parameters', y='Accuracy', figsize=(15,4), kind='line')
```

```
<Axes: xlabel='Parameters'>
```



Training the ANN model with the best parameters

```
# Fitting the ANN to the Training set
model.fit(X_train, y_train ,batch_size = 15, epochs = 5, verbose=0)

# Generating Predictions on testing data
Predictions=model.predict(X_test)

# Scaling the predicted Price data back to original price scale
Predictions=TargetVarScalerFit.inverse_transform(Predictions)

# Scaling the y_test Price data back to original price scale
y_test_orig=TargetVarScalerFit.inverse_transform(y_test)

# Scaling the test data back to original scale
Test_Data=PredictorScalerFit.inverse_transform(X_test)

TestingData=pd.DataFrame(data=Test_Data, columns=Predictors)
TestingData['Price']=y_test_orig
TestingData['PredictedPrice']=Predictions
TestingData.head()
```

```
14/14 [==============================] - 0s 2ms/step
```

	Age	KM	Weight	HP	MetColor	CC	Doors	Price	PredictedPrice
0	59.0	80430.0	1065.0	110.0	1.0	1600.0	3.0	9950.0	9869.646484
1	62.0	64797.0	1075.0	110.0	1.0	1600.0	5.0	7995.0	10026.861328
2	59.0	130000.0	1135.0	72.0	1.0	2000.0	4.0	7500.0	8500.763672
3	69.0	42800.0	1050.0	110.0	1.0	1600.0	3.0	9950.0	9267.764648
4	65.0	47014.0	1015.0	86.0	1.0	1300.0	3.0	8950.0	8989.233398

Finding the accuracy of the model

```
# Find the best parameters from ResultsData
best_params_index = ResultsData['Accuracy'].idxmax()
best_params = ResultsData.loc[best_params_index, 'Parameters']
best_batch_size, best_epochs = map(int, best_params.split('-'))

# Train the model with the best parameters
best_model = Sequential()
best_model.add(Dense(units=5, input_dim=7, kernel_initializer='normal', activation='relu'))
best_model.add(Dense(units=5, kernel_initializer='normal', activation='relu'))
best_model.add(Dense(1, kernel_initializer='normal'))
best_model.compile(loss='mean_squared_error', optimizer='adam')
best_model.fit(X_train, y_train, batch_size=best_batch_size, epochs=best_epochs, verbose=1)

# Evaluate the model on the test set
test_loss = best_model.evaluate(X_test, y_test)
print("Test Loss:", test_loss)

# Calculate and print the accuracy on the test set
test_predictions = best_model.predict(X_test)
MAPE_test = np.mean(100 * (np.abs(y_test - test_predictions) / y_test))
accuracy_test = 100 - MAPE_test
print("Test Accuracy:", accuracy_test)
```

```
Epoch 1/50
201/201 [==============================] - 1s 2ms/step - loss: 0.9297
Epoch 2/50
201/201 [==============================] - 0s 2ms/step - loss: 0.3909
Epoch 3/50
201/201 [==============================] - 0s 2ms/step - loss: 0.1591
Epoch 4/50
201/201 [==============================] - 0s 2ms/step - loss: 0.1426
Epoch 5/50
201/201 [==============================] - 0s 2ms/step - loss: 0.1352
Epoch 6/50
201/201 [==============================] - 0s 2ms/step - loss: 0.1293
Epoch 7/50
201/201 [==============================] - 0s 2ms/step - loss: 0.1262
Epoch 8/50
201/201 [==============================] - 0s 2ms/step - loss: 0.1237
Epoch 9/50
201/201 [==============================] - 0s 2ms/step - loss: 0.1204
Epoch 10/50
201/201 [==============================] - 0s 2ms/step - loss: 0.1183
Epoch 11/50
201/201 [==============================] - 0s 2ms/step - loss: 0.1160
Epoch 12/50
201/201 [==============================] - 1s 3ms/step - loss: 0.1146
Epoch 13/50
201/201 [==============================] - 0s 2ms/step - loss: 0.1136
Epoch 14/50
201/201 [==============================] - 0s 2ms/step - loss: 0.1122
Epoch 15/50
201/201 [==============================] - 0s 2ms/step - loss: 0.1110
Epoch 16/50
201/201 [==============================] - 0s 2ms/step - loss: 0.1113
Epoch 17/50
201/201 [==============================] - 0s 2ms/step - loss: 0.1103
Epoch 18/50
201/201 [==============================] - 0s 2ms/step - loss: 0.1098
Epoch 19/50
201/201 [==============================] - 0s 2ms/step - loss: 0.1099
Epoch 20/50
201/201 [==============================] - 0s 2ms/step - loss: 0.1091
Epoch 21/50
201/201 [==============================] - 0s 2ms/step - loss: 0.1092
Epoch 22/50
201/201 [==============================] - 0s 2ms/step - loss: 0.1089
Epoch 23/50
201/201 [==============================] - 0s 2ms/step - loss: 0.1078
Epoch 24/50
201/201 [==============================] - 0s 2ms/step - loss: 0.1078
Epoch 25/50
201/201 [==============================] - 0s 2ms/step - loss: 0.1077
Epoch 26/50
201/201 [==============================] - 0s 2ms/step - loss: 0.1074
Epoch 27/50
201/201 [==============================] - 0s 2ms/step - loss: 0.1059
Epoch 28/50
201/201 [==============================] - 0s 2ms/step - loss: 0.1072
Epoch 29/50
201/201 [==============================] - 0s 2ms/step - loss: 0.1055
```

Further Training

```
# Further training the best model with additional epochs
additional_epochs = 50  # You can adjust this number as needed

best_model.fit(X_train, y_train, batch_size=best_batch_size, epochs=best_epochs + additional_epochs, verbose=1)

# Evaluate the model on the test set after further training
test_loss = best_model.evaluate(X_test, y_test)
print("Test Loss after further training:", test_loss)

# Calculate and print the accuracy on the test set after further training
test_predictions = best_model.predict(X_test)
MAPE_test = np.mean(100 * (np.abs(y_test - test_predictions) / y_test))
accuracy_test = 100 - MAPE_test
print("Test Accuracy after further training:", accuracy_test)
```

```
Epoch 1/100
201/201 [==============================] - 0s 2ms/step - loss: 0.0951
Epoch 2/100
201/201 [==============================] - 0s 2ms/step - loss: 0.0959
Epoch 3/100
201/201 [==============================] - 0s 2ms/step - loss: 0.0944
Epoch 4/100
201/201 [==============================] - 0s 2ms/step - loss: 0.0946
Epoch 5/100
201/201 [==============================] - 0s 2ms/step - loss: 0.0940
Epoch 6/100
201/201 [==============================] - 0s 2ms/step - loss: 0.0942
Epoch 7/100
201/201 [==============================] - 0s 2ms/step - loss: 0.0949
Epoch 8/100
201/201 [==============================] - 1s 4ms/step - loss: 0.0933
Epoch 9/100
201/201 [==============================] - 1s 3ms/step - loss: 0.0937
Epoch 10/100
201/201 [==============================] - 1s 3ms/step - loss: 0.0942
Epoch 11/100
201/201 [==============================] - 1s 3ms/step - loss: 0.0932
Epoch 12/100
201/201 [==============================] - 1s 4ms/step - loss: 0.0933
Epoch 13/100
201/201 [==============================] - 1s 3ms/step - loss: 0.0928
Epoch 14/100
201/201 [==============================] - 1s 4ms/step - loss: 0.0926
Epoch 15/100
201/201 [==============================] - 1s 3ms/step - loss: 0.0930
Epoch 16/100
201/201 [==============================] - 1s 6ms/step - loss: 0.0926
Epoch 17/100
201/201 [==============================] - 1s 5ms/step - loss: 0.0933
Epoch 18/100
201/201 [==============================] - 1s 7ms/step - loss: 0.0926
Epoch 19/100
201/201 [==============================] - 2s 8ms/step - loss: 0.0922
Epoch 20/100
201/201 [==============================] - 1s 5ms/step - loss: 0.0922
Epoch 21/100
201/201 [==============================] - 1s 4ms/step - loss: 0.0909
Epoch 22/100
201/201 [==============================] - 1s 5ms/step - loss: 0.0926
Epoch 23/100
201/201 [==============================] - 1s 4ms/step - loss: 0.0918
Epoch 24/100
201/201 [==============================] - 1s 4ms/step - loss: 0.0916
Epoch 25/100
201/201 [==============================] - 1s 3ms/step - loss: 0.0913
Epoch 26/100
201/201 [==============================] - 1s 3ms/step - loss: 0.0924
Epoch 27/100
201/201 [==============================] - 1s 3ms/step - loss: 0.0913
Epoch 28/100
201/201 [==============================] - 1s 4ms/step - loss: 0.0913
Epoch 29/100
201/201 [==============================] - 1s 3ms/step - loss: 0.0915
```

Now using cross validation

```
from sklearn.model_selection import KFold

# Define number of folds for cross-validation
num_folds = 5
kf = KFold(n_splits=num_folds)

# Initialize a list to store accuracy scores for each fold
accuracy_scores = []

# Perform k-fold cross-validation
for train_index, val_index in kf.split(X_train):
```

```
for train_index, val_index in kf.split(X_train):
    X_train_fold, X_val_fold = X_train[train_index], X_train[val_index]
    y_train_fold, y_val_fold = y_train[train_index], y_train[val_index]

    # Create and train a new model for each fold
    cv_model = Sequential()
    cv_model.add(Dense(units=5, input_dim=7, kernel_initializer='normal', activation='relu'))
    cv_model.add(Dense(units=5, kernel_initializer='normal', activation='relu'))
    cv_model.add(Dense(1, kernel_initializer='normal'))
    cv_model.compile(loss='mean_squared_error', optimizer='adam')

    # Fit the model on the training data for this fold
    cv_model.fit(X_train_fold, y_train_fold, batch_size=best_batch_size, epochs=best_epochs, verbose=0)

    # Evaluate the model on the validation data for this fold
    val_loss = cv_model.evaluate(X_val_fold, y_val_fold)
    val_predictions = cv_model.predict(X_val_fold)
    MAPE_val = np.mean(100 * (np.abs(y_val_fold - val_predictions) / y_val_fold))
    accuracy_val = 100 - MAPE_val
    accuracy_scores.append(accuracy_val)

# Calculate the average accuracy across all folds
average_accuracy = np.mean(accuracy_scores)
print("Average Cross-Validation Accuracy:", average_accuracy)
```