Understanding the code

**1. from sklearn.preprocessing import MinMaxScaler**

* This line imports the MinMaxScaler from Scikit-learn. It will be used to normalize the data so that all the feature values are scaled between 0 and 1, which is essential for neural networks like LSTM models.

**2. Normalize the data using MinMaxScaler**

python

Copy code

scaler = MinMaxScaler()

scaled\_data = scaler.fit\_transform(df\_final)

* **scaler = MinMaxScaler()**: Initializes an instance of the MinMaxScaler. This will scale the values of each feature to a range (by default between 0 and 1).
* **scaled\_data = scaler.fit\_transform(df\_final)**:
  + fit\_transform(df\_final) first learns (or "fits") the minimum and maximum values of each feature in the DataFrame df\_final and then transforms each value in the DataFrame to the 0-1 range.
  + scaled\_data will be the normalized version of df\_final.

**3. Define sequence length and number of features**

python

Copy code

sequence\_length = 10 # Number of time steps in each sequence

num\_features = len(df\_final.columns) # Number of features (columns) in the dataset

* **sequence\_length = 10**: This defines the number of time steps (or "look-back" steps) in each sequence that will be fed into the LSTM model. In this case, each input sequence will have data for 10 time steps.
* **num\_features = len(df\_final.columns)**: This determines the number of features (columns) in the dataset. For example, if your dataset has columns like temperature, humidity, and wind speed, num\_features would represent the number of these columns.

**4. Create sequences and corresponding labels**

python

Copy code

sequences = []

labels = []

for i in range(len(scaled\_data) - sequence\_length):

seq = scaled\_data[i:i + sequence\_length] # A sequence of 10 time steps

label = scaled\_data[i + sequence\_length][6] # The target value (7th column) for prediction

sequences.append(seq)

labels.append(label)

* **sequences = []** and **labels = []**: These lists will hold the input sequences and their corresponding labels (what you're predicting).
* **for i in range(len(scaled\_data) - sequence\_length):**: This loop goes through the scaled\_data to create multiple sequences of data, each with sequence\_length time steps.
* **seq = scaled\_data[i:i + sequence\_length]**:
  + This line creates a sequence of data from index i to i + sequence\_length. Each sequence contains sequence\_length (10) time steps.
* **label = scaled\_data[i + sequence\_length][6]**:
  + Here, the **label** for the sequence is taken from the 7th column (index 6) of the row immediately after the sequence. This assumes that the target for prediction is in the 7th column.
* **sequences.append(seq)**: The current sequence of time steps is added to the sequences list.
* **labels.append(label)**: The corresponding label (the value to be predicted) is added to the labels list.

**5. Convert sequences and labels to numpy arrays**

python

Copy code

sequences = np.array(sequences)

labels = np.array(labels)

* This converts the sequences and labels lists into **NumPy arrays**. NumPy arrays are the standard format expected by most machine learning libraries (like TensorFlow and Keras) and make mathematical operations faster and easier.

**6. Split into training and testing sets**

python

Copy code

train\_size = int(0.8 \* len(sequences)) # 80% of the data for training

train\_x, test\_x = sequences[:train\_size], sequences[train\_size:] # Split the sequences

train\_y, test\_y = labels[:train\_size], labels[train\_size:] # Split the labels

* **train\_size = int(0.8 \* len(sequences))**: This calculates the number of samples to be used for training. In this case, 80% of the total sequences are used for training.
* **train\_x, test\_x = sequences[:train\_size], sequences[train\_size:]**: This splits the sequences into two sets:
  + **train\_x** contains the first 80% of the sequences (used for training the model).
  + **test\_x** contains the remaining 20% of the sequences (used for testing or validating the model).
* **train\_y, test\_y = labels[:train\_size], labels[train\_size:]**: Similarly, this splits the labels into train\_y (for training) and test\_y (for testing).

**7. Print the shapes of training and test data**

python

Copy code

print("Train X shape:", train\_x.shape)

print("Train Y shape:", train\_y.shape)

print("Test X shape:", test\_x.shape)

print("Test Y shape:", test\_y.shape)

* **train\_x.shape**: Prints the shape of the training sequences. The shape should be (number\_of\_training\_samples, sequence\_length, num\_features).
  + For example, if you have 1000 training samples, each with 10 time steps and 7 features, the shape would be (1000, 10, 7).
* **train\_y.shape**: Prints the shape of the training labels. The shape should be (number\_of\_training\_samples,), because each sequence has one corresponding label.
* **test\_x.shape** and **test\_y.shape**: Similarly, these print the shapes of the test data and labels.

**Summary:**

* You first normalize your data using MinMaxScaler to scale it between 0 and 1.
* Then, you create sequences of data, each containing 10 time steps (as per sequence\_length), and assign the label as the value of the 7th column at the next time step.
* The sequences and labels are converted to NumPy arrays and split into training and test sets.
* Finally, you print the shapes of the training and test data to confirm they are correctly structured for feeding into an LSTM model.

LSTM MODEL IMPLEMENTATION:

**Importing Necessary Modules**

python

Copy code

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import LSTM, Dense, Dropout, Input

* **Sequential**: This is a type of model in Keras where layers are added one after another in sequence. It’s simpler and commonly used for building neural networks with one input and one output.
* **LSTM**: Long Short-Term Memory is a type of recurrent neural network (RNN) that is effective for time-series forecasting and handling sequential data because it can maintain context across long sequences.
* **Dense**: This is a fully connected (dense) layer, where each neuron receives input from all neurons of the previous layer. This is usually used for the output layer.
* **Dropout**: This is a regularization technique that randomly drops units (neurons) during training to prevent overfitting. It helps make the model generalize better.
* **Input**: This is used to explicitly define the shape of the input data for the model, which is preferred in modern Keras practices.

**Creating the LSTM Model**

python

Copy code

model = Sequential()

* **Sequential()**: You create an instance of the Sequential model. It allows you to stack layers on top of each other in sequence, starting with the input layer and ending with the output layer.

**Defining the Input Layer**

python

Copy code

model.add(Input(shape=(train\_x.shape[1], train\_x.shape[2])))

* **Input(shape=(train\_x.shape[1], train\_x.shape[2]))**: This defines the shape of the input data.
  + train\_x.shape[1] represents the number of time steps (sequence length).
  + train\_x.shape[2] represents the number of features in each time step.
  + You pass this shape to Input(), which specifies how the input data will be structured when entering the model.

**Adding the First LSTM Layer**

python

Copy code

model.add(LSTM(units=128, return\_sequences=True))

* **LSTM(units=128)**: This adds the first LSTM layer with 128 units (neurons) to learn patterns from the data.
  + **Units**: The number of LSTM neurons. More units increase the model's capacity to learn complex patterns, but also increase the risk of overfitting and computational cost.
  + **return\_sequences=True**: This ensures that the LSTM layer outputs the full sequence of data for the next LSTM layer, instead of just the last output of the sequence.

**Adding Dropout after the First LSTM Layer**

python

Copy code

model.add(Dropout(0.2))

* **Dropout(0.2)**: This applies dropout regularization, which randomly sets 20% of the units to 0 during training to prevent overfitting. Dropout helps the model to generalize better by not depending too much on certain neurons.

**Adding the Second LSTM Layer**

python

Copy code

model.add(LSTM(units=64, return\_sequences=True))

* **LSTM(units=64)**: This is the second LSTM layer with 64 neurons. It continues learning from the sequence of data passed from the first LSTM layer.
  + **return\_sequences=True**: This ensures that this LSTM layer outputs the full sequence of data for the next layer.

**Adding Dropout after the Second LSTM Layer**

python

Copy code

model.add(Dropout(0.2))

* **Dropout(0.2)**: Again, this applies dropout regularization to reduce overfitting by dropping 20% of the neurons randomly.

**Adding the Third LSTM Layer**

python

Copy code

model.add(LSTM(units=32, return\_sequences=False))

* **LSTM(units=32)**: This is the third LSTM layer with 32 neurons. It further processes the data but outputs only the final result of the sequence, not the whole sequence.
  + **return\_sequences=False**: Here, you only want the last output of the LSTM sequence (not the entire sequence) because you’re about to send this to the output layer.

**Adding Dropout after the Third LSTM Layer**

python

Copy code

model.add(Dropout(0.2))

* **Dropout(0.2)**: This again applies dropout to reduce overfitting by randomly dropping 20% of the neurons during training.

**Adding a Dense Output Layer**

python

Copy code

model.add(Dense(units=1))

* **Dense(units=1)**: This is the output layer. A Dense layer connects all the neurons from the previous LSTM layer to this output.
  + **units=1**: Since this is a regression problem (predicting continuous values like time-series data), the output layer has 1 unit. This is where the model will produce the final prediction (a single value for the forecasted data point).

**Compiling the Model**

python

Copy code

model.compile(optimizer='adam', loss='mean\_squared\_error')

* **optimizer='adam'**: Adam (Adaptive Moment Estimation) is an efficient optimization algorithm that adjusts learning rates during training. It’s commonly used because it’s fast and works well across many models.
* **loss='mean\_squared\_error'**: This defines the loss function to be minimized during training. Mean Squared Error (MSE) is commonly used in regression tasks, as it calculates the average of the squared differences between the predicted and true values.

**Printing the Model Summary**

python

Copy code

model.summary()

* **model.summary()**: This prints out a summary of the model architecture, including the layers, output shapes, and the number of parameters in each layer. It's useful for verifying that the model has been built correctly.

**Full Flow of the Model:**

1. Input data flows through 3 LSTM layers, each extracting and learning features from the time-series data.
2. Dropout layers are applied after each LSTM layer to prevent overfitting.
3. The Dense layer outputs the final predicted value for the next time step in the sequence.

**#DEFINE CALLBACKS**

Here's a line-by-line explanation of your code:

**1. Importing Callbacks**

python

Copy code

from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint

* **EarlyStopping** and **ModelCheckpoint** are two commonly used callbacks in Keras for monitoring and controlling the training process.
  + **EarlyStopping**: Stops training early when the model's performance (e.g., loss) doesn't improve after a certain number of epochs.
  + **ModelCheckpoint**: Saves the model's weights after every epoch if the model's performance improves.

**2. Defining Early Stopping Callback**

python

Copy code

early\_stopping = EarlyStopping(monitor='val\_loss', patience=5, restore\_best\_weights=True)

* **monitor='val\_loss'**: Monitors the **validation loss** during training. The training process will focus on minimizing this loss.
* **patience=5**: If the validation loss does not improve after 5 consecutive epochs, the training will stop. This prevents overfitting and unnecessary training time.
* **restore\_best\_weights=True**: After stopping early, it will restore the model weights from the epoch where the validation loss was the best. This ensures that the model retains the best performance before overfitting began.

**3. Defining Model Checkpoint Callback**

python

Copy code

model\_checkpoint = ModelCheckpoint('best\_model\_weights.keras', monitor='val\_loss', save\_best\_only=True)

* **'best\_model\_weights.keras'**: Specifies the file where the best model weights will be saved (in the .keras format).
* **monitor='val\_loss'**: The callback monitors the validation loss. After each epoch, if the validation loss improves, the model weights will be saved.
* **save\_best\_only=True**: Ensures that only the weights from the best-performing epoch (lowest validation loss) are saved, rather than saving after every epoch.

**4. Training the Model**

python

Copy code

history = model.fit(

train\_x, train\_y,

epochs=100,

batch\_size=64,

validation\_split=0.2, # Use part of the training data as validation

callbacks=[early\_stopping, model\_checkpoint]

)

* **train\_x** and **train\_y**: The input data (train\_x) and the corresponding target labels (train\_y) for training.
* **epochs=100**: Specifies that the model should train for up to 100 epochs, but this could be interrupted by early stopping if the validation loss stops improving.
* **batch\_size=64**: The number of samples per batch. During training, the dataset is divided into batches of 64 samples at a time. This helps with memory efficiency and faster training.
* **validation\_split=0.2**: Uses 20% of the training data for validation. This means 80% of the data will be used for training and 20% will be used to evaluate the model's performance at each epoch.
* **callbacks=[early\_stopping, model\_checkpoint]**: The two callbacks (early\_stopping and model\_checkpoint) will be active during training.
  + **early\_stopping**: Will stop training if the validation loss doesn’t improve after 5 epochs.
  + **model\_checkpoint**: Will save the model's weights to 'best\_model\_weights.keras' whenever there is an improvement in validation loss.

**Summary:**

* **Early Stopping** prevents overfitting by stopping the training once the validation loss stops improving for 5 consecutive epochs.
* **Model Checkpoint** saves the best model weights during the training, so you can retrieve the best model configuration once training finishes.
* The model will train for a maximum of 100 epochs but may stop early if early stopping is triggered.

**EVALUATING THE BEST MODEL BASED ON TEST SET**

Certainly! Let's go through the code line by line:

python

Copy code

import tensorflow as tf

* **Explanation**: This imports the TensorFlow library, giving you access to all its machine learning tools, including Keras for neural network models. In this case, you need it to load and evaluate your pre-trained LSTM model.

python

Copy code

best\_model = tf.keras.models.load\_model('best\_model\_weights.keras')

* **Explanation**: This line loads the previously saved model from the file best\_model\_weights.keras. The model, which was saved during training with ModelCheckpoint, is restored with all its architecture, weights, and configurations intact. The file path 'best\_model\_weights.keras' must be the location of the model weights you saved.

python

Copy code

test\_loss = best\_model.evaluate(test\_x, test\_y)

* **Explanation**:
  + **best\_model.evaluate()**: This method evaluates the model on the test dataset (test\_x, test\_y) and returns the loss value for the test data.
  + **Arguments**:
    - test\_x: Input features from the test dataset.
    - test\_y: True output values (targets) for the test dataset.
  + This function computes the loss (in this case, **mean squared error** as specified during the model compilation) between the predicted values and the actual values in test\_y.
  + The result, test\_loss, stores the computed test loss (a measure of the model’s performance on unseen data).

python

Copy code

print("Test Loss:", test\_loss)

* **Explanation**: This prints out the test loss (the difference between the actual test set values and the predicted values by the model). A lower loss means the model performed better on the test data.

**Summary:**

* The code evaluates the performance of your best-trained model on the test data and prints the loss, which shows how well the model generalizes to unseen data.

**# Plot training & validation loss values**

Sure! Let’s break down this code that visualizes the training and validation loss over epochs:

python

Copy code

# Plot training & validation loss values

plt.plot(history.history['loss'])

* **Explanation**: This line uses the plot function from the Matplotlib library to plot the training loss over the epochs.
  + **history.history['loss']**: This accesses the training loss values recorded during model training. The history object was returned by the model.fit() method and contains the loss values for each epoch.

python

Copy code

plt.plot(history.history['val\_loss'])

* **Explanation**: Similar to the previous line, this plots the validation loss over the epochs.
  + **history.history['val\_loss']**: This accesses the validation loss values recorded during the training. The validation loss is calculated using the validation split provided during the training, which helps in monitoring how well the model generalizes to unseen data.

python

Copy code

plt.title('Model Loss')

* **Explanation**: This sets the title of the plot to "Model Loss". It helps convey what the graph represents.

python

Copy code

plt.xlabel('Epoch')

* **Explanation**: This labels the x-axis as "Epoch", indicating that the values plotted along this axis represent the number of epochs (iterations over the training dataset).

python

Copy code

plt.ylabel('Loss')

* **Explanation**: This labels the y-axis as "Loss", indicating that the values plotted along this axis represent the loss (error) values calculated during training and validation.

python

Copy code

plt.legend(['Train', 'Validation'], loc='upper right')

* **Explanation**: This adds a legend to the plot.
  + The legend will help distinguish between the training loss and validation loss curves.
  + **loc='upper right'** specifies the location of the legend in the plot, in this case, the upper right corner.

python

Copy code

plt.show()

* **Explanation**: This command displays the plot. It renders all the plotting commands issued above in a single figure, allowing you to visualize the training and validation loss over epochs.

**Summary:**

* The code provides a visual representation of how the model's training loss and validation loss changed over epochs, which can help you diagnose issues like overfitting (when the training loss decreases while the validation loss increases) or underfitting (when both losses are high).

**Cell 20**

Let's break down the code step by step, which is used to evaluate the performance of a trained model using common regression metrics.

python

Copy code

from sklearn.metrics import mean\_absolute\_error, mean\_squared\_error

* **Explanation**: This line imports two evaluation metrics from the sklearn.metrics module:
  + **mean\_absolute\_error (MAE)**: Measures the average magnitude of errors in a set of predictions, without considering their direction (i.e., the absolute values of the differences).
  + **mean\_squared\_error (MSE)**: Measures the average of the squares of the errors, which means it gives higher weight to larger errors.

python

Copy code

# Assuming you have trained the model and have the 'best\_model' object

# Also, 'test\_x' and 'test\_y' should be available

* **Explanation**: This comment serves as a reminder or note to indicate that the code assumes the model (best\_model) has already been trained and that the test data (test\_x for input features and test\_y for true output values) is available for evaluation.

python

Copy code

# Predict temperatures using the trained model

predictions = best\_model.predict(test\_x)

* **Explanation**: This line uses the trained model to make predictions on the test dataset.
  + **best\_model.predict(test\_x)**: The predict method generates predictions based on the input features in test\_x. The result is stored in the predictions variable, which contains the model's predicted values for the test set.

python

Copy code

# Calculate evaluation metrics

mae = mean\_absolute\_error(test\_y, predictions)

* **Explanation**: This line calculates the Mean Absolute Error (MAE) between the true values (test\_y) and the predicted values (predictions).
  + **mean\_absolute\_error(test\_y, predictions)**: This function compares the actual temperatures in test\_y with the predicted temperatures in predictions and returns the average absolute error, which is stored in the variable mae.

python

Copy code

mse = mean\_squared\_error(test\_y, predictions)

* **Explanation**: This line calculates the Mean Squared Error (MSE) between the true and predicted values.
  + **mean\_squared\_error(test\_y, predictions)**: This function computes the average of the squares of the errors between the actual and predicted values, which is stored in the variable mse.

python

Copy code

rmse = np.sqrt(mse)

* **Explanation**: This line calculates the Root Mean Squared Error (RMSE) by taking the square root of the Mean Squared Error (MSE).
  + **np.sqrt(mse)**: The np.sqrt function computes the square root of mse, giving you the RMSE, which is a measure of how spread out the errors are, in the same units as the original data. It is stored in the variable rmse.

python

Copy code

print("Mean Absolute Error (MAE):", mae)

* **Explanation**: This line prints the calculated Mean Absolute Error (MAE) to the console, providing an indication of the average error in the predictions.

python

Copy code

print("Mean Squared Error (MSE):", mse)

* **Explanation**: This line prints the calculated Mean Squared Error (MSE) to the console, giving insight into the average squared error in the predictions.

python

Copy code

print("Root Mean Squared Error (RMSE):", rmse)

* **Explanation**: This line prints the calculated Root Mean Squared Error (RMSE) to the console, providing a measure of the model's prediction error that is interpretable in the same units as the target variable.

**Summary:**

This code evaluates the performance of the trained model on the test dataset by calculating three common regression metrics: Mean Absolute Error (MAE), Mean Squared Error (MSE), and Root Mean Squared Error (RMSE). These metrics provide insight into how well the model's predictions align with the actual values.

**Cell 21:#y true values**

Let's break down this code step by step. This section is used to obtain the actual temperature values from the scaled predictions and the true values.

python

Copy code

# y\_true values

test\_y\_copies = np.repeat(test\_y.reshape(-1, 1), test\_x.shape[-1], axis=-1)

* **Explanation**: This line creates copies of the true values (test\_y) to match the shape required for inverse scaling.
  + **test\_y.reshape(-1, 1)**: Reshapes the test\_y array to have one column and as many rows as needed, transforming it into a 2D array with a shape of (n\_samples, 1).
  + **np.repeat(..., test\_x.shape[-1], axis=-1)**: Repeats each value in test\_y\_copies across the number of features in the dataset (assumed to be the same as test\_x.shape[-1]). This means if there are 8 features, each value in test\_y will be duplicated to form a shape of (n\_samples, 8). This is done to prepare the true values for inverse scaling.

python

Copy code

true\_temp = scaler.inverse\_transform(test\_y\_copies)[:,6]

* **Explanation**: This line performs inverse scaling on the copied true values to transform them back to their original scale.
  + **scaler.inverse\_transform(test\_y\_copies)**: This method applies the inverse transformation of the scaling that was done initially on the dataset, restoring the true values to their original range.
  + **[:,6]**: This selects the 7th column (index 6) from the resulting array, which presumably corresponds to the temperature feature. The result is stored in true\_temp.

python

Copy code

# predicted values

prediction = best\_model.predict(test\_x)

* **Explanation**: This line uses the trained model (best\_model) to make predictions based on the test dataset (test\_x).
  + **best\_model.predict(test\_x)**: The predict method generates the predicted values from the input features in test\_x, and the predictions are stored in the prediction variable.

python

Copy code

# Ensure you repeat the prediction to match the number of features the scaler was trained on (probably 8)

num\_features = test\_x.shape[-1] # This should be 8 if your original dataset had 8 features

* **Explanation**: This line sets the variable num\_features to the number of features in test\_x.
  + **test\_x.shape[-1]**: This retrieves the last dimension of the test\_x array, which corresponds to the number of features. It is expected to be 8 in this context.

python

Copy code

prediction\_copies = np.repeat(prediction, num\_features, axis=-1)

* **Explanation**: This line creates copies of the predicted values to match the shape required for inverse scaling.
  + **np.repeat(prediction, num\_features, axis=-1)**: Repeats each prediction across the number of features (8 in this case). This results in an array with the same number of samples as prediction, but with 8 columns, effectively preparing the predictions for inverse scaling.

python

Copy code

# Perform inverse scaling on predicted data

predicted\_temp = scaler.inverse\_transform(prediction\_copies)[:,6]

* **Explanation**: This line performs inverse scaling on the copied predicted values to transform them back to their original scale.
  + **scaler.inverse\_transform(prediction\_copies)**: Applies the inverse transformation to the predicted values, restoring them to their original range.
  + **[:,6]**: This selects the 7th column (index 6) from the resulting array, which corresponds to the temperature feature in the dataset. The result is stored in predicted\_temp.

**Summary:**

This code is designed to prepare the true and predicted values for inverse scaling, which transforms them back to their original scale after predictions have been made. It makes sure that both the true values and predictions match the number of features the scaler was originally trained on, allowing for accurate comparison and evaluation of model performance.

4o mini

**Last cell: PLOTTING PREDICTED AND ACTUAL TEMPERATURES**

Certainly! Let's break down the code used for plotting the predicted and actual temperatures step by step.

python

Copy code

# Plotting predicted and actual temperatures

plt.figure(figsize=(10, 6))

* **Explanation**: This line initializes a new figure for the plot with a specific size.
  + **plt.figure(figsize=(10, 6))**: Creates a new figure with a width of 10 inches and a height of 6 inches. This sets up the canvas for the subsequent plotting commands.

python

Copy code

plt.plot(df\_final.index[-100:], true\_temp[-100:], label='Actual')

* **Explanation**: This line plots the actual temperature values.
  + **df\_final.index[-100:]**: Retrieves the last 100 entries from the index of the DataFrame df\_final, which is used for the x-axis (representing time).
  + **true\_temp[-100:]**: Retrieves the last 100 actual temperature values from the true\_temp array, which will be plotted on the y-axis.
  + **label='Actual'**: Assigns the label "Actual" to this plot line, which will be shown in the legend.

python

Copy code

plt.plot(df\_final.index[-100:], predicted\_temp[-100:], label='Predicted')

* **Explanation**: This line plots the predicted temperature values.
  + **df\_final.index[-100:]**: Again retrieves the last 100 entries from the index of df\_final for the x-axis.
  + **predicted\_temp[-100:]**: Retrieves the last 100 predicted temperature values from the predicted\_temp array for the y-axis.
  + **label='Predicted'**: Assigns the label "Predicted" to this plot line, which will be shown in the legend.

python

Copy code

plt.title('Temperature Prediction vs Actual')

* **Explanation**: This line sets the title of the plot.
  + **plt.title('Temperature Prediction vs Actual')**: Sets the title of the plot to "Temperature Prediction vs Actual", which provides context to the viewer about what the plot represents.

python

Copy code

plt.xlabel('Time')

* **Explanation**: This line labels the x-axis.
  + **plt.xlabel('Time')**: Sets the label for the x-axis to "Time", indicating that the values plotted on this axis represent time.

python

Copy code

plt.ylabel('Temperature')

* **Explanation**: This line labels the y-axis.
  + **plt.ylabel('Temperature')**: Sets the label for the y-axis to "Temperature", indicating that the values plotted on this axis represent temperature values.

python

Copy code

plt.legend()

* **Explanation**: This line displays the legend for the plot.
  + **plt.legend()**: Shows the legend on the plot, which will include the labels "Actual" and "Predicted" for the respective lines plotted earlier. This helps to distinguish between the actual and predicted values in the plot.

python

Copy code

plt.show()

* **Explanation**: This line renders and displays the plot.
  + **plt.show()**: Displays the figure with the plotted data, allowing the user to visualize the comparison between the actual and predicted temperature values.

**Summary:**

This code snippet creates a line plot that visually compares the predicted temperatures against the actual temperatures over the last 100 time points in your dataset. The plot includes titles, axis labels, and a legend for clarity, helping to assess the model's performance visually.