ChatGPT

Boston Housing Price Prediction Code Explained

This code builds a neural network with TensorFlow/Keras to predict house prices from the Boston housing dataset. The steps include importing libraries, loading and cleaning data, preprocessing features, building and training the neural network, and evaluating its performance. The explanation below walks through each part of the code, describing what it does and why.

1. Importing Libraries

The first lines of code import the necessary libraries:

- `pandas as pd`: Pandas is used for data manipulation and analysis. It provides the `read_csv()` function to load data from a CSV file into a DataFrame [1].
- `numpy as np`: NumPy offers support for large, multi-dimensional arrays and matrices, along with mathematical functions. It is often used for numerical operations on the data.
- `tensorflow as tf` and `from tensorflow import keras`: TensorFlow (and its high-level Keras API) are used to build and train the neural network model.
- `matplotlib.pyplot as plt`: Matplotlib is a plotting library. `plt` will be used to create visualizations such as loss curves and scatter plots.
- `sklearn.preprocessing.StandardScaler`: This tool standardizes features by removing the mean and scaling to unit variance [2], which is important for many machine learning algorithms.
- `sklearn.model_selection.train_test_split`: This function splits the data into training and testing sets in a randomized way [3].
- `sklearn.metrics`: The functions `mean_squared_error`, `mean_absolute_error`, and `r2_score` are imported to evaluate the model's performance after training.

Importing these libraries sets up the tools needed for data loading, preparation, model building, training, and evaluation.

2. Loading the Dataset

```
df = pd.read_csv('boston_housing.csv')
```

This line uses Pandas' `read_csv()` function to load the Boston housing data from a CSV file into a DataFrame. A DataFrame is like a table where columns are features and rows are individual samples (houses) [1]. The Boston dataset typically contains 13 features (e.g. crime rate, number of rooms) and a target column which is the median house price (often in \$1000s or similar). By reading the CSV, the code now has a structured table `df` with all the data for analysis.

3. Data Cleaning and Preprocessing

The code may include steps like:

```
df = df.drop(['ID'], axis=1)
df = df.dropna()
```

- **Dropping columns**: If the dataset has an irrelevant column (for example, an `ID` or index column), it's common to drop it. The `drop()` method with `axis=1` removes column(s) by name.
- **Handling missing values**: `dropna()` removes any rows that contain missing values (NaNs). This ensures the model only sees complete data. Handling missing data is crucial because many algorithms (including neural networks) cannot process NaNs properly. By dropping or filling missing entries, the code makes the dataset clean.

After cleaning, `df` contains only the relevant numeric columns (no NaNs or extraneous ID column), ready for analysis and modeling.

4. Exploratory Analysis: Correlation

```
corr_matrix = df.corr()
```

This computes a **correlation matrix** for the DataFrame. The `corr()` method finds the pairwise correlation between columns (features and target) [4]. By default, this is the Pearson correlation, which measures linear relationships. The resulting `corr_matrix` shows how strongly each feature is linearly related to every other feature (and to the target). For example, a high positive correlation between a feature and the house price suggests that when the feature increases, the price also tends to increase. This analysis can help identify which features might be most relevant to predict the target. The correlation matrix excludes any missing values in the computation [4].

5. Defining Features and Labels

```
X = df.drop('PRICE', axis=1)
y = df['PRICE']
```

Here the code separates the data into **features** (`X`) and **labels/target** (`y`).

- `X` (features) contains all the independent variables that the model will use to predict the house price. This is typically done by dropping the target column (`'PRICE'` or whatever the target is named) from `df`.
- `y` (label) contains the dependent variable (the target we want to predict), which is the house price.

In machine learning terms, features are the input data (the predictors), and the label is the output (the value to predict). Splitting into `X` and `y` prepares the data for training the model. A simple example line

could be: `X = df.drop('MEDV', axis=1); y = df['MEDV']` if the target column is named `'MEDV'` (Median Value). The exact column names depend on the dataset used.

6. Splitting into Training and Test Sets

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

This line uses Scikit-learn's `train_test_split()` function to randomly divide the data into training and testing subsets [3]. Here, `test_size=0.2` means 20% of the data is held out for testing, and 80% is used for training. The `random_state=42` sets a seed so that the split is reproducible.

- **Training set (`X_train`, `y_train`)**: used to train (fit) the model.
- **Test set (`X_test`, `y_test`)**: used to evaluate the model on unseen data.

Splitting ensures that after training, we can assess the model's performance on data it hasn't seen, which is crucial for checking how well the model generalizes to new cases [3].

7. Feature Scaling (Standardization)

```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

This block standardizes the feature values using `StandardScaler`:

- `StandardScaler()` creates an object that will compute the mean and standard deviation of each feature from the training data.
- `scaler.fit_transform(X_train)` fits the scaler on the training features (computes means and variances) and then transforms the training data by subtracting the mean and dividing by the standard deviation for each feature.
- `scaler.transform(X_test)` uses the same transformation (same means and variances) on the test data.

Standardization makes each feature have **zero mean and unit variance**, which is important because many machine learning algorithms work best when features are on similar scales [2]. According to the Scikit-learn documentation, "StandardScaler" *"Standardize[s] features by removing the mean and scaling to unit variance."* [2]. This prevents features with larger scales from dominating the learning process and helps the neural network train more efficiently.

8. Building the Neural Network Model

```
model = keras.Sequential([
    keras.layers.Dense(64, activation='relu', input_shape=(X_train_scaled.shape[1],)),
```

```
    keras.layers.Dense(64, activation='relu'),
    keras.layers.Dense(1)
])
```

This code defines the neural network architecture using Keras' **Sequential** API [5]. A Sequential model is a linear stack of layers [5]. The layers here are:

- **First Dense layer**: `Dense(64, activation='relu', input_shape=(n,))`.
- **Units**: 64 neurons.
- **Activation**: ReLU (Rectified Linear Unit). ReLU is a piecewise linear function that outputs the input directly if it is positive, otherwise outputs zero [6]. It introduces nonlinearity, allowing the network to learn complex patterns [6].
- **Input shape**: `(X_train_scaled.shape[1],)` specifies the number of input features (`n`). This tells the model how many inputs to expect.
- **Second Dense layer**: `Dense(64, activation='relu')`. Another hidden layer with 64 neurons and ReLU activation. Having multiple hidden layers can allow the model to learn more complex functions by stacking nonlinear transformations.
- **Output layer**: `Dense(1)`. A single neuron with no activation (default linear). For a regression task like price prediction, we use one output neuron that produces a continuous value (the predicted price). A linear activation is implicit here (no `activation` argument means linear).

In summary, this network has two hidden layers with ReLU activations and one output layer. The ReLU activation is popular in deep learning because it is simple and helps mitigate issues like vanishing gradients [6]. The output layer has a single neuron, since we predict one continuous target (house price).

9. Compiling the Model

```
model.compile(optimizer='adam', loss='mse', metrics=['mae'])
```

Before training, the model is **compiled** by specifying how it should learn:

- **Optimizer**: `'adam'`. Adam stands for Adaptive Moment Estimation. It is a stochastic gradient descent method known for efficiency and good convergence in many problems. Adam automatically adjusts the learning rate during training.
- **Loss function**: `'mse'` (Mean Squared Error). This is common for regression tasks. MSE computes the average of the squared differences between predicted and true values [7]. Lower MSE means predictions are closer to actual values. According to documentation, "Mean squared error [is a] regression loss" [7].
- **Metrics**: `['mae']` means the training process will also track Mean Absolute Error (MAE). MAE is the average absolute difference between predictions and true values, also a regression loss [8].

So, the model will train by minimizing MSE using the Adam optimizer, and it will report MAE as an additional performance measure. MSE punishes larger errors more strongly due to the squaring, while MAE is more interpretable as "average error in original units" [8].

10. Training the Model

```
history = model.fit(X_train_scaled, y_train, validation_split=0.2, epochs=100, batch_size=32)
```

This line trains the neural network on the scaled training data:

- `X_train_scaled`, `y_train` are the input features and targets for training.
- `validation_split=0.2` means 20% of the training data is set aside as a validation set. During training, the model will report loss on both the training data and this held-out validation set.
- `epochs=100` specifies that the entire training set will be passed through the network 100 times. Each full pass is one epoch.
- `batch_size=32` means the training data is processed in small groups of 32 samples at a time; gradients are updated after each batch.

The `fit()` function returns a `history` object that contains the loss values for each epoch. Specifically, `history.history['loss']` holds the training loss (MSE) and `history.history['val_loss']` holds the validation loss. These can be plotted to check learning.

Visualizing Training and Validation Loss

After training, it is common to plot the loss over epochs:

```
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.legend()
plt.show()
```

- **Training loss curve**: shows how the loss on the training set decreases over epochs.
- **Validation loss curve**: shows how the loss on unseen validation data changes.

Typically, both should decrease initially. If the validation loss starts to rise while training loss continues to drop, it may indicate **overfitting** (the model is learning patterns too specific to the training data). Plotting these curves helps diagnose learning behavior [9]. In deep learning, monitoring both metrics is crucial because "training loss refers to the error on the data the model was trained on" and "validation loss is the error on unseen data" [9]. A good model will have both losses decreasing and close to each other by the end of training.

11. Evaluating the Model

After training, the code evaluates performance on the test set:

```
y_pred = model.predict(X_test_scaled)
mse = mean_squared_error(y_test, y_pred)
```

```
mae = mean_absolute_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
```

- `model.predict(X_test_scaled)` generates predicted prices for the test features.
- `mean_squared_error(y_test, y_pred)` computes the MSE on test data [7].
- `mean_absolute_error(y_test, y_pred)` computes the MAE [8].
- `r2_score(y_test, y_pred)` computes the R^2 (coefficient of determination).

Interpreting metrics: - **MSE**: A lower MSE indicates predictions are, on average, closer to actual values (since errors are squared, it heavily penalizes large errors) [7]. - **MAE**: The average absolute difference between predictions and true values. Also, lower is better. (This gives error in the same units as the target) [8]. - **R^2 score**: Measures how well the predictions explain the variance in the data [10]. The best possible R^2 is 1.0, meaning perfect prediction. An R^2 of 0 means the model does no better than predicting the average of the target. R^2 can also be negative if the model is worse than predicting the average [10].

By checking these metrics, we see quantitatively how accurate the model's predictions are. For a good model, MSE and MAE should be reasonably small, and R^2 close to 1.

12. Visualizing Predictions vs. Actual

```
plt.scatter(y_test, y_pred)
plt.xlabel('Actual Prices')
plt.ylabel('Predicted Prices')
plt.title('Actual vs Predicted Prices')
plt.show()
```

This scatter plot compares the actual target values (`y_test`) to the predicted values (`y_pred`). Each point represents one test sample's true price (x-axis) versus predicted price (y-axis).

Interpreting the scatter plot: Ideally, all points lie near the diagonal line `y = x`. If predictions were perfect, every point would be on that line. As experts note, "Scatter plots of Actual vs Predicted are one of the richest forms of data visualization... Ideally, all your points should be close to a regressed diagonal line. If your model had a high R^2, all the points would be close to this diagonal line" [11].

Thus, by examining this plot, you can visually assess how well the model is performing across the range of actual prices. If points deviate far from the diagonal, it indicates prediction errors. A cluster of points below or above the line in certain regions may suggest systematic under- or over-prediction in that range.

Each section above corresponds to a part of the code. By walking through imports, data loading, cleaning, model building, training, and evaluation, we see how the Boston housing price prediction is implemented step by step. The key ideas—such as separating features and labels, scaling data, building layers in a neural network, choosing loss functions, and interpreting metrics and plots—are explained in simple terms. This should help readers new to machine learning understand both *what* the code does and *why* each step is taken.

Sources: The explanations above draw on official documentation and tutorials for the tools and concepts used. For example, Pandas and Scikit-learn documentation explain functions like `read_csv`, `corr`, `train_test_split`, and metrics [1] [4] [3] [7] [8] [10]. Keras documentation describes model structures like Sequential models and layers [5] [12]. The interpretations of loss curves and prediction scatter plots are based on standard explanations of model training behavior and visual diagnostics [9] [11].

---

1. **Pandas Read CSV in Python | GeeksforGeeks**
   https://www.geeksforgeeks.org/python-read-csv-using-pandas-read_csv/
2. **StandardScaler — scikit-learn 1.6.1 documentation**
   https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html
3. **train_test_split — scikit-learn 1.6.1 documentation**
   https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html
4. **pandas.DataFrame.corr — pandas 2.2.3 documentation**
   https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.corr.html
5. **tf.keras.Sequential | TensorFlow v2.16.1**
   https://www.tensorflow.org/api_docs/python/tf/keras/Sequential
6. **ReLU Activation Function in Deep Learning | GeeksforGeeks**
   https://www.geeksforgeeks.org/relu-activation-function-in-deep-learning/
7. **mean_squared_error — scikit-learn 1.6.1 documentation**
   https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_squared_error.html
8. **mean_absolute_error — scikit-learn 1.6.1 documentation**
   https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_absolute_error.html
9. **Training and Validation Loss in Deep Learning | GeeksforGeeks**
   https://www.geeksforgeeks.org/training-and-validation-loss-in-deep-learning/
10. **r2_score — scikit-learn 1.6.1 documentation**
    https://scikit-learn.org/stable/modules/generated/sklearn.metrics.r2_score.html
11. **r - what does an actual vs fitted graph tell us? - Cross Validated**
    https://stats.stackexchange.com/questions/104622/what-does-an-actual-vs-fitted-graph-tell-us
12. **Dense layer**
    https://keras.io/api/layers/core_layers/dense/