**Google:**

**import numpy as np # Library for numerical computing**

**import pandas as pd # Library for data manipulation and analysis**

**import matplotlib.pyplot as plt # Library for creating static, animated, and interactive visualizations**

**from sklearn.preprocessing import MinMaxScaler # Module for data preprocessing**

**from keras.models import Sequential # Class for creating sequential models in Keras**

**from keras.layers import Dense, LSTM, Dropout # Classes for defining layers in a neural network**

**data = pd.read\_csv('Google\_train\_data.csv') # Reading the CSV file "Google\_train\_data.csv" and storing it in the variable "data"**

**data.head() # Displaying the first few rows of the DataFrame "data"**

**data.info()**

The **info()** function is a method of the pandas DataFrame object. When called on a DataFrame, it provides a summary of the DataFrame's structure and content. It displays information such as the number of rows and columns, column names, data types of each column, and the count of non-null values in each column. This information is useful for understanding the structure and integrity of the data in the DataFrame.

**data["Close"] = pd.to\_numeric(data.Close, errors='coerce')**

**# Convert the "Close" column of the DataFrame to numeric data type, replacing any non-numeric values with NaN** The **errors='coerce'** parameter is used to replace any non-numeric values with NaN.

**data = data.dropna() # Drop any rows in the DataFrame that contain NaN values, removing missing or invalid data**

**trainData = data.iloc[:, 4:5].values**

**# Extract a subset of the DataFrame containing only the values from the column at index 4 (5th column) and store it in "trainData"**

The first output represents the information of the original DataFrame "data" before preprocessing. It shows that the DataFrame has a total of 1258 entries (rows) and 6 columns. The column names are "Date", "Open", "High", "Low", "Close", and "Volume". The data types of the columns are shown as "float64" for the numeric columns (Open, High, Low) and "object" for the non-numeric columns (Date, Close, Volume). It also indicates the memory usage of the DataFrame.

The second output represents the information of the DataFrame "data" after performing the preprocessing steps and dropping the rows with missing values. It shows that the DataFrame now has 1149 entries (rows) and 6 columns. The column names and data types remain the same as in the first output. However, the "Close" column's data type has changed from "object" to "float64" after converting it to numeric values. The memory usage of the DataFrame has also changed slightly.

In summary, the difference between the two outputs is that the second output reflects the DataFrame after preprocessing, where the non-numeric values in the "Close" column have been converted to NaN and the rows with missing values have been dropped.

**sc = MinMaxScaler(feature\_range=(0,1)) # Create an instance of the MinMaxScaler class with the feature range set to (0, 1)**

**trainData = sc.fit\_transform(trainData) # Use the fit\_transform() method of the MinMaxScaler object to scale the "trainData" array**

**trainData.shape # Display the shape of the "trainData" array**

**X\_train = []**

**y\_train = []**

**for i in range(60, 1149): # Iterate over the range from 60 to 1149 (length of the data) #60 : timestep // 1149 : length**

**X\_train.append(trainData[i-60:i, 0])**

**y\_train.append(trainData[i, 0])**

**X\_train, y\_train = np.array(X\_train), np.array(y\_train)**

In the first two lines, empty lists **X\_train** and **y\_train** are created to store the input and output sequences for the model.

The **for** loop iterates over the range from 60 to 1149, with an increment of 1. This loop is used to create sequences for training the model.

Inside the loop, the line **X\_train.append(trainData[i-60:i, 0])** appends a sequence of 60 previous values (from index **i-60** to **i-1**) from the first column (**0th** column) of the **trainData** array to the **X\_train** list. These sequences will be used as input features for the model.

The line **y\_train.append(trainData[i, 0])** appends the value at index **i** from the first column (**0th** column) of the **trainData** array to the **y\_train** list. These values will be used as the corresponding target output for each input sequence.

Finally, the **np.array()** function is used to convert **X\_train** and **y\_train** from lists to numpy arrays. This step is necessary for further processing and training the model.

**X\_train = np.reshape(X\_train, (X\_train.shape[0], X\_train.shape[1], 1))**

**# Reshape the X\_train array to add an additional dimension for the batch size**

**X\_train.shape # Display the shape of the X\_train array**

In the first line, the **np.reshape()** function is used to reshape the **X\_train** array. The new shape is specified as **(X\_train.shape[0], X\_train.shape[1], 1)**, where **X\_train.shape[0]** represents the number of samples (rows) in the **X\_train** array, **X\_train.shape[1]** represents the number of time steps (columns) in each sample, and **1** represents the new dimension for the batch size.

By adding the batch size dimension, the reshaped **X\_train** array now has three dimensions: **(number of samples, number of time steps, 1)**. This format is often required when working with sequential data in deep learning models, where each sample represents a sequence of values.

In the second line, the **shape** attribute of the **X\_train** array is accessed to display its new shape, reflecting the addition of the batch size dimension.

**model = Sequential()**

**# Create an instance of the Sequential class to build the model**

**model.add(LSTM(units=100, return\_sequences=True, input\_shape=(X\_train.shape[1], 1)))**

**# Add an LSTM layer with 100 units, returning sequences, and specify the input shape**

**model.add(Dropout(0.2))**

**# Add a dropout layer with a dropout rate of 0.2**

**model.add(LSTM(units=100, return\_sequences=True))**

**# Add another LSTM layer with 100 units, returning sequences**

**model.add(Dropout(0.2))**

**# Add another dropout layer with a dropout rate of 0.2**

**model.add(LSTM(units=100, return\_sequences=True))**

**# Add another LSTM layer with 100 units, returning sequences**

**model.add(Dropout(0.2))**

**# Add another dropout layer with a dropout rate of 0.2**

**model.add(LSTM(units=100, return\_sequences=False))**

**# Add a final LSTM layer with 100 units, not returning sequences**

**model.add(Dropout(0.2))**

**# Add another dropout layer with a dropout rate of 0.2**

**model.add(Dense(units=1))**

**# Add a dense layer with 1 unit**

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

**# Compile the model with the Adam optimizer and the mean squared error loss function**

In this code, a sequential model is created using the **Sequential** class from Keras. The model consists of multiple LSTM layers, dropout layers, and a dense layer.

LSTM layers are a type of recurrent neural network layer that can handle sequential data. Each LSTM layer has 100 units and is specified to return sequences except for the last LSTM layer, which does not return sequences.

Dropout layers are used to prevent overfitting in the model. They randomly set a fraction of input units to 0 during training. In this code, dropout layers with a dropout rate of 0.2 are added after each LSTM layer.

Overfitting refers to a situation where a machine learning model performs exceptionally well on the training data but fails to generalize well to new, unseen data. In other words, the model becomes too complex and starts to memorize the training data instead of learning the underlying patterns and relationships.

A dense layer with 1 unit is added as the final layer of the model.

The model is compiled with the Adam optimizer and the mean squared error loss function, which is commonly used for regression tasks.

Regression, in the context of machine learning, refers to a type of supervised learning algorithm that is used to predict continuous numerical values based on input features. It is primarily used for solving regression problems where the goal is to estimate or predict a numerical outcome or target variable.

Overall, this code defines a model architecture suitable for sequence-to-sequence prediction tasks, such as time series forecasting.

**hist = model.fit(X\_train, y\_train, epochs=20, batch\_size=32, verbose=2)**

n this code, the **fit()** function of the model object is called to train the model. The function takes several parameters:

* **X\_train**: The input training data (sequences) for the model.
* **y\_train**: The target training data (corresponding outputs) for the model.
* **epochs=20**: The number of epochs or iterations over the entire training dataset.
* **batch\_size=32**: The number of samples per gradient update.
* **verbose=2**: The level of verbosity. In this case, **verbose=2** means that progress bars are displayed for each epoch.

The **fit()** function trains the model on the provided training data by iterating over the specified number of epochs. It performs forward propagation, calculates the loss, performs backward propagation to update the model's weights, and repeats this process for each epoch. The training history, including loss values and other metrics, is stored in the **hist** variable.

**plt.plot(hist.history['loss']) # Plot the training loss values from the training history**

**plt.title('Training model loss') # Set the title of the plot to 'Training model loss'**

**plt.ylabel('loss')# Set the label for the y-axis to 'loss'**

**plt.xlabel('epoch') # Set the label for the x-axis to 'epoch'**

**plt.legend(['train'], loc='upper left') # Add a legend to the plot with the label 'train' and position it in the upper left corner**

**plt.show() # Display the plot**

In this code, the **plot()** function from the **matplotlib.pyplot** module is used to create a line plot of the training loss values. The training loss values are accessed from the **history** attribute of the **hist** object, which contains the training history returned by the **fit()** function.

The **title()**, **xlabel()**, and **ylabel()** functions are used to set the title of the plot and labels for the x-axis and y-axis, respectively.

The **legend()** function is used to add a legend to the plot, with the label 'train'. The **loc='upper left'** parameter specifies the position of the legend in the upper left corner of the plot.

Finally, the **show()** function is called to display the plot.

This code allows you to visualize the training loss of the model over the epochs, which can provide insights into the training progress and convergence of the model.

**testData = pd.read\_csv('Google\_test\_data.csv')**

**# Read the test data from the 'Google\_test\_data.csv' file into a DataFrame**

**testData["Close"] = pd.to\_numeric(testData.Close, errors='coerce')**

**# Convert the 'Close' column of the testData DataFrame to numeric values, handling any conversion errors**

**testData = testData.dropna()**

**# Drop any rows in the testData DataFrame that contain missing values**

**testData = testData.iloc[:, 4:5]**

**# Select only the 'Close' column of the testData DataFrame as the input data**

**y\_test = testData.iloc[60:, 0:].values**

**# Extract the target values for the test data, starting from index 60**

**inputClosing = testData.iloc[:, 0:].values**

**# Extract the input closing prices from the testData DataFrame**

**inputClosing\_scaled = sc.transform(inputClosing)**

**# Scale the input closing prices using the previously fitted scaler object 'sc'**

**inputClosing\_scaled.shape # Display the shape of the scaled input closing prices**

**X\_test = []**

**length = len(testData)**

**timestep = 60**

**for i in range(timestep, length):**

**X\_test.append(inputClosing\_scaled[i-timestep:i, 0])**

**# Create sequences of length 60 from the scaled input closing prices and append them to the X\_test list**

**X\_test = np.array(X\_test) # Convert the X\_test list to a numpy array**

**X\_test = np.reshape(X\_test, (X\_test.shape[0], X\_test.shape[1], 1)) # Reshape the X\_test array to include the batch size dimension**

**X\_test.shape# Display the shape of the X\_test array**

In this code, the test data is loaded from the 'Google\_test\_data.csv' file into the **testData** DataFrame. The 'Close' column is converted to numeric values and any rows with missing values are dropped.

The target values **y\_test** are extracted starting from index 60, representing the target outputs corresponding to the test input sequences.

The input closing prices are stored in the **inputClosing** variable. They are then scaled using the **transform()** method of the previously fitted **sc** scaler object.

A loop is used to create sequences of length 60 from the scaled input closing prices. These sequences are appended to the **X\_test** list.

Finally, the **X\_test** list is converted to a numpy array and reshaped to include the batch size dimension. The shape of the resulting **X\_test** array is displayed. This prepares the test data in a format suitable for evaluating the trained model.

**y\_pred = model.predict(X\_test)**

**y\_pred**

In this code, the **predict()** function of the trained model is called to make predictions on the **X\_test** data. The **X\_test** data contains the input sequences for which we want to predict the corresponding output values.

The **predict()** function takes the **X\_test** data as input and returns the predicted output values. The predictions are stored in the **y\_pred** variable.

The resulting **y\_pred** variable contains the predicted output values for the test data, generated by the trained model.

By using the trained model to make predictions on the test data, you can assess the performance of the model and compare the predicted values with the actual target values (**y\_test**) to evaluate its accuracy and effectiveness.

**predicted\_price = sc.inverse\_transform(y\_pred)**

In this code, the **inverse\_transform()** method of the **sc** scaler object is used to reverse the scaling transformation applied to the **y\_pred** values. The purpose of this step is to obtain the predicted prices in their original scale or units.

The **inverse\_transform()** function takes the **y\_pred** array as input and applies the inverse scaling transformation to it. The resulting **predicted\_price** variable contains the predicted prices in their original scale.

By using **inverse\_transform()**, you can convert the scaled predicted values back to their original values, allowing for better interpretation and comparison with the actual prices.

**plt.plot(y\_test, color='red', label='Actual Stock Price')**

**plt.plot(predicted\_price, color='green', label='Predicted Stock Price')**

**plt.title('Google Stock Price Prediction')**

**plt.xlabel('Time')**

**plt.ylabel('Stock Price')**

**plt.legend()**

**plt.show()**

In this code, the **plot()** function from the **matplotlib.pyplot** module is used to create a line plot. The actual stock prices are plotted in red, and the predicted stock prices are plotted in green.

The **title()**, **xlabel()**, and **ylabel()** functions are used to set the title of the plot and labels for the x-axis and y-axis, respectively.

The **legend()** function is used to add a legend to the plot, which displays the labels 'Actual Stock Price' and 'Predicted Stock Price'.

Finally, the **show()** function is called to display the plot, showing the actual and predicted stock prices over time.

This code allows you to visually compare the actual stock prices with the predicted stock prices, helping you evaluate the performance of the model in predicting the stock price trends.

Certainly! The LSTM (Long Short-Term Memory) model is a type of recurrent neural network (RNN) that is particularly effective in handling long-term dependencies in sequential data. The LSTM model introduces memory cells and gates to control the flow of information through the network. Here are the equations that describe the LSTM model:

1. Input Gate:
   * The input gate determines how much new information is added to the memory cell.
   * Equation:
     + Input = sigmoid(W<sub>ix</sub> \* X<sub>t</sub> + W<sub>ih</sub> \* H<sub>t-1</sub> + b<sub>i</sub>)
     + Here, X<sub>t</sub> represents the input at time step t, H<sub>t-1</sub> represents the hidden state from the previous time step, W<sub>ix</sub> and W<sub>ih</sub> are the weight matrices for the input and hidden state, respectively, and b<sub>i</sub> is the bias vector.
2. Forget Gate:
   * The forget gate determines how much of the previous memory cell state should be forgotten.
   * Equation:
     + Forget = sigmoid(W<sub>fx</sub> \* X<sub>t</sub> + W<sub>fh</sub> \* H<sub>t-1</sub> + b<sub>f</sub>)
3. Update Memory Cell:
   * The update gate calculates the new values to be stored in the memory cell.
   * Equation:
     + Cell<sub>t</sub> = tanh(W<sub>cx</sub> \* X<sub>t</sub> + W<sub>ch</sub> \* H<sub>t-1</sub> + b<sub>c</sub>)
4. Output Gate:
   * The output gate determines the output at the current time step.
   * Equation:
     + Output = sigmoid(W<sub>ox</sub> \* X<sub>t</sub> + W<sub>oh</sub> \* H<sub>t-1</sub> + b<sub>o</sub>)
5. Hidden State:
   * The hidden state is updated based on the previous hidden state, the current input, and the memory cell state.
   * Equation:
     + H<sub>t</sub> = Output \* tanh(Cell<sub>t</sub>)

In summary, the LSTM model uses these equations to update and control the flow of information through the memory cell and hidden state. The input gate determines new information to be stored, the forget gate controls what information is forgotten, the update gate calculates the new memory cell values, and the output gate determines the output at the current time step. The hidden state is then updated based on the output and memory cell values. This allows the LSTM model to effectively capture and retain long-term dependencies in sequential data.