Implementation and Evaluation of a Multi-Layer Perceptron (MLP)

Muneeb Hassan
22p9199

May 7, 2025

1 Introduction

This report describes the implementation of a Multi-Layer Perceptron (MLP) to predict house prices using the Boston housing dataset. The process includes data preprocessing, model building, and performance evaluation of different network architectures.

2 Data Preprocessing

The dataset is loaded and checked for missing values. Missing numeric values are filled with the mean, and all features are normalized to the range [-1, 1] using the MaxAbsScaler.

Data Loading and Normalization

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
from sklearn.preprocessing import MinMaxScaler, RobustScaler, MaxAbsScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam

# Loading the data
df = pd.read_csv('Boston.csv')
df = df.fillna(df.mean(numeric_only=True))  # handle missing
scale = MaxAbsScaler()
normalized_df = scale.fit_transform(df)
new_df = pd.DataFrame(normalized_df, columns=df.columns)
```

3 Model Architecture

The MLP model is constructed using Keras Sequential API. Different hidden layer configurations are tested.

MLP Model Definition

```
def buildMLP(hidden_layers, input_shape):
    model = Sequential()
    model.add(Dense(units=hidden_layers[0], activation='relu', input_shape=input
    for units in hidden_layers[1:]:
        model.add(Dense(units, activation='relu'))
    model.add(Dense(1))  # output layer
    model.compile(optimizer=Adam(learning_rate=0.001), loss='mse', metrics=['mae
    return model
```

4 Experiments

The model is trained using 80/20 train/test split and evaluated on test data.

Training Loop for Different Architectures

```
architectures = {
    "1-layer-(16-units)": [16],
    "2-layer-(32,-16)": [32, 16],
    "3-layer-(64,-32,-16)": [64, 32, 16],
    "4-layer-(128,-64,-32,-16)": [128, 64, 32, 16]
}

for name, layers in architectures.items():
    print(f"\nTraining-{name}-architecture...")
    model = buildMLP(hidden_layers=layers, input_shape=(X_train.shape[1],))
    history = model.fit(
        X_train, y_train,
        validation_split=0.2,
        epochs=100,
        batch_size=28,
        verbose=0
    )
    y_pred = model.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    print(f"{name}---Test-MSE:-{mse:.4f}")
```

5 Results

The test MSE (Mean Squared Error) for each architecture is reported below:

- **1-layer (16 units)**: Test MSE = *reported value*
- **2-layer (32, 16)**: Test MSE = *reported value*
- **3-layer (64, 32, 16)**: Test MSE = *reported value*
- **4-layer (128, 64, 32, 16)**: Test MSE = *reported value*

6 Conclusion

The MLP was implemented successfully with various architectures. Performance improved with deeper architectures up to a point, balancing complexity and generalization.