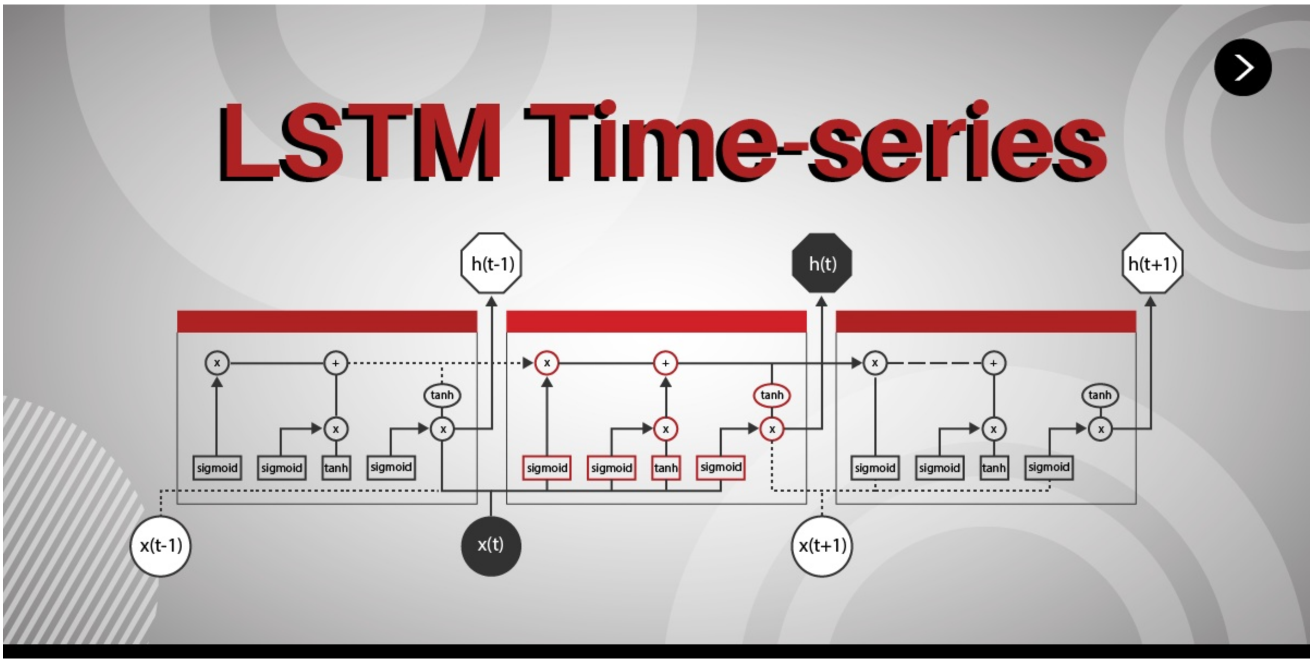


In [ ]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline

import os
from sklearn.preprocessing import MinMaxScaler
from keras.layers import Dense, LSTM, Dropout, GRU, Bidirectional
from keras.models import Sequential
from keras.optimizers import SGD
from sklearn.metrics import mean_squared_error
import math

import warnings
warnings.filterwarnings("ignore")
```

In [2]:
```
dataset = pd.read_csv('/kaggle/input/indian-energy-stock-price-data/The Tata Power Company Limited (TATAPOWER.N
dataset.head()
```

Out[2]:

Date	Open	High	Low	Close	Adj Close	Volume
2000-01-24	7.305179	7.623635	6.996374	7.092876	4.329916	391765.0
2000-01-25	7.092876	7.662236	6.986724	7.662236	4.677489	1479196.0
2000-01-26	7.662236	7.662236	7.662236	7.662236	4.677489	0.0
2000-01-27	8.106144	8.106144	7.430632	7.556084	4.612687	1148622.0
2000-01-28	7.527133	8.106144	7.358255	7.889015	4.815927	1659535.0

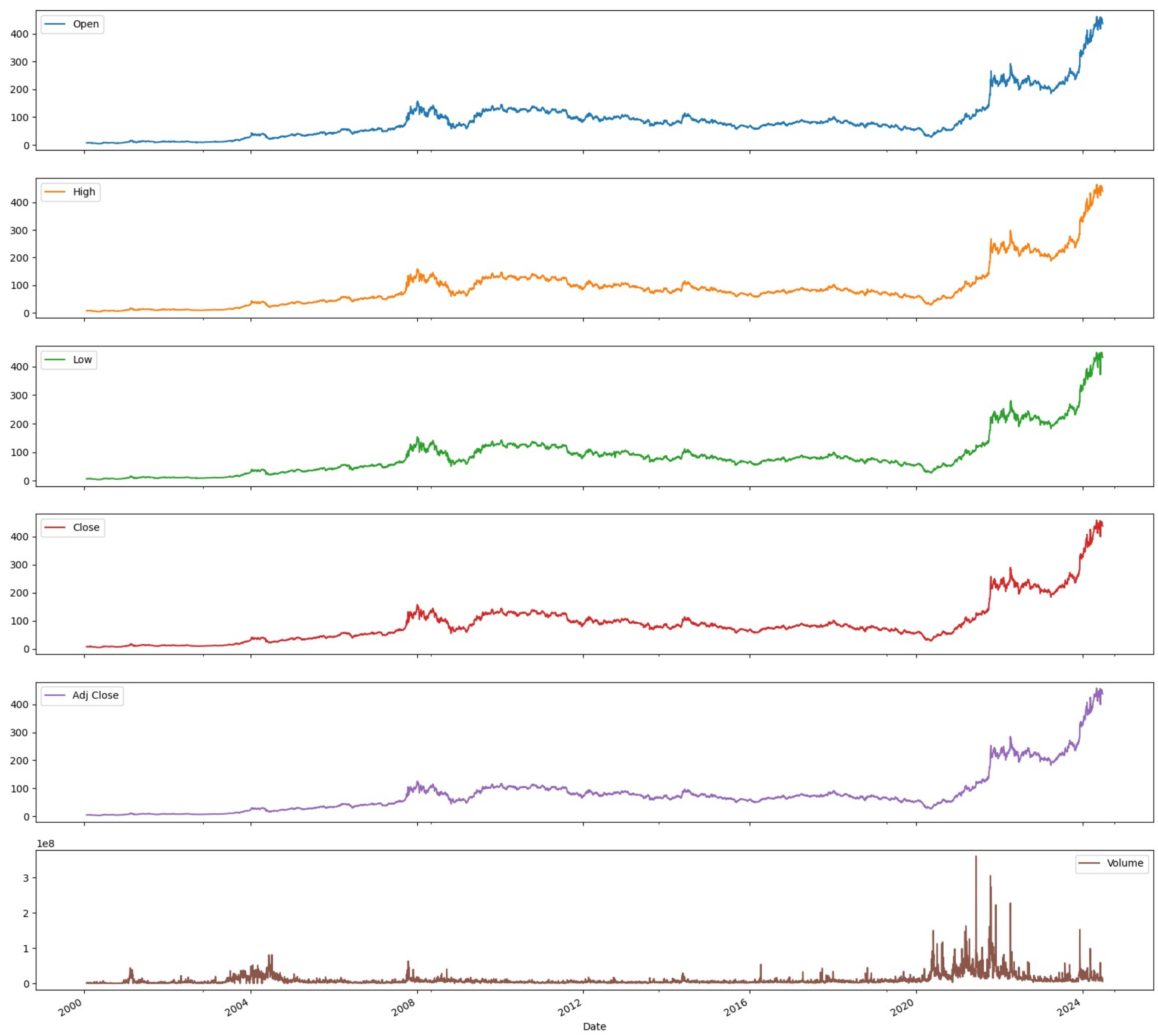
In [3]:
```
dataset.isnull().sum()
```

Out[3]:
```
Open         10
High         10
Low          10
Close        10
Adj Close    10
Volume       10
dtype: int64
```

In [4]:
```
dataset = dataset.dropna(how='all')
```

In [5]:
```
dataset.plot(subplots=True, figsize=(20,20))
plt.suptitle('Tata Power stock price from 2000 to 2024',color="Red")
plt.savefig('Tata_Power_stocks.png')
plt.show()
```

Tata Power stock price from 2000 to 2024



In [6]:
```python
def plot_predictions(test,predicted):
    plt.plot(test, color='red',label='Real Tata Power Stock Price')
    plt.plot(predicted, color='blue',label='Predicted Tata Power Stock Price')
    plt.xlabel('Date')
    plt.ylabel('Tata Power Stock Price')
    plt.title('Tata Power Stock Price Prediction')
    plt.legend()
    plt.show()

def return_rmse(test,predicted):
    rmse = math.sqrt(mean_squared_error(test, predicted))
    print("The root mean squared error is {}.".format(rmse))
```

In [7]:
```python
training_set = dataset[:'2022'].iloc[:,1:2].values
test_set = dataset['2023':].iloc[:,1:2].values
```

In [8]:
```python
training_set.shape
```

Out[8]:
```
(5730, 1)
```

In [9]:
```python
test_set.shape
```

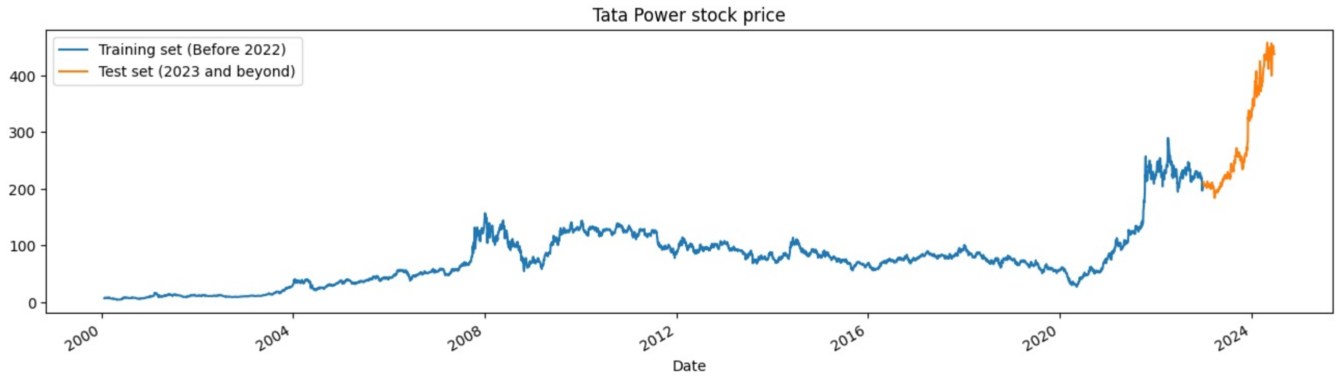
Out[9]:
```
(361, 1)
```

In [10]:
```python
test_set.shape[0] / dataset.shape[0]
```

Out[10]:
```
0.05926777212280414
```

In [11]:
```python
dataset["Close"][:'2022'].plot(figsize=(16,4),legend=True)
```

```
dataset["Close"]['2023':].plot(figsize=(16,4),legend=True)
plt.legend(['Training set (Before 2022)','Test set (2023 and beyond)'])
plt.title('Tata Power stock price')
plt.show()
```



In [12]:
```
# Scaling the training set
sc = MinMaxScaler(feature_range=(0,1))
training_set_scaled = sc.fit_transform(training_set)
```

In [13]:
```
training_set
```

Out[13]:
```
array([[  7.623635],
       [  7.662236],
       [  7.662236],
       ...,
       [210.300003],
       [208.800003],
       [210.      ]])
```

In [14]:
```
# Since LSTMs store long term memory state, we create a data structure with 60 timesteps and 1 output
# So for each element of training set, we have 60 previous training set elements
X_train = []
y_train = []
for i in range(60,training_set.shape[0]):
    X_train.append(training_set_scaled[i-60:i,0])
    y_train.append(training_set_scaled[i,0])
X_train, y_train = np.array(X_train), np.array(y_train)
```

In [15]:
```
# Reshaping X_train for efficient modelling
X_train = np.reshape(X_train, (X_train.shape[0],X_train.shape[1],1))
```

In [16]:
```
X_train.shape
```

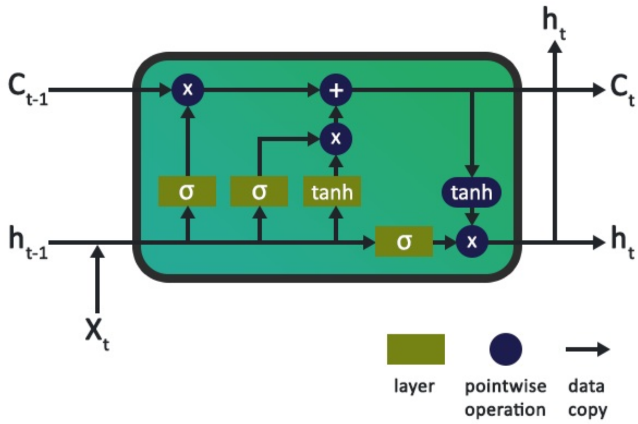
Out[16]:
```
(5670, 60, 1)
```

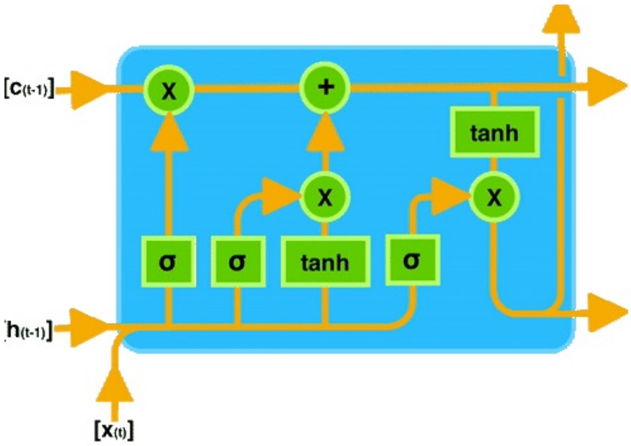
LSTM (Long short-term memory) for Stock Price Prediction in Time Series

Understanding LSTMs and Time Series Data

Time Series Data: Sequences of data points ordered by time, such as historical stock prices.

- **Long Short-Term Memory (LSTM):** A type of Recurrent Neural Network (RNN) that excels at handling time series data. LSTMs address the vanishing gradient problem that limits standard RNNs in capturing long-term dependencies and store information over extended time lags.





Why Use LSTM for Stock Prediction?

- Stock prices exhibit complex patterns, trends, and fluctuations.
- LSTM can learn from historical data and predict future stock movements.
- Keep in mind that stock market prices are highly unpredictable, but we can still model their behavior.

LSTM Architecture for Stock Prediction

- **Input Layer:** Receives a sequence of past stock prices (e.g., closing prices for the last 30 days).
- **Forget Gate:** Decides what information to remember from the previous cell state (past information).
- **Input Gate:** Determines what new information to incorporate from the current input.
- **Cell State:** Internal memory of the LSTM cell, updated based on the forget gate, input gate, and a tanh activation function. This stores the relevant information for the current step, considering both past and present.
- **Output Gate:** Controls what information from the cell state is passed to the next LSTM cell or used for prediction.

LSTM Training Process

- The model is trained using historical stock price data.
- The network learns to identify patterns and relationships within the data sequences.
- The loss function (RMSE, MAE) measures the difference between predicted and actual stock prices.
- The model adjusts its weights (internal parameters) to minimize the loss and improve future predictions.

Advantages of LSTMs for Stock Price Prediction

- **Long-Term Dependency Capture:** LSTMs can effectively capture long-term relationships in stock prices, unlike standard RNNs.
- **Sequential Learning:** LSTMs process information sequentially, considering past data points when making predictions.
- **Flexibility:** LSTMs can handle various input features beyond just closing prices (e.g., trading volume, news sentiment).

Limitations of LSTMs for Stock Prediction

- **Data Dependence:** LSTM accuracy relies heavily on the quality and quantity of historical data.
- **Black Box Nature:** While LSTMs can make good predictions, it can be challenging to interpret the exact reasons behind those predictions.
- **Market Volatility:** Stock prices are inherently volatile, and LSTM predictions should not be considered foolproof.

In [17]:

```
# The LSTM architecture
regressor = Sequential()
# First LSTM layer with Dropout regularisation
regressor.add(LSTM(units=50, return_sequences=True, input_shape=(X_train.shape[1],1)))
regressor.add(Dropout(0.2))
# Second LSTM layer
regressor.add(LSTM(units=50, return_sequences=True))
regressor.add(Dropout(0.2))
# Third LSTM layer
regressor.add(LSTM(units=50, return_sequences=True))
regressor.add(Dropout(0.2))
# Fourth LSTM layer
regressor.add(LSTM(units=50))
regressor.add(Dropout(0.2))
# The output layer
regressor.add(Dense(units=1))

# Compiling the RNN
regressor.compile(optimizer='rmsprop',loss='mean_squared_error')
# Fitting to the training set
regressor.fit(X_train,y_train,epochs=50,batch_size=32)
```

```
Epoch 1/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 6s 12ms/step - loss: 0.0104
Epoch 2/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 0.0027
```

```
Epoch 3/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 0.0020
Epoch 4/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 0.0016
Epoch 5/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 0.0016
Epoch 6/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 0.0014
Epoch 7/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 0.0012
Epoch 8/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 0.0012
Epoch 9/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 0.0011
Epoch 10/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 9.7430e-04
Epoch 11/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 9.2679e-04
Epoch 12/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 9.5647e-04
Epoch 13/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 12ms/step - loss: 8.8511e-04
Epoch 14/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 9.0793e-04
Epoch 15/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 8.0251e-04
Epoch 16/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 9.2836e-04
Epoch 17/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 8.5186e-04
Epoch 18/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 7.9232e-04
Epoch 19/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 7.5607e-04
Epoch 20/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 7.8636e-04
Epoch 21/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 7.3735e-04
Epoch 22/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 7.0365e-04
Epoch 23/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 7.2968e-04
Epoch 24/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 7.4810e-04
Epoch 25/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 7.0717e-04
Epoch 26/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 7.0088e-04
Epoch 27/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 7.1376e-04
Epoch 28/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 7.6382e-04
Epoch 29/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 12ms/step - loss: 6.6805e-04
Epoch 30/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 7.0538e-04
Epoch 31/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 7.9669e-04
Epoch 32/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - loss: 6.4198e-04
Epoch 33/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 6.2193e-04
Epoch 34/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 6.3061e-04
Epoch 35/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 7.0633e-04
Epoch 36/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 6.2920e-04
Epoch 37/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 6.0973e-04
Epoch 38/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 5.9423e-04
Epoch 39/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 6.1865e-04
Epoch 40/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 5.8394e-04
Epoch 41/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 5.9194e-04
Epoch 42/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 5.7750e-04
Epoch 43/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 5.5370e-04
Epoch 44/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 12ms/step - loss: 5.9870e-04
Epoch 45/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 5.9592e-04
Epoch 46/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 6.0716e-04
Epoch 47/50
```

```
178/178 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - loss: 6.1686e-04
Epoch 48/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 5.9089e-04
Epoch 49/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 6.2591e-04
Epoch 50/50
178/178 ━━━━━━━━━━━━━━━━━━━━ 2s 11ms/step - loss: 5.8955e-04
```

Out[17]:

```
<keras.src.callbacks.history.History at 0x7e583afcc670>
```

In [18]:

```python
# Now to get the test set ready in a similar way as the training set.
# The following has been done so forst 60 entires of test set have 60 previous values which
#is impossible to get unless we take the whole
# 'Close' attribute data for processing
dataset_total = pd.concat((dataset["Close"][:'2022'],dataset["Close"]['2023':]),axis=0)
test_inputs = dataset_total[len(dataset_total)-len(test_set) - 60:].values
test_inputs = test_inputs.reshape(-1,1)
test_inputs  = sc.transform(test_inputs)
```

In [19]:

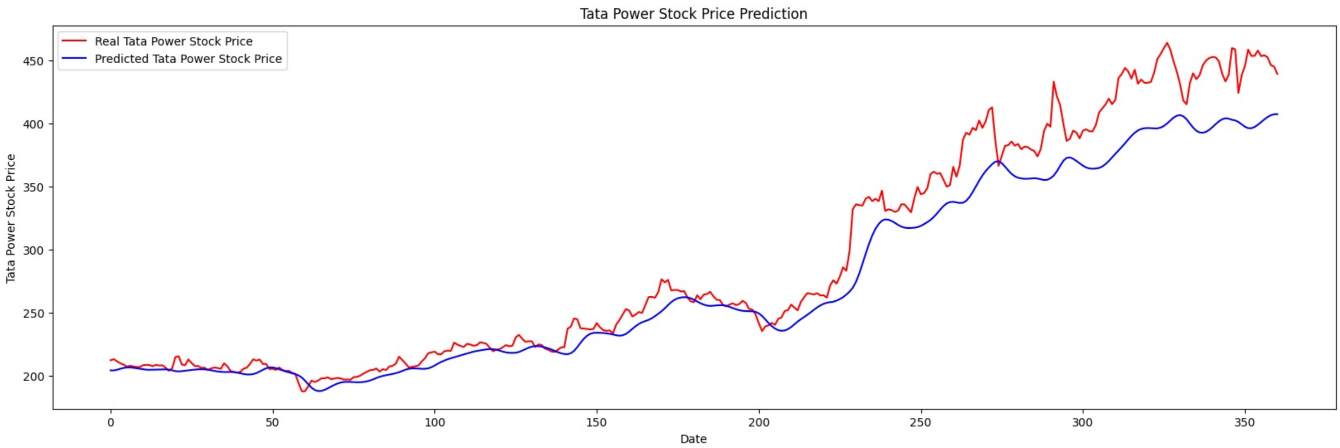
```python
# Preparing X_test and predicting the prices
X_test = []
for i in range(60,test_inputs.shape[0]):
    X_test.append(test_inputs[i-60:i,0])
X_test = np.array(X_test)
X_test = np.reshape(X_test, (X_test.shape[0],X_test.shape[1],1))
predicted_stock_price = regressor.predict(X_test)
predicted_stock_price = sc.inverse_transform(predicted_stock_price)
```

```
12/12 ━━━━━━━━━━━━━━━━━━━━ 1s 31ms/step
```

In [20]:

```python
from pylab import rcParams
rcParams['figure.figsize'] = 20, 6

# Visualizing the results for LSTM
plot_predictions(test_set,predicted_stock_price)
```



In [21]:

```python
# Evaluating our model
return_rmse(test_set,predicted_stock_price)
```

```
The root mean squared error is 24.970305515335173.
```

GRU (Gated Recurrent Unit) for Stock Price Prediction in Time Series

What is GRU?

GRU (Gated Recurrent Unit) is a type of Recurrent Neural Network (RNN) designed to address the vanishing gradient problem that can hinder traditional RNNs in capturing long-term dependencies within time series data like stock prices. While similar to LSTMs (Long Short-Term Memory), GRUs offer a simpler architecture and potentially faster training times.

How Does GRU Work?

- **Input Layer:** Receives a sequence of past stock prices (e.g., closing prices for the last 30 days).
- **Reset Gate:** Determines how much of the previous cell state (past information) to forget.
- **Update Gate:** Controls how much of the current input and the filtered previous cell state to incorporate into the current cell state (effectively combining past and present information).
- **Hidden State:** This is the output of the GRU unit, containing the information relevant for the current time step.

Training Process

Similar to LSTMs, a GRU model is trained using historical stock price data. The network learns to identify patterns and relationships within the data sequences. A loss function (e.g., Mean Squared Error) measures the difference between predicted and actual stock prices. The model then adjusts its internal parameters (weights) to minimize this loss and improve future predictions.

Advantages of GRUs for Stock Prediction

- **Long-Term Dependency Capture:** GRUs can effectively capture long-term dependencies in stock prices, unlike standard RNNs.
- **Sequential Learning:** GRUs process information sequentially, considering past data points when making predictions.
- **Simpler Architecture:** Compared to LSTMs, GRUs have fewer gates and parameters, potentially leading to faster training times.

Limitations of GRUs for Stock Prediction

- **Data Dependence:** GRU accuracy relies heavily on the quality and quantity of historical data.
- **Black Box Nature:** While GRUs can make good predictions, it can be challenging to interpret the exact reasons behind those predictions.
- **Market Volatility:** Stock prices are inherently volatile, and GRU predictions should not be considered foolproof.

In [22]:

```
# The GRU architecture
regressorGRU = Sequential()
# First GRU layer with Dropout regularisation
regressorGRU.add(GRU(units=50, return_sequences=True, input_shape=(X_train.shape[1],1), activation='tanh'))
regressorGRU.add(Dropout(0.2))
# Second GRU layer
regressorGRU.add(GRU(units=50, return_sequences=True, input_shape=(X_train.shape[1],1), activation='tanh'))
regressorGRU.add(Dropout(0.2))
# Third GRU layer
regressorGRU.add(GRU(units=50, return_sequences=True, input_shape=(X_train.shape[1],1), activation='tanh'))
regressorGRU.add(Dropout(0.2))
# Fourth GRU layer
regressorGRU.add(GRU(units=50, activation='tanh'))
regressorGRU.add(Dropout(0.2))
# The output layer
regressorGRU.add(Dense(units=1))
# Compiling the RNN
regressorGRU.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9, nesterov=False),loss='mean_squared_error')
# Fitting to the training set
regressorGRU.fit(X_train,y_train,epochs=50,batch_size=150)
```

```
Epoch 1/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - loss: 0.0506
Epoch 2/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 0.0140
Epoch 3/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 0.0026
Epoch 4/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 0.0014
Epoch 5/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 0.0011
Epoch 6/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 0.0011
Epoch 7/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 0.0011
Epoch 8/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - loss: 0.0013
Epoch 9/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 0.0011
Epoch 10/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 0.0012
Epoch 11/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 9.8385e-04
```

```
Epoch 12/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 0.0011
Epoch 13/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 9.7906e-04
Epoch 14/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 0.0011
Epoch 15/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 0.0012
Epoch 16/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - loss: 0.0010
Epoch 17/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 0.0010
Epoch 18/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 9.6226e-04
Epoch 19/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 0.0011
Epoch 20/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 9.3911e-04
Epoch 21/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 8.9698e-04
Epoch 22/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - loss: 9.2506e-04
Epoch 23/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - loss: 9.6802e-04
Epoch 24/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - loss: 0.0010
Epoch 25/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 9.3294e-04
Epoch 26/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 9.3587e-04
Epoch 27/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 9.6371e-04
Epoch 28/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 9.4381e-04
Epoch 29/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - loss: 9.0767e-04
Epoch 30/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - loss: 8.8853e-04
Epoch 31/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - loss: 8.9117e-04
Epoch 32/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 8.2408e-04
Epoch 33/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 8.6773e-04
Epoch 34/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 8.7028e-04
Epoch 35/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 8.3005e-04
Epoch 36/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 8.2552e-04
Epoch 37/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 8.3844e-04
Epoch 38/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 8.2529e-04
Epoch 39/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 8.1912e-04
Epoch 40/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 8.7000e-04
Epoch 41/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 8.2698e-04
Epoch 42/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 8.3476e-04
Epoch 43/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 8.0036e-04
Epoch 44/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 8.4177e-04
Epoch 45/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 8.6852e-04
Epoch 46/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 7.7520e-04
Epoch 47/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 7.6955e-04
Epoch 48/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 7.8813e-04
Epoch 49/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 7.6915e-04
Epoch 50/50
38/38 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - loss: 8.4655e-04
```

Out[22]:

```
<keras.src.callbacks.history.History at 0x7e5842b2e6e0>
```

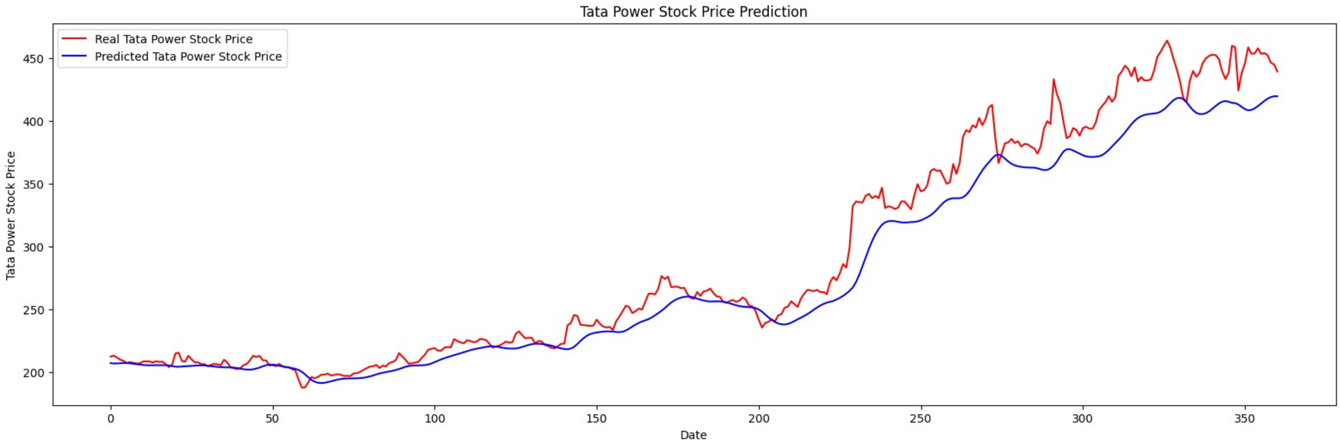
In [23]:

```python
# Preparing X_test and predicting the prices
X_test = []
for i in range(60,test_inputs.shape[0]):
    X_test.append(test_inputs[i-60:i,0])
X_test = np.array(X_test)
X_test = np.reshape(X_test, (X_test.shape[0],X_test.shape[1],1))
GRU_predicted_stock_price = regressorGRU.predict(X_test)
GRU_predicted_stock_price = sc.inverse_transform(GRU_predicted_stock_price)
```

```
12/12 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step
```

In [24]:
```python
from pylab import rcParams
rcParams['figure.figsize'] = 20, 6

# Visualizing the results for GRU
plot_predictions(test_set,GRU_predicted_stock_price)
```



In [25]:
```python
# Evaluating GRU
return_rmse(test_set,GRU_predicted_stock_price)
```

```
The root mean squared error is 21.796039607707403.
```

GRU algorithm is better in this case. The RMSE is lower, and we can see a similar trend between the real stock prices and the predicted stock prices by the GRU algorithm.

https://www.kaggle.com/code/pythonafroz/predict-price-of-tata-power-stocks-with-lstm-gru

In [ ]:
```python

```