**Problem Statement:** 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.

Let's break down this Python code step by step. It's a script that performs a linear regression task using TensorFlow to predict housing prices.

* import tensorflow as tf: This line imports the TensorFlow library, which is a powerful open-source library for numerical computation and large-scale machine learning. We're giving it the alias tf for easier use later in the code.
* from sklearn.model\_selection import train\_test\_split: This line imports a specific function, train\_test\_split, from the model\_selection module of the scikit-learn (often abbreviated as sklearn) library. This function is used to split a dataset into training and testing sets.
* from sklearn.preprocessing import StandardScaler: This line imports the StandardScaler class from the preprocessing module of scikit-learn. This class is used for standardizing numerical features by removing the mean and scaling to unit variance.
* import pandas as pd: This line imports the pandas library, which provides data structures like DataFrames that are excellent for working with tabular data. We're giving it the alias pd.
* import numpy as np: This line imports the NumPy library, which is fundamental for numerical operations in Python. It provides support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. We're giving it the alias np.
* import matplotlib.pyplot as plt: This line imports the pyplot module from the matplotlib library, which is used for creating static, interactive, and animated visualizations in Python. We're giving it the alias plt.
* import seaborn as sns: This line imports the seaborn library, which is built on top of matplotlib and provides a high-level interface for drawing attractive and informative statistical graphics. We're giving it the alias sns.
* data = pd.read\_csv('HousingData.csv'): This line uses the read\_csv function from the pandas library to read data from a CSV (Comma Separated Values) file named 'HousingData.csv' and stores it in a pandas DataFrame called data.
* data: This line, when executed in an interactive environment like a Jupyter Notebook or a Python interpreter, will display the data DataFrame. This allows you to inspect the loaded data.
* data.isnull().sum(): This line performs two operations:
  + data.isnull(): This method is called on the data DataFrame and returns a new DataFrame of the same shape, where each cell contains a boolean value indicating whether the corresponding value in data is missing (True) or not (False).
  + .sum(): This method is then called on the resulting boolean DataFrame. When applied to boolean values, True is treated as 1 and False as 0. So, this calculates the sum of missing values for each column in the data DataFrame, giving you a count of how many missing values exist in each feature.
* data = data.dropna(): This line uses the dropna() method on the data DataFrame. This method removes rows that contain any missing values (NaN - Not a Number). The result is a new DataFrame (which is then assigned back to the data variable) containing only rows with complete data.
* plt.figure(figsize=(14, 10)): This line creates a new figure using matplotlib's plt.figure() function. The figsize argument is a tuple specifying the width and height of the figure in inches (14 inches wide and 10 inches tall). This sets the size of the plot that will follow.
* sns.heatmap(data.corr(), cmap='crest', annot=True): This line creates a heatmap using the heatmap() function from the seaborn library.
  + data.corr(): This calculates the pairwise correlation between all columns in the data DataFrame. The result is a correlation matrix.
  + cmap='crest': This argument specifies the color map to be used for the heatmap. 'crest' is a specific seaborn color palette.
  + annot=True: This argument instructs seaborn to display the correlation values (annotations) on the heatmap cells. This helps in understanding the strength and direction of the relationships between the features.
* sns.pairplot(data[['RM', 'DIS', 'LSTAT', 'MEDV']]): This line creates a pair plot using the pairplot() function from seaborn. A pair plot visualizes the pairwise relationships between multiple variables.
  + data[['RM', 'DIS', 'LSTAT', 'MEDV']]: This selects only the columns 'RM', 'DIS', 'LSTAT', and 'MEDV' from the data DataFrame. The pair plot will then show scatter plots for each pair of these variables and histograms (or density plots) for the distribution of each individual variable along the diagonal.
* X = data[['RM', 'DIS', 'LSTAT']]: This line selects three specific columns ('RM', 'DIS', 'LSTAT') from the data DataFrame and assigns them to a new variable named X. This is typically done to define the features (independent variables) that will be used to predict the target variable.
* y = data['MEDV']: This line selects the column named 'MEDV' from the data DataFrame and assigns it to a new variable named y. This column is assumed to be the target variable (dependent variable) that we want to predict (likely the median value of owner-occupied homes).
* train\_x, test\_x, train\_y, test\_y = train\_test\_split(X, y, test\_size=0.2): This line uses the train\_test\_split function (imported earlier from scikit-learn) to split the data into training and testing sets.
  + X: The feature data.
  + y: The target data.
  + test\_size=0.2: This argument specifies that 20% of the data should be used for the test set, and the remaining 80% will be used for the training set.
  + The function returns four variables:
    - train\_x: The features for the training set.
    - test\_x: The features for the testing set.
    - train\_y: The target variable for the training set.
    - test\_y: The target variable for the testing set.
* scaler = StandardScaler(): This line creates an instance (an object) of the StandardScaler class that was imported earlier. This scaler object will be used to standardize the features.
* train\_x\_scaled = scaler.fit\_transform(train\_x): This line performs two operations on the training features (train\_x):
  + scaler.fit(train\_x): This calculates the mean and standard deviation of each feature in the train\_x data. These statistics will be used for scaling.
  + scaler.transform(train\_x): This applies the standardization (subtracting the mean and dividing by the standard deviation) to each feature in train\_x using the statistics learned in the fit step. The resulting scaled training features are stored in the train\_x\_scaled variable.
* test\_x\_scaled = scaler.transform(test\_x): This line applies the same standardization (using the mean and standard deviation learned from the *training* data in the previous step) to the testing features (test\_x). It's crucial to use the statistics from the training data to avoid data leakage from the test set. The scaled testing features are stored in the test\_x\_scaled variable.
* model = tf.keras.models.Sequential(): This line creates a sequential model using TensorFlow's Keras API. A sequential model is a linear stack of layers.
* model.add(tf.keras.layers.Input(shape=(3,))): This line adds the input layer to the model.
  + tf.keras.layers.Input(): This creates an input layer.
  + shape=(3,): This specifies the expected shape of the input data. In this case, it indicates that each input sample will have 3 features (corresponding to 'RM', 'DIS', and 'LSTAT'). The , is used to indicate that it's a tuple representing the shape.
* model.add(tf.keras.layers.Dense(128, activation='relu')): This line adds a dense (fully connected) layer with 128 neurons (units) to the model.
  + tf.keras.layers.Dense(128): Creates a dense layer with 128 output units.
  + activation='relu': Specifies the Rectified Linear Unit (ReLU) as the activation function for this layer. ReLU is a common activation function in neural networks.
* model.add(tf.keras.layers.Dense(64, activation='relu')): This line adds another dense layer with 64 neurons and the ReLU activation function.
* model.add(tf.keras.layers.Dense(32, activation='relu')): This line adds another dense layer with 32 neurons and the ReLU activation function.
* model.add(tf.keras.layers.Dense(1, activation='linear')): This line adds the output layer, which is also a dense layer.
  + tf.keras.layers.Dense(1): Creates a dense layer with 1 output unit. Since this is a regression task (predicting a continuous value), we typically have a single output neuron.
  + activation='linear': Specifies the linear activation function. This means the output of this layer will be a direct linear combination of its inputs, which is appropriate for predicting a continuous value.
* early\_stopping = tf.keras.callbacks.EarlyStopping(monitor='val\_loss', min\_delta=0.001, patience=3, restore\_best\_weights=True): This line creates an EarlyStopping callback from TensorFlow Keras. Callbacks are utilities called at certain points during the training process. Early stopping is used to prevent overfitting.
  + monitor='val\_loss': Specifies that the callback should monitor the validation loss.
  + min\_delta=0.001: Sets the minimum change in the monitored quantity to qualify as an improvement. If the validation loss doesn't decrease by at least 0.001 for a certain number of epochs, training will be stopped.
  + patience=3: Specifies the number of epochs with no improvement after which training will be stopped. In this case, if the validation loss doesn't improve for 3 consecutive epochs (by at least min\_delta), training will halt.
  + restore\_best\_weights=True: If set to True, the model's weights will be restored to the values of the epoch that yielded the best value for the monitored quantity (the lowest validation loss in this case).
* model.compile(optimizer='adam', loss='mse', metrics=['mse']): This line configures the model for training.
  + optimizer='adam': Specifies the optimization algorithm to be used for updating the model's weights during training. Adam is a popular and efficient optimization algorithm.
  + loss='mse': Specifies the loss function to be minimized during training. Mean Squared Error (MSE) is a common loss function for regression tasks, calculating the average of the squared differences between the predicted and actual values.
  + metrics=['mse']: Specifies the metrics to be evaluated during training and testing. Here, we're using Mean Squared Error again as a metric to track the model's performance.
* model.fit(train\_x\_scaled, train\_y, validation\_data=(test\_x\_scaled, test\_y), epochs=100, callbacks=[early\_stopping]): This line starts the training process of the model.
  + train\_x\_scaled: The scaled training features.
  + train\_y: The training target variable.
  + validation\_data=(test\_x\_scaled, test\_y): This provides the model with the scaled testing features and the corresponding testing target variable to evaluate its performance on unseen data during training. The validation loss is what the EarlyStopping callback monitors.
  + epochs=100: Specifies the maximum number of times the entire training dataset will be passed through the model during training. However, the training might stop earlier due to the EarlyStopping callback.
  + callbacks=[early\_stopping]: Passes the early\_stopping callback to the fit method, so it can monitor the validation loss and potentially stop training early.

model.evaluate(test\_x\_scaled, test\_y) instructs the previously trained TensorFlow Keras model to assess its performance on the unseen, scaled test dataset (test\_x\_scaled) by comparing its predictions to the actual target values (test\_y). It calculates and returns the loss (Mean Squared Error, as defined during compilation) and any specified metrics (also Mean Squared Error in this case) on this test data, providing an indication of how well the model generalizes to new data. A lower returned value signifies better predictive accuracy on the unseen data.