Deep Learning Practical Prep

lets prepare for my deep learning practical

Great! Let's get started.

To help you best, could you tell me a bit more about your Deep Learning practical?

1. **Topics covered** (e.g., CNNs, RNNs, Autoencoders, Transfer Learning, etc.)
2. **Platform/tools used** (e.g., TensorFlow, Keras, PyTorch, Google Colab, etc.)
3. **Type of practical** – Is it coding-focused (implement models), theory + viva, or both?
4. **Any specific datasets or tasks given?**

Once I have this, I'll create a clear revision and practice plan for you.

what is deep neural network

A **Deep Neural Network (DNN)** is an artificial neural network with **multiple hidden layers** between the input and output layers. These hidden layers allow the network to learn complex patterns in data through **layered abstraction**.

📌 Basic Structure of a Neural Network:

```scss
Input Layer → Hidden Layer 1 → Hidden Layer 2 → ... → Output Layer
```

✅ Key Concepts:

1. **Neurons**: Each unit in a layer. It receives inputs, applies a weight, bias, and an activation function.
2. **Layers**:
   - **Input Layer**: Takes raw data (e.g., pixel values of an image).
   - **Hidden Layers**: Perform feature extraction and pattern learning.
   - **Output Layer**: Produces the final result (e.g., classification).
3. **Weights and Biases**: Learnable parameters that help the network make predictions.
4. **Activation Functions**: Add non-linearity (e.g., ReLU, Sigmoid, Tanh).

🤖 Why "Deep"?

The term **"deep"** refers to the number of **hidden layers**. A **deep** network has **more than one hidden layer**, compared to a **shallow** network which may have only one.

🔁 How It Learns (Training Process):

1. **Forward Propagation**: Input data flows through the network to generate predictions.
2. **Loss Calculation**: Measures error between prediction and actual result.
3. **Backpropagation**: Adjusts weights using gradients to reduce the error.
4. **Optimization**: Done using algorithms like Gradient Descent.

🧠 Applications:

- Image Recognition (e.g., using CNNs)
- Natural Language Processing (e.g., RNNs)
- Recommendation Systems
- Autonomous Vehicles
- Medical Diagnosis

Would you like a simple diagram or example code of a DNN in Keras or PyTorch?

i will also tell you the practicle name
linear regression by using deep neural network: implement boston housing price
Import necessary libraries

import numpy as np # For numerical operations
import pandas as pd # For handling datasets
from sklearn.model_selection import train_test_split # Splitting data into train & test sets
from sklearn.linear_model import LinearRegression # Linear Regression Model
from sklearn.preprocessing import StandardScaler # Standardization of data
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score # Evaluation metrics

Importing Keras (for Neural Network)

import keras
from keras.models import Sequential # To define a sequential model
from keras.layers import Dense # Fully connected layers

Importing Google Colab file handling utility

from google.colab import files

```
# Uploading and Loading Dataset

uploaded = files.upload()  # Opens file upload dialogue in Google Colab
boston = pd.read\_csv("boston\_house\_prices.csv")  # Reads CSV file into a DataFrame

# Selecting Features and Target

# Selecting 3 input features:

# 1. LSTAT (Percentage of lower status population)

# 2. RM (Average number of rooms per dwelling)

# 3. PTRATIO (Pupil-teacher ratio by town)

X = boston\[\['LSTAT', 'RM', 'PTRATIO']]

# Target variable: House Price

y = boston\['PRICE']

# Splitting the Dataset into Training and Testing Sets

# 80% of data used for training, 20% for testing

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=4)

# Standardizing the Dataset (Feature Scaling)

# Standardization improves model performance by normalizing feature values

scaler = StandardScaler()  # Initializing StandardScaler
X\_train\_scaled = scaler.fit\_transform(X\_train)  # Fit and transform training data
X\_test\_scaled = scaler.transform(X\_test)  # Transform test data using the same scaler

# Linear Regression Model

lr\_model = LinearRegression()  # Initializing Linear Regression Model
lr\_model.fit(X\_train\_scaled, y\_train)  # Training the model using scaled training data

# Predicting house prices on test data

y\_pred\_lr = lr\_model.predict(X\_test\_scaled)

# Evaluating Linear Regression Model

mse\_lr = mean\_squared\_error(y\_test, y\_pred\_lr)  # Mean Squared Error
mae\_lr = mean\_absolute\_error(y\_test, y\_pred\_lr)  # Mean Absolute Error
r2\_lr = r2\_score(y\_test, y\_pred\_lr)  # R² Score (Model accuracy measure)

# Displaying evaluation metrics
```

```
print("Linear Regression Model Evaluation:")
print(f"Mean Squared Error: {mse\_lr}")
print(f"Mean Absolute Error: {mae\_lr}")
print(f"R2 Score: {r2\_lr}")

# Neural Network (ANN) Model

# Creating a Deep Learning Model using Keras Sequential API

model = Sequential(\[
Dense(128, activation='relu', input\_dim=3),  # Input layer (3 features) & first hidden layer (128
neurons)
Dense(64, activation='relu'),  # Second hidden layer with 64 neurons
Dense(32, activation='relu'),  # Third hidden layer with 32 neurons
Dense(16, activation='relu'),  # Fourth hidden layer with 16 neurons
Dense(1)  # Output layer (Predicting a single value - House Price)
])

# Compiling the model

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

# Optimizer: Adam (Adaptive Learning Rate Optimization)

# Loss function: Mean Squared Error (MSE) - Suitable for regression problems

# Metric: Mean Absolute Error (MAE) - Helps measure performance

# Training the Neural Network

history = model.fit(X\_train\_scaled, y\_train, epochs=100, validation\_split=0.05, verbose=1)

# Training for 100 epochs

# Using 5% of training data as validation set to monitor overfitting

# `verbose=1` displays detailed training progress

# Evaluating the Neural Network Model

y\_pred\_nn = model.predict(X\_test\_scaled)  # Predicting house prices on test data
mse\_nn, mae\_nn = model.evaluate(X\_test\_scaled, y\_test)  # Evaluating model performance

# Displaying Neural Network Evaluation Metrics

print("\nNeural Network Model Evaluation:")
print(f"Mean Squared Error: {mse\_nn}")
print(f"Mean Absolute Error: {mae\_nn}")

# House Price Prediction for New Data
```

new_data = np.array(\[\[0.1, 10.0, 5.0]])

New input values: LSTAT=0.1, RM=10.0, PTRATIO=5.0

new_data_scaled = scaler.transform(new_data)

Applying the same standardization as training data

Predicting price using trained neural network model

prediction = model.predict(new_data_scaled)

Displaying the predicted house price

print("\nPredicted House Price:", prediction\[0]\[0])

ok explain code and this project

dont explain it rightnow i will ask qn then you answer

Perfect! I'm ready whenever you are. Just ask your questions one by one, and I'll explain each part clearly and simply.

what is linear regression

Linear Regression is a **supervised machine learning algorithm** used for **predicting a continuous numerical value** based on the relationship between **input features (independent variables)** and a **target variable (dependent variable)**.

🔹 Simple Explanation:

It tries to draw a **straight line** (called a **regression line**) through the data points so that it can **predict the target value (Y)** for any given input (X).

🔹 Equation:

$$Y = mX + c$$

For multiple features:

$$Y = w_1X_1 + w_2X_2 + ... + w_nX_n + b$$

Where:

- X_i are input features

- w_i are weights (like slopes)
- b is bias (like y-intercept)
- Y is the predicted output

🔹 Example in Your Project:

You used linear regression to predict **house prices** based on:

- **LSTAT**: % of lower status population
- **RM**: average number of rooms
- **PTRATIO**: pupil-teacher ratio

It finds the best combination of weights w_1, w_2, w_3 so that:

$$\text{PRICE} = w_1 \cdot \text{LSTAT} + w_2 \cdot \text{RM} + w_3 \cdot \text{PTRATIO} + b$$

🔹 Why Use It?

- Simple and interpretable.
- Good baseline model for regression tasks.
- Fast to train and easy to implement.

Would you like to see how it compares with your neural network?

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

from keras.models import Sequential
from keras.layers import Dense
```

This code imports the necessary libraries for implementing both **Linear Regression** and a **Deep Neural Network (DNN)** using Keras for predicting the **Boston housing prices**. Let's go through each import:

📦 Library Imports Breakdown:

1. `import numpy as np`:
    - **NumPy** is used for numerical operations like array handling, mathematical functions, and matrix manipulations.
    - Commonly used for data preprocessing and transformation.

2. `import pandas as pd`:
    - **Pandas** is a powerful library for data manipulation and analysis.
    - It helps load, manipulate, and preprocess datasets (e.g., CSV files, data cleaning, and feature selection).
3. `from sklearn.model_selection import train_test_split`:
    - Splits the dataset into **training** and **testing** sets. This is crucial for evaluating how well the model performs on unseen data.
4. `from sklearn.linear_model import LinearRegression`:
    - Imports the **Linear Regression** model from **scikit-learn**, which you use for predicting the target variable (house prices) based on the input features (LSTAT, RM, PTRATIO).
5. `from sklearn.preprocessing import StandardScaler`:
    - **StandardScaler** normalizes the input features so they have a mean of 0 and a standard deviation of 1.
    - It is important for neural networks and some regression models to work well with scaled data.
6. `from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score`:
    - These are evaluation metrics used to assess the performance of both the **Linear Regression** model and the **Neural Network** model:
        - **Mean Squared Error (MSE)**: Measures the average squared difference between predicted and actual values.
        - **Mean Absolute Error (MAE)**: Measures the average absolute difference.
        - **R^2 Score**: Indicates the proportion of variance in the target variable explained by the model.
7. `from keras.models import Sequential`:
    - **Keras** is a high-level neural network library that simplifies model building and training.
    - **Sequential** allows you to build models layer by layer (stacking layers from input to output).
8. `from keras.layers import Dense`:
    - **Dense** refers to fully connected layers in a neural network.
    - This is used to define the layers of the neural network, where each neuron is connected to every neuron in the next layer.

These imports lay the foundation for building, training, and evaluating both the Linear Regression and Neural Network models for predicting house prices. Let me know if you need clarification on any of these!

X = boston[['LSTAT', 'RM', 'PTRATIO']]

Target variable: House Price

y = boston['PRICE']

from no of column why we only take [['LSTAT', 'RM', 'PTRATIO']] column for input

give correlation code cause external gone ask this queations

Why only select `[['LSTAT', 'RM', 'PTRATIO']]` for input?

In this case, the code selects only three specific features (**LSTAT**, **RM**, **PTRATIO**) from the dataset to predict the house prices. This selection is based on the **domain knowledge** that these features are believed to have a significant influence on the house price.

- **LSTAT**: Percentage of lower status population.
- **RM**: Average number of rooms per dwelling.
- **PTRATIO**: Pupil-teacher ratio by town.

Why not all columns?

- **Feature Selection**: The dataset contains multiple columns, but not all of them may be relevant to the target variable (house prices). Using irrelevant or highly correlated features can cause overfitting or reduce model performance.
- **Correlation**: We usually select features that are **highly correlated** with the target variable and potentially **not highly correlated** with each other.

Correlation Analysis Code

To justify feature selection, you can calculate the **correlation** between features and the target variable, as well as between features themselves.

Here's how you can do it in Python:

```python
import seaborn as sns
import matplotlib.pyplot as plt

# Correlation matrix
correlation_matrix = boston.corr()

# Display the correlation matrix
print(correlation_matrix)

# Visualize the correlation using a heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt='.2f', linewidths=0.5)
plt.show()

# Correlation of features with the target variable 'PRICE'
correlation_with_target = correlation_matrix['PRICE'].sort_values(ascending=False)
print("\nCorrelation with target variable 'PRICE':")
print(correlation_with_target)
```

Explanation:

- **Correlation Matrix**: This shows the correlation coefficients between all pairs of features in the dataset, including the relationship between the target variable `PRICE` and each feature.
- **Heatmap**: A visual representation of the correlation matrix, where:
  - **High positive correlation** (closer to 1) means the variables are strongly related.
  - **High negative correlation** (closer to -1) means they are inversely related.
  - **Low correlation** (around 0) means no significant relationship.
- **Correlation with `PRICE`**: By sorting the correlation values of `PRICE` with other features, you can identify which features are most related to the target variable.

Why This is Important:

By analyzing the correlation, you can identify which features most affect the target variable (house prices) and avoid redundant or irrelevant features, ensuring your model is efficient and interpretable.

Let me know if you need help with the output or further explanation!

PRICE 1.000000
RM 0.695360
ZN 0.360445
B 0.333461
DIS 0.249929
CHAS 0.175260
AGE -0.376955
RAD -0.381626
CRIM -0.388305
NOX -0.427321
TAX -0.468536
INDUS -0.483725
PTRATIO -0.507787
LSTAT -0.737663

this is out now explain how we cj=hoice that 3 input

Great! Based on the **correlation values** you've provided, we can now explain why the features **LSTAT**, **RM**, and **PTRATIO** are chosen as inputs for predicting house prices.

Correlation Analysis with Target (`PRICE`):

- **LSTAT (−0.737)**: This has a **strong negative correlation** with the target variable **PRICE**. This means as the percentage of lower-status population increases, house prices tend to decrease. It's one of the most influential features for predicting house prices.
- **RM (0.695)**: This has a **moderate positive correlation** with **PRICE**. It indicates that as the average number of rooms per dwelling increases, the house price also tends to increase. This feature is also quite important for predicting house prices.

- **PTRATIO (−0.507)**: This has a **moderate negative correlation** with **PRICE**. As the pupil-teacher ratio increases, house prices tend to decrease. This feature also has a relatively significant impact on the price of the house.

Why Choose These 3 Features?

1. **High Correlation with Target**: The primary reason for selecting these three features (**LSTAT**, **RM**, **PTRATIO**) is that they have the **strongest correlations** with the target variable, **PRICE**. Features that are more correlated with the target generally provide more meaningful information to the model, which helps improve prediction accuracy.
2. **Avoiding Redundancy**:
   - **LSTAT** and **RM** have the highest correlations with **PRICE**. Choosing these features ensures that the model focuses on the most relevant information.
   - **PTRATIO** is chosen as the third feature because although its correlation is negative, it still has a moderate influence on the house price.
3. **Feature Importance**: Based on the correlation values, features like **LSTAT**, **RM**, and **PTRATIO** are likely to provide the most valuable insights for predicting house prices. Features like **ZN**, **TAX**, **INDUS** have weaker correlations and might not contribute as significantly.
4. **Avoiding Multicollinearity**: The model can suffer from **multicollinearity** if two or more features are highly correlated with each other. From the correlation matrix, we can see that **RM** and **ZN** have a **moderate correlation (0.36)**, which suggests that **RM** is a more significant feature for prediction. Therefore, we can avoid choosing **ZN** because it's somewhat redundant when **RM** is included.

Final Feature Selection (LSTAT, RM, PTRATIO):

- These three features were selected because:
  - They have **strong correlations with the target variable (PRICE)**.
  - They **provide unique information** without too much overlap (minimizing redundancy).
  - They represent factors that are intuitively relevant to house prices, such as the **socioeconomic status of the population (LSTAT)**, **size of the house (RM)**, and **educational environment (PTRATIO)**.

By selecting these features, you're ensuring that the model focuses on the most influential factors for predicting house prices while keeping the model simple and effective.

Let me know if you need more details on any of these steps!

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=4)

Explanation of `train_test_split`:

The code snippet:

```python
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=4)
```

is used to **split the dataset** into training and testing sets. Let's break down what each part does:

`train_test_split()` Function:

- **Purpose**: This function splits the dataset into two parts:
  - **Training set**: Used to train the model.
  - **Test set**: Used to evaluate the performance of the model on unseen data.

Parameters:

1. `X` (Features/Inputs):
   - The feature matrix containing all the input variables (in your case, it contains the features **LSTAT**, **RM**, and **PTRATIO**).
2. `y` (Target/Output):
   - The target variable (in this case, the house prices).
3. `test_size=0.2`:
   - Specifies that **20% of the data will be reserved for testing**, and the remaining **80%** will be used for training the model.
   - This is a common ratio used in machine learning to ensure that there is enough data to train the model while still having a separate set for testing.
4. `random_state=4`:
   - This is a seed for the random number generator. It ensures that the split is **reproducible**. Every time you run the code with the same `random_state`, you will get the same train-test split, which is useful for consistency and debugging.
   - If you omit this parameter or use `random_state=None`, the split would be random each time.

Outputs:

- `X_train`: The feature matrix for the training set (80% of the data).
- `X_test`: The feature matrix for the test set (20% of the data).
- `y_train`: The target variable for the training set (corresponding to `X_train`).
- `y_test`: The target variable for the test set (corresponding to `X_test`).

Why Split the Data?

- **Training the Model**: You train the model on the **training set** (`X_train`, `y_train`) so it can learn from the data.

- **Testing the Model**: After training, you use the **test set** (`X_test`, `y_test`) to evaluate how well the model performs on data it hasn't seen before. This gives a measure of the model's **generalization ability**, which is important to ensure the model works on real-world, unseen data.

Let me know if you need further clarification!

scaler = StandardScaler() # Initializing StandardScaler
X_train_scaled = scaler.fit_transform(X_train) # Fit and transform training data
X_test_scaled = scaler.transform(X_test) # Transform test data using the same scaler

Explanation of Standardization and the Code:

The code:

```python
scaler = StandardScaler()  # Initializing StandardScaler
X_train_scaled = scaler.fit_transform(X_train)  # Fit and transform training data
X_test_scaled = scaler.transform(X_test)  # Transform test data using the same scaler
```

is used to **standardize the feature data** so that the model performs better and more reliably. Let's break it down:

Why Standardize the Data?

1. **Feature Scaling**:
   - Many machine learning algorithms, including linear regression and neural networks, perform better when the features are on similar scales. If the features have vastly different scales (e.g., one feature ranges from 0 to 1, and another ranges from 1,000 to 10,000), the model may struggle to converge or give more weight to the larger-scaled features, leading to poor predictions.
2. **Standardization**:
   - Standardization (also known as **z-score normalization**) transforms the data so that it has a **mean of 0** and a **standard deviation of 1**. This ensures that each feature contributes equally to the model, improving performance and convergence during training.
   - Formula for standardization:

     $$z = \frac{x - \mu}{\sigma}$$

     where:
     - x is the value of a feature,
     - μ is the mean of that feature,
     - σ is the standard deviation of that feature.

Code Breakdown:

1. `` `scaler = StandardScaler()` ``:
   - This line creates an instance of the **StandardScaler** class from the `` `sklearn.preprocessing` `` module. The `` `StandardScaler` `` will be used to standardize the feature data.
2. `` `X_train_scaled = scaler.fit_transform(X_train)` ``:
   - The `` `fit_transform()` `` method is used on the **training data** (`` `X_train` ``).
     - `` `fit()` `` computes the mean and standard deviation of the training data for each feature.
     - `` `transform()` `` then uses these computed values to standardize the training data.
   - **Result**: The training data is now scaled to have a mean of 0 and a standard deviation of 1 for each feature.
3. `` `X_test_scaled = scaler.transform(X_test)` ``:
   - The `` `transform()` `` method is applied to the **test data** (`` `X_test` ``), but instead of recalculating the mean and standard deviation (which should be derived from the training set only), it uses the values learned from the **training set** (`` `X_train` ``).
   - **Result**: The test data is transformed using the same scaling factors (mean and standard deviation) that were applied to the training data, ensuring consistency across both training and test sets.

Why Not `` `fit_transform()` `` on `` `X_test` ``?

- It's crucial to only **fit** the scaler on the **training data**. Fitting on the test data would lead to **data leakage** where the model might indirectly "see" the test data during training. This can lead to over-optimistic results and poor generalization.
- **Test data must remain unseen** to simulate how the model will perform on real, unseen data.

Why Is This Important for Your Model?

- **Neural Networks** and **Linear Regression** are sensitive to the scale of input features, and when features have varying scales, the optimization process might not be as efficient. Standardizing the data helps the model learn better and faster.

Let me know if you'd like further clarification or have more questions!

ok from now decrease the size of answer
# Linear Regression Model
lr_model = LinearRegression() # Initializing Linear Regression Model
lr_model.fit(X_train_scaled, y_train) # Training the model using scaled training data

The code:

```python
lr_model = LinearRegression()  # Initializing Linear Regression Model
lr_model.fit(X_train_scaled, y_train)  # Training the model using scaled training data
```

- **`LinearRegression()`**: Initializes the linear regression model.
- **`fit()`**: Trains the model using the scaled training data (**`X_train_scaled`**) and the target variable (**`y_train`**). The model learns the relationship between the features and target.

y_pred_lr = lr_model.predict(X_test_scaled)

The code:

```python
y_pred_lr = lr_model.predict(X_test_scaled)
```

- **`predict()`**: Uses the trained linear regression model (**`lr_model`**) to make predictions on the scaled test data (**`X_test_scaled`**).
- **`y_pred_lr`**: Stores the predicted house prices for the test set.

mse_lr = mean_squared_error(y_test, y_pred_lr) # Mean Squared Error
mae_lr = mean_absolute_error(y_test, y_pred_lr) # Mean Absolute Error
r2_lr = r2_score(y_test, y_pred_lr) # R^2 Score (Model accuracy measure)

The code:

```python
mse_lr = mean_squared_error(y_test, y_pred_lr)  # Mean Squared Error
mae_lr = mean_absolute_error(y_test, y_pred_lr)  # Mean Absolute Error
r2_lr = r2_score(y_test, y_pred_lr)  # R² Score (Model accuracy measure)
```

- **`mean_squared_error()`**: Calculates the average squared difference between actual and predicted values. A lower value indicates better model performance.
- **`mean_absolute_error()`**: Measures the average absolute difference between actual and predicted values.
- **`r2_score()`**: Computes the R^2 score, which shows how well the model's predictions match the actual values. A score closer to 1 indicates better performance.

why three

Using three evaluation metrics — **MSE**, **MAE**, and **R²** — gives a more comprehensive view of the model's performance:

1. **MSE (Mean Squared Error)**:
   - Sensitive to large errors due to squaring the differences.
   - Provides a clear signal when large errors occur, but can be heavily influenced by outliers.
2. **MAE (Mean Absolute Error)**:
   - Gives a simpler, linear view of errors.
   - Less sensitive to outliers than MSE, providing a more robust performance measure when outliers are present.
3. **R^2 Score**:
   - Indicates how well the model fits the data.
   - Ranges from 0 (no fit) to 1 (perfect fit). Helps assess the overall accuracy of the model.

Together, these metrics offer a balanced perspective on both the model's error size and its ability to explain the variance in the data.

model = Sequential([
  Dense(128, activation='relu', input_dim=3), # Input layer (3 features) & first hidden layer (128 neurons)
  Dense(64, activation='relu'), # Second hidden layer with 64 neurons
  Dense(32, activation='relu'), # Third hidden layer with 32 neurons
  Dense(16, activation='relu'), # Fourth hidden layer with 16 neurons
  Dense(1) # Output layer (Predicting a single value - House Price)
])

explain in detail

Explanation of the Neural Network Architecture:

```python
model = Sequential([
    Dense(128, activation='relu', input_dim=3),  # Input layer (3 features) & first hidden layer (1
    Dense(64, activation='relu'),  # Second hidden layer with 64 neurons
    Dense(32, activation='relu'),  # Third hidden layer with 32 neurons
    Dense(16, activation='relu'),  # Fourth hidden layer with 16 neurons
    Dense(1)  # Output layer (Predicting a single value - House Price)
])
```

This code builds a **fully connected feed-forward neural network** using the **Keras Sequential API**. Here's a breakdown of each layer:

---

1. `Sequential()`:

- The `Sequential` model allows you to stack layers in a linear order, one after the other. Each layer's output becomes the input to the next layer.

---

2. First Layer: `Dense(128, activation='relu', input_dim=3)`:

- `Dense(128)`: This creates a fully connected layer with **128 neurons**. Each neuron connects to every neuron in the previous layer.
- `activation='relu'`: The **ReLU (Rectified Linear Unit)** activation function introduces non-linearity to the model. ReLU transforms negative values to 0 and keeps positive values unchanged, which helps with learning complex patterns.
- `input_dim=3`: Specifies that the input data has 3 features (LSTAT, RM, PTRATIO). The input layer expects 3 values, matching the number of features used for training.

---

3. Second Layer: `Dense(64, activation='relu')`:

- `Dense(64)`: This layer has 64 neurons, and it's fully connected to the previous layer (128 neurons).
- `activation='relu'`: Again, ReLU is applied, helping the network learn non-linear relationships.

---

4. Third Layer: `Dense(32, activation='relu')`:

- `Dense(32)`: This layer has 32 neurons. The number of neurons reduces as we move deeper into the network to gradually extract more abstract features.
- `activation='relu'`: ReLU activation ensures non-linearity.

---

5. Fourth Layer: `Dense(16, activation='relu')`:

- `Dense(16)`: This layer has 16 neurons, further reducing the number of features being processed as we approach the output.
- `activation='relu'`: Again, applying ReLU activation.

---

6. Output Layer: `Dense(1)`:

- `Dense(1)`: This layer has a single neuron, which produces the final output. For regression tasks like predicting house prices, the output layer has only one neuron to predict a continuous value (house price in this case).
- **No activation function**: By default, the output layer has no activation function because we want the output to be a real-valued number, suitable for regression tasks.

Why This Architecture?

- **Multiple Hidden Layers**: The layers with decreasing neurons (128 → 64 → 32 → 16) allow the network to learn complex patterns in the data. As we progress deeper into the network, the model

learns more abstract and intricate features.
- **ReLU Activation**: ReLU is commonly used because it is computationally efficient and helps with the vanishing gradient problem, making training faster.
- **Single Output Neuron**: Since this is a regression task (predicting a continuous value), we use a single output neuron without an activation function.

This neural network is designed to predict house prices based on the 3 input features (LSTAT, RM, PTRATIO). The deeper layers allow the model to capture complex relationships between these features.

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

The code:

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

Explanation:

1. **`optimizer='adam'`**:
   - **Adam (Adaptive Moment Estimation)** is an optimizer that computes adaptive learning rates for each parameter. It's commonly used due to its efficiency and ability to work well in a variety of tasks.
   - It combines the advantages of two other extensions of stochastic gradient descent: **Momentum** and **RMSprop**.
   - Adam adjusts the learning rate during training, which makes it suitable for most deep learning tasks.
2. **`loss='mse'`**:
   - **MSE (Mean Squared Error)** is the loss function used to measure the model's prediction error.
   - It's calculated as the average of the squared differences between the predicted values and the actual values.
   - This is appropriate for **regression** tasks like predicting house prices, as it penalizes larger errors more than smaller ones.
3. **`metrics=['mae']`**:
   - **MAE (Mean Absolute Error)** is used as an additional evaluation metric.
   - MAE calculates the average of the absolute differences between predicted and actual values, providing a straightforward measure of model accuracy.
   - It’s helpful to monitor MAE in addition to MSE because it gives a more intuitive understanding of how close the predictions are to actual values, without squaring the differences as in MSE.

# Training the Neural Network
history = model.fit(X_train_scaled, y_train, epochs=100, validation_split=0.05, verbose=1)

The code:

```python
history = model.fit(X_train_scaled, y_train, epochs=100, validation_split=0.05, verbose=1)
```

Explanation:

1. `X_train_scaled`:
   - This is the training data that has been **standardized** (scaled) using `StandardScaler` to ensure that all features are on the same scale. This helps improve the performance and convergence of the neural network.
2. `y_train`:
   - This is the target variable, which is the house price (`PRICE`) for the training dataset.
3. `epochs=100`:
   - An **epoch** refers to one complete pass through the entire training dataset.
   - In this case, the model will train for **100 epochs**, meaning the model will make 100 passes through the data.
   - More epochs generally improve the model, but too many can lead to overfitting.
4. `validation_split=0.05`:
   - This parameter indicates that **5%** of the training data will be set aside as a validation set to monitor the model's performance during training.
   - It helps detect overfitting by providing feedback on how well the model generalizes to unseen data (i.e., not used for training).
   - The validation set is used for evaluating the model after each epoch without affecting the training process.
5. `verbose=1`:
   - This controls how much information is displayed during training.
   - `verbose=1` provides detailed progress information for each epoch, showing the loss and metrics (like MAE) for both the training and validation datasets.

What Happens During Training:

- During each epoch, the model updates its weights based on the gradients computed from the loss function (MSE).
- After each epoch, the model evaluates its performance on the **validation set** (5% of the data) and reports the validation loss and MAE.

- This allows you to track both the training progress and how well the model is performing on unseen data.

Outcome:

- The `history` object contains details about the training process, such as loss and accuracy metrics for each epoch, which you can analyze later to understand how the model's performance evolved over time.

y_pred_nn = model.predict(X_test_scaled) # Predicting house prices on test data
mse_nn, mae_nn = model.evaluate(X_test_scaled, y_test) # Evaluating model performance

Code Explanation:

```python
y_pred_nn = model.predict(X_test_scaled)
```

- Predicts house prices using the **trained neural network** on the **test set**.
- `X_test_scaled` is the standardized version of the test data.

```python
mse_nn, mae_nn = model.evaluate(X_test_scaled, y_test)
```

- Evaluates model performance on the **test set**.
- Returns:
  - `mse_nn`: Mean Squared Error
  - `mae_nn`: Mean Absolute Error
- Helps compare how well the neural network performs vs linear regression.

Want the code to display R^2 score too?

new_data = np.array([[0.1, 10.0, 5.0]])

This line creates a new data point for prediction:

```python
new_data = np.array([[0.1, 10.0, 5.0]])
```

It represents:

- `LSTAT = 0.1`: Very low percentage of lower-status population.
- `RM = 10.0`: High average number of rooms per house.
- `PTRATIO = 5.0`: Very low pupil-teacher ratio.

➡️ This is a **high-value home scenario**, used to test the model's prediction on new inputs.

Want the prediction steps explained next?

prediction = model.predict(new_data_scaled)

# Displaying the predicted house price
print("\nPredicted House Price:", prediction[0][0])

This code predicts the house price using the trained neural network:

```python
prediction = model.predict(new_data_scaled)
```

- Uses the **standardized input** (`new_data_scaled`) to make prediction.
- Returns a 2D array, so `prediction[0][0]` extracts the scalar value.

```python
print("\nPredicted House Price:", prediction[0][0])
```

- Displays the predicted price for the new house data.

✅ It simulates a real-world prediction for unseen data. Want a summary of the full flow?

now explain the practicle theory

Here's a **short and clear explanation** of your practical titled:

✅ Linear Regression using Deep Neural Network: Boston Housing Price Prediction

---

🔍 Objective

To predict house prices using both:

- **Linear Regression (ML)**
- **Deep Neural Network (ANN - Artificial Neural Network)**

on the **Boston Housing dataset**, using three features:

- **LSTAT** (low-income population %),
- **RM** (rooms per house),
- **PTRATIO** (pupil-teacher ratio)

📊 Steps Involved

1. **Import Libraries**
   - `numpy`, `pandas`, `sklearn`, `keras` for model building and evaluation.
2. **Load Dataset**
   - Load Boston housing dataset (CSV file).
3. **Feature Selection**
   - Chosen based on correlation with `PRICE` (target):
     - LSTAT (−0.73), RM (+0.69), PTRATIO (−0.50)
4. **Train-Test Split**
   - 80% training, 20% testing.
5. **Standardization**
   - Normalize input features using `StandardScaler`.
6. **Model 1: Linear Regression**
   - Train on scaled features.
   - Evaluate using:
     - MSE, MAE, R^2 Score
7. **Model 2: Deep Neural Network**
   - Layers: 128 → 64 → 32 → 16 → 1
   - Activation: `ReLU`, optimizer: `Adam`, loss: `MSE`
   - Trained for 100 epochs with validation split.
8. **Evaluation**
   - Compare both models using MSE, MAE.
   - Neural network usually performs better with complex patterns.
9. **New Prediction**
   - Given new input `[0.1, 10.0, 5.0]`, predict price using ANN.

📌 Conclusion

- Linear regression is simple and fast, but limited.
- Deep neural networks capture non-linear relationships better.

- ANN gives more accurate predictions in this case.

Want a short viva Q&A list for this practical?

so we have to draw 128 neuron
then on next line 64 so on

Yes, that's correct! In a **Deep Neural Network (DNN)**, each line of neurons in the model corresponds to a **layer**, and you can visualize it as:

- **Input Layer**: 3 features → 3 input nodes.
- **Hidden Layer 1**: 128 neurons.
- **Hidden Layer 2**: 64 neurons.
- **Hidden Layer 3**: 32 neurons.
- **Hidden Layer 4**: 16 neurons.
- **Output Layer**: 1 neuron for predicting the house price.

Visual Representation:

```scss
Input Layer (3 features) → 128 neurons → 64 neurons → 32 neurons → 16 neurons → Output (1 predicted
```

You can think of it like this:

- Arrows (or edges) between layers represent the weights connecting neurons.
- Each neuron in a layer is connected to all neurons in the next layer, hence **fully connected**.

Would you like me to generate this diagram for you?

draw it butdo not generate image just use text and line

Here's a text-based visualization of your **Deep Neural Network (DNN)** architecture:

```less
Input Layer (3 Features)
    |       |       |
   [X1]    [X2]    [X3]
    |       |       |
    -----------------
        Hidden Layer 1 (128 neurons)
    |       |       |        ...
   [H1]    [H2]    [H3]    [H128]
    |       |       |        ...
    -----------------
        Hidden Layer 2 (64 neurons)
```

```
     |       |       |        ...
    [H1]    [H2]    [H3]    [H64]
     |       |       |        ...
    -----------------
        Hidden Layer 3 (32 neurons)
     |       |       |        ...
    [H1]    [H2]    [H3]    [H32]
     |       |       |        ...
    -----------------
        Hidden Layer 4 (16 neurons)
     |       |       |        ...
    [H1]    [H2]    [H3]    [H16]
     |       |       |        ...
    -----------------
        Output Layer (1 neuron)
                |
            [Predicted House Price]
```

Breakdown:

1. **Input Layer (3 features)**: `X1`, `X2`, `X3` are the three input features (LSTAT, RM, PTRATIO).
2. **Hidden Layers**: Each hidden layer consists of neurons (128, 64, 32, 16).
3. **Output Layer**: Single neuron for the predicted house price.

This is how the data flows from the input features through the hidden layers and finally to the output.

Let me know if you need further explanation!

in horizontal way

Here's a horizontal text-based visualization of your **Deep Neural Network (DNN)** architecture:

```scss
Input Layer (3 Features)      Hidden Layer 1 (128 neurons)      Hidden Layer 2 (64 neurons)      Hi
    [X1] ------------> [H1]      [H1] ------------> [H1]          [H1] ------------> [H1]          [H
    [X2] ------------> [H2]      [H2] ------------> [H2]          [H2] ------------> [H2]          [H
    [X3] ------------> [H3]      [H3] ------------> [H3]          [H3] ------------> [H3]          [H
                     [H128]   --> [H64]           --> [H32]          --> [H16]
```

Breakdown:

- **Input Layer**: 3 features, connected to the first hidden layer (128 neurons).
- **Hidden Layers**: 128 neurons in the first layer, 64 in the second, 32 in the third, and 16 in the fourth.
- **Output Layer**: 1 neuron to predict the house price.

Each layer is fully connected to the next one, represented by arrows pointing from each neuron to the next.

Would you like further details or modifications to this diagram?

now explain how this dnn model works

Great! Let's break down how your **Deep Neural Network (DNN)** model works, step by step:

🧠 Structure Recap

- **Input Layer**: 3 features — `LSTAT`, `RM`, `PTRATIO`
- **Hidden Layers**: 4 layers (128 → 64 → 32 → 16 neurons)
- **Output Layer**: 1 neuron (predicts the house price)

⚙️ Working Step-by-Step

1. Input Feeding

You feed a row of 3 input values (e.g., `[0.1, 10.0, 5.0]`) into the network.
These inputs are already **scaled** (using `StandardScaler`) for better training performance.

---

2. Layer-wise Processing

🔹 Hidden Layer 1:

- 128 neurons, each neuron:
  - Takes **all 3 input values**
  - Applies a **weighted sum** (like: `w1*LSTAT + w2*RM + w3*PTRATIO + bias`)
  - Passes the result through **ReLU activation** (sets negative values to 0)
- This layer extracts **complex patterns** from the inputs.

🔹 Hidden Layer 2 (64 neurons):

- Each neuron receives outputs from all 128 neurons of the previous layer.
- Again, applies weighted sums + **ReLU**.
- Starts refining the pattern into more meaningful shapes.

🔹 Hidden Layer 3 (32 neurons):

- Connects to the 64 outputs from previous.
- Further **refines and combines features**.

🔹 Hidden Layer 4 (16 neurons):

- More condensation of learned features — prepares data for prediction.

---

3. Output Layer (1 neuron):

- Combines the 16 values from the last hidden layer.
- **No activation** (or linear activation) — just outputs a number.
- Final result: **Predicted House Price**

🎯 Training the Model

- The model compares predictions with real prices using **Mean Squared Error (MSE)**.
- It uses **backpropagation** and the **Adam optimizer** to adjust weights after each epoch.

🔁 Epochs and Learning

- Repeats forward + backward passes for 100 epochs.
- Each time, it **reduces the error**, so predictions improve over time.

✅ Final Output

After training:

- You input new data like `[0.1, 10.0, 5.0]`
- It goes through all layers
- You get a predicted house price (e.g., `$25.3k`)

Would you like a simple analogy or visual flow for understanding this process further?

what you give that like w1 LSTAT + ... + bias
what was that

Great question!

The expression like `w1 * LSTAT + w2 * RM + w3 * PTRATIO + bias` is the **core mathematical operation** that happens inside a single neuron.

🔢 What does it mean?

Each **neuron** performs a calculation:

```ini
z = (w1 × x1) + (w2 × x2) + (w3 × x3) + b
```

Where:

- `x1, x2, x3` = input values (`LSTAT`, `RM`, `PTRATIO`)

- `w1, w2, w3` = weights (learned by the model during training)
- `b` = bias (another value the model learns)
- `z` = output of that neuron *before* applying activation

Then it passes `z` through an **activation function**, like **ReLU**:

```ini
output = ReLU(z) = max(0, z)
```

🔁 This repeats

- Every neuron in every layer does this.
- Weights (`w`) and biases (`b`) are **automatically adjusted** by the training process (backpropagation) to reduce prediction error.

This is how the model **learns relationships** between features and the target (price). Want a quick numerical example to make it clearer?

giveme this whole page in pdf

Your PDF explaining the inner workings of a neuron in the DNN model is ready.
You can download it using the link below:

👉

Would you like a diagram version as well?