Practical-

Linear regression by using Deep Neural network: Implement Boston housing price prediction problem by Linear regression using Deep Neural network. Use Boston House price prediction dataset.

Cell 2: Code - Importing Libraries

```
Python

import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

```

- This cell imports several important Python libraries commonly used in data science and machine learning:
    - `tensorflow`: A powerful open-source library developed by Google for building and training machine learning models, especially deep neural networks. The `as tf` alias lets you refer to it concisely as `tf` in the rest of your code.
    - `sklearn.model_selection`: Specifically, the `train_test_split` function from scikit-learn (sklearn) is imported. This function is essential for dividing your dataset into training and testing sets, allowing you to evaluate your model's performance on unseen data.
    - `sklearn.preprocessing`: The `StandardScaler` is imported from scikit-learn. This is used for feature scaling, a crucial preprocessing step in machine learning to standardize the range of your data.

Feature scaling is a technique used in data preprocessing to standardize or normalize the range of independent variables (features) in a dataset. It ensures that all features contribute equally to a model’s performance, especially those that are sensitive to the scale of input data.

Without scaling:

- A feature with a large range (e.g., salary in lakhs) can dominate a feature with a smaller range (e.g., age in years).

- The model may become biased toward certain features.

    - `pandas`: A library providing high-performance, easy-to-use data structures (like DataFrames) and data analysis tools. It's excellent for working with structured data, such as CSV files.
    - `numpy`: The fundamental library for numerical computing in Python. It supports arrays, matrices, and mathematical functions.
    - `matplotlib.pyplot`: A plotting library for creating visualizations in Python. The `as plt` alias is standard.
    - `seaborn`: Another data visualization library built on top of matplotlib. It provides a higher-level interface for creating more aesthetically pleasing and informative statistical[1] graphics.
- The `stderr` and `stdout` messages are related to library versions and TensorFlow deprecation warnings. They don't necessarily indicate errors in your code but are important to be aware of.

Cell 3: Code - Loading Data

```
Python

data = pd.read_csv('HousingData.csv')

data
```

- This cell uses pandas to read a CSV file named "HousingData.csv" into a DataFrame called `data`.
- The `data` variable now holds your dataset.
- The `data` by itself at the end of the cell displays the first few and last few rows of the DataFrame, along with the column names and data types. This gives you a quick look at your data.

Cell 4: Code - Checking for Missing Values

```
Python

data.isnull().sum()
```

- This cell checks for missing values in your dataset.
    - `data.isnull()`: This part returns a DataFrame of the same shape as `data`, where each element is `True` if the corresponding element in `data` is missing (NaN - Not a Number) and `False` otherwise.
    - `.sum()`: This sums the `True` values along each column. Since `True` is treated as 1 and `False` as 0, the result is a Series showing the number of missing values in each column.

Cell 5: Code - Handling Missing Values

```
Python

data = data.dropna()
```

- This cell handles the missing values by removing rows that contain any missing values.
    - `data.dropna()`: This method returns a new DataFrame with the rows containing missing values removed. The changes are then assigned back to the `data` variable, updating the DataFrame to only include complete rows.

Cell 6:- Heatmap

This heatmap visually represents the pairwise linear correlations between the different features (columns) in your housing dataset.

- **Axes: The x-axis and y-axis both list the names of the features in your dataset (CRIM, ZN, INDUS, CHAS, NOX, RM, AGE, DIS, RAD, TAX, PTRATIO, B, LSTAT, MEDV).**
- **Cells: Each cell in the heatmap represents the correlation coefficient between the feature on the y-axis and the feature on the x-axis.**

- **Color Intensity: The color intensity of each cell indicates the strength of the correlation.**
    - **Darker shades (in this case, towards the blue end of the spectrum): Indicate a stronger positive correlation. This means that as one feature increases, the other tends to increase as well.**
    - **Darker shades of the opposite color (towards the green end of the spectrum): Indicate a stronger negative correlation. This means that as one feature increases, the other tends to decrease.**
    - **Lighter shades (closer to the middle color): Indicate a weak or near-zero correlation, meaning there's little linear relationship between the two features.**
- **Numerical Values: The numerical value within each cell is the actual correlation coefficient, ranging from -1 to +1.**
    - **+1: Perfect positive correlation.**
    - **-1: Perfect negative correlation.**
    - **0: No linear correlation.**
- **Color Bar: The color bar on the right provides a visual scale for interpreting the correlation coefficients.**
- **`annot=True`: This parameter in the `sns.heatmap()` function ensures that the correlation coefficient values are displayed within each cell of the heatmap.**
- **`cmap='crest'`: This specifies the color map used for the heatmap. "Crest" is a sequential colormap that transitions from dark to light.**

Interpreting Key Correlations in Your Housing Data

Based on the heatmap, here are some notable correlations:

- **Strong Positive Correlations:**
    - **RAD and TAX (0.91): The index of accessibility to radial highways and the full-value property tax rate per $10,000 are very strongly positively correlated. This suggests that areas with better highway access tend to have higher property taxes.**
    - **INDUS and NOX (0.76): The proportion of non-retail business acres per town and the nitric oxides concentration (parts per 10 million) show a strong positive correlation. This makes sense as industrial areas are likely to have higher levels of nitrogen oxides.**
    - **LSTAT and CRIM (0.46): The percentage of lower status of the population has a moderate positive correlation with the per capita crime rate by town. This might suggest that areas with a higher**

proportion of lower-status individuals tend to have higher crime rates.

- **AGE and NOX (0.73): The proportion of owner-occupied units built prior to 1940 has a strong positive correlation with nitric oxides concentration. Older buildings might be located in more industrialized areas or have less efficient heating systems.**

- **Strong Negative Correlations:**

  - **LSTAT and MEDV (-0.74): The percentage of lower status of the population has a strong negative correlation with the median value of owner-occupied homes in $1000s. This is a common finding – as the proportion of lower-status individuals in an area increases, the median house value tends to decrease.**
  - **RM and MEDV (0.72): The average number of rooms per dwelling has a strong positive correlation with the median value of owner-occupied homes. Larger houses (more rooms) tend to be more expensive.**
  - **DIS and INDUS (-0.77): The weighted distances to five Boston employment centres have a strong negative correlation with the proportion of non-retail business acres per town. Areas closer to employment centers tend to have less industrial land.**
  - **DIS and NOX (-0.77): Similarly, distance to employment centers has a strong negative correlation with nitric oxides concentration, likely because industrial activity is concentrated closer to these centers.**

Understanding the Pair Plot

A pair plot is a powerful tool for visualizing the pairwise relationships between multiple variables in a dataset. It creates a grid of plots where:

- **Diagonal: The diagonal plots are typically histograms (or kernel density estimates) showing the distribution of each individual variable. This helps you understand the spread and shape of the data for each feature.**
- **Off-Diagonal: The off-diagonal plots are scatter plots showing the relationship between each pair of variables. Each point in the scatter plot represents a single data instance, and its position is determined by its values for the two corresponding features.**

Analyzing the Pair Plot for 'RM', 'DIS', 'LSTAT', and 'MEDV'

Let's examine the individual plots and the scatter plots to understand the relationships:

1. Histograms (Diagonal)

- **RM (Average number of rooms per dwelling): The histogram shows a roughly normal distribution, perhaps slightly skewed to the right, indicating that most houses have around 5 to 7 rooms, with fewer houses having significantly more or fewer rooms.**
- **DIS (Weighted distances to five Boston employment centres): The histogram appears skewed to the right, suggesting that there are more houses located at larger distances from the employment centers, with a decreasing number of houses as the distance decreases.**
- **LSTAT (Percentage of lower status of the population): The histogram is skewed to the right, indicating that a larger proportion of the census tracts have a lower percentage of lower-status individuals, with a decreasing number of tracts as this percentage increases.**
- **MEDV (Median value of owner-occupied homes in $1000s): The histogram shows a distribution that is somewhat bell-shaped but might have a slight positive skew and potentially some outliers at the higher end. The median house value seems to be concentrated around the $20,000 to $30,000 range.**

2. Scatter Plots (Off-Diagonal)

- **RM vs. MEDV (Top Right and Bottom Left): This scatter plot shows a clear positive trend. As the average number of rooms (RM) increases, the median**

house value (MEDV) tends to increase as well. This aligns with our earlier observation from the correlation heatmap.

- **LSTAT vs. MEDV (Middle Right and Bottom Left): This scatter plot shows a strong negative trend. As the percentage of lower-status population (LSTAT) increases, the median house value (MEDV) tends to decrease. This also matches the strong negative correlation we saw before. The relationship appears somewhat non-linear, possibly curving downwards at higher LSTAT values.**
- **DIS vs. MEDV (Top Right of the 'DIS' row, Bottom Left of the 'DIS' column): This scatter plot shows a positive trend, although perhaps not as strong as the relationship between RM and MEDV. Generally, as the distance to employment centers (DIS) increases, the median house value (MEDV) tends to increase. This could suggest that houses further from the city center might be larger or located in more suburban, potentially more expensive, areas.**
- **RM vs. LSTAT (Middle Left and Top Right): This scatter plot shows a negative trend. As the average number of rooms (RM) increases, the percentage of lower-status population (LSTAT) tends to decrease. Larger houses might be less common in areas with a higher proportion of lower-status individuals.**

Training:-
Okay, I see the code and the output from training a neural network. Let's break it down step by step.

Cell 9: Data Preparation

```
Python

X = data[['RM', 'DIS', 'LSTAT']]
y = data['MEDV']
```

- **Feature Selection:** This line selects three features from your `data` DataFrame and assigns them to the variable `X`. These features are:
    - `'RM'`: Average number of rooms per dwelling.
    - `'DIS'`: Weighted distances to five Boston employment centres.
    - `'LSTAT'`: Percentage of lower status of the population.
    - `X` will now be a DataFrame containing only these three columns, which will be used as the input features for your model.

- **Target Variable Selection:** This line selects the 'MEDV' column (Median value of owner-occupied homes in $1000s) from your `data` DataFrame and assigns it to the variable `y`. This is the target variable that your model will learn to predict.

Cell 10: Train-Test Split

```
Python

train_X, test_X, train_y, test_y = train_test_split(X, y, test_size=0.2, random_state=42)
```

- **Splitting the Data:** This line uses the `train_test_split` function from scikit-learn to divide your dataset into training and testing sets.
    - `X` and `y`: These are the input features and the target variable, respectively.
    - `test_size=0.2`: This specifies that 20% of your data will be used for the test set, and the remaining 80% will be used for the training set.
    - `random_state=42`: This sets a seed for the random number generator. Using a fixed `random_state` ensures that the data is split in the same way each time you run the code, which is important for reproducibility.
- **Resulting Sets:** This line creates four new DataFrames/Series:
    - `train_X`: The features for the training set.
    - `test_X`: The features for the test set.
    - `train_y`: The target variable for the training set.
    - `test_y`: The target variable for the test set.

Cell 11: Feature Scaling

```
Python

scaler = StandardScaler()
train_X_scaled = scaler.fit_transform(train_X)
test_X_scaled = scaler.transform(test_X)
```

- **Initializing the Scaler:** This line creates an instance of the `StandardScaler` from scikit-learn. The `StandardScaler` standardizes features by removing the mean and scaling to unit variance. This is a common preprocessing step[1] for neural networks as it can help the model converge faster and improve performance.
- **Fitting and Transforming the Training Data:** `scaler.fit_transform(train_X)` first calculates the mean and standard deviation of each feature in the `train_X` dataset

and then applies the scaling transformation. The result, `train_X_scaled`, is a NumPy array containing the scaled training features.

- **Transforming the Test Data:** `scaler.transform(test_X)` applies the *same* scaling transformation (using the mean and standard deviation learned from the training data) to the `test_X` dataset. It's crucial to use the parameters learned from the training data to scale the test data to avoid data leakage. `test_X_scaled` is a NumPy array containing the scaled test features.

Cell 12: Building the Neural Network Model

```
Python

model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(3,)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1)
])
```

- **Creating a Sequential Model:** This line creates a sequential neural network model using TensorFlow's Keras API. `tf.keras.Sequential` means the layers are stacked linearly.
- **Input Layer:** `tf.keras.layers.Input(shape=(3,))` defines the input layer of the network. `shape=(3,)` specifies that the model expects input samples with 3 features (corresponding to 'RM', 'DIS', and 'LSTAT').
- **Hidden Layer 1:** `tf.keras.layers.Dense(128, activation='relu')` adds a dense (fully connected) layer with 128 neurons (units). The `relu` (Rectified Linear Unit) activation function is applied to the outputs of this layer. ReLU is a common activation function that introduces non-linearity, allowing the model to learn complex relationships.
- **Hidden Layer 2:** `tf.keras.layers.Dense(64, activation='relu')` adds another dense layer with 64 neurons and ReLU activation. Stacking multiple hidden layers allows the model to learn hierarchical representations of the data.
- **Output Layer:** `tf.keras.layers.Dense(1)` adds the output layer. Since this is likely a regression task (predicting a continuous value - house price), the output layer has only one neuron and typically no activation function is specified (or a linear activation is implicitly used).

Cell 13: Early Stopping Callback

```
Python
```

```
early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', min_delta=0.001, patience=3, restore_best_weights=True)
```

- **Creating an Early Stopping Callback:** This line creates an `EarlyStopping` callback. This callback is used during training to monitor a specific metric (in this case, `val_loss`, the loss on the validation set) and stop the training process early if the metric stops improving. This helps prevent overfitting.
    - `monitor='val_loss'`: Specifies the metric to monitor.
    - `min_delta=0.001`: The minimum change in the monitored metric to qualify as an improvement. Changes smaller than this are considered no improvement.
    - `patience=3`: The number of epochs with no improvement after which training will be stopped. If the validation loss doesn't decrease by at least `min_delta` for 3 consecutive epochs, training will halt.
    - `restore_best_weights=True`: If training is stopped early, this will restore the model's weights to the values it had at the epoch with the best (lowest) validation loss.

Cell 14: Compiling and Training the Model

```
Python

model.compile(optimizer='adam', loss='mse', metrics=['mse'])
history = model.fit(train_X_scaled, train_y, validation_data=(test_X_scaled, test_y), epochs=100, callbacks=[early_stopping])
```

- **Compiling the Model:** This line configures the model for training.
    - `optimizer='adam'`: Specifies the optimization algorithm to be used. Adam is a popular and efficient gradient-based optimization algorithm.
    - `loss='mse'`: Specifies the loss function to be minimized during training. Mean Squared Error (MSE) is a common loss function for regression[2] tasks, measuring the average squared difference between the predicted and actual values.[3]
    - `metrics=['mse']`: Specifies the evaluation metric to be monitored during training and testing. In this case, it's also MSE. You could include other metrics like Mean Absolute Error ('mae') as well.
- **Training the Model:** `model.fit()` starts the training process.
    - `train_X_scaled`: The scaled training features.

    - `train_y`: The training target variable.
    - `validation_data=(test_X_scaled, test_y)`: Provides the scaled test features and test target variable to be used as the validation set during training. The model's performance on this unseen data is monitored to detect overfitting and for the early stopping callback.
    - `epochs=100`: The maximum number of training epochs (passes through the entire training dataset). The training might stop earlier if the `early_stopping` callback is triggered.
    - `callbacks=[early_stopping]`: Passes the `early_stopping` callback to the `fit` method, so it can monitor the validation loss and potentially stop training early.

Output of Training

The output shows the progress of the training for each epoch. You can see:

- **Epoch Number (e.g., 1/100, 2/100):** Indicates the current training epoch and the total number of epochs set.
- **Time per Step:** The time taken for each training step (processing a batch of data).
- **loss:** The mean squared error on the training data for that epoch.
- **mse:** The mean squared error on the training data for that epoch (same as `loss` in this case).
- **val_loss:** The mean squared error on the validation data for that epoch. This is the crucial metric being monitored by the `early_stopping` callback.
- **val_mse:** The mean squared error on the validation data for that epoch (same as `val_loss`).

Notice that the training stopped after epoch 15 (15/100). This is because the `early_stopping` callback detected that the `val_loss` was no longer improving significantly for the specified `patience` of 3 epochs. The `restore_best_weights=True` setting would have ensured that the model's weights are now set to those that resulted in the lowest `val_loss` during training (likely around epoch 12-13).

In summary, this code performs the essential steps for training a neural network for a regression task: preparing the data, splitting it into training and testing sets, scaling the features, building the model, and training it with early stopping to prevent overfitting. The output shows the training progress and indicates that the training was stopped early based on the validation loss.