simple-time-series-lstm-project

January 24, 2024

LSTM (Long Short-Term Memory)

LSTM, or Long Short-Term Memory, is a type of recurrent neural network (RNN) architecture designed to address the vanishing gradient problem associated with traditional RNNs. Proposed by Sepp Hochreiter and Jürgen Schmidhuber in 1997, LSTMs are particularly effective in capturing long-range dependencies and patterns in sequential data, making them well-suited for tasks such as time series forecasting, natural language processing, and speech recognition.

Key Components: 1. **Memory Cells:** LSTMs have memory cells that allow them to store information for long durations. These cells maintain a constant value over time, enabling the network to capture and remember important patterns.

2. **Gates:** LSTMs use three types of gates to control the flow of information:
    - **Forget Gate:** Determines what information from the previous state should be discarded.
    - **Input Gate:** Modifies the current state by adding new information.
    - **Output Gate:** Selects the information to be output as the final prediction.
3. **Cell State:** The memory cells also maintain a cell state that runs through the entire sequence. This cell state can be seen as a conveyor belt that carries information across time steps.

How LSTMs Work: 1. **Input Processing:** At each time step, the LSTM receives an input and the previous hidden state and cell state.

2. **Gating Mechanisms:** The forget, input, and output gates determine how information flows within the cell. These gates are regulated by sigmoid and tanh activation functions.

3. **Updating Cell State:** The cell state is updated by forgetting certain information, adding new information, and outputting relevant information.

4. **Hidden State:** The hidden state is computed based on the updated cell state and is passed to the next time step.

Advantages of LSTMs: - **Long-Term Dependencies:** LSTMs excel at capturing long-term dependencies in sequential data, making them suitable for tasks requiring memory of past information. - **Resistance to Vanishing Gradient:** The design of LSTMs helps mitigate the vanishing gradient problem, allowing for effective learning even in the presence of long sequences.

Applications: - **Time Series Forecasting:** LSTMs are widely used for predicting future values in time series data, such as stock prices or energy consumption. - **Natural Language Processing:** LSTMs play a crucial role in tasks like language translation, sentiment analysis, and text generation. - **Speech Recognition:** Due to their ability to handle sequential data, LSTMs contribute to accurate speech recognition systems.

In summary, LSTMs are a powerful variant of RNNs, offering enhanced capabilities to model and learn intricate patterns in sequential data, making them indispensable in various machine learning applications.

Below is an example.

[1]:
```
# Import libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")
```

[2]:
```
#Read the CSV file into a DataFrame using pandas
df = pd.read_csv('Electric_production.csv',index_col='DATE',parse_dates=True)
df.index.freq='MS'  #  Set the frequency of the DataFrame's index to 'MS'␣
 ↪(Month Start)

# Print first 5 rows
df.head()
```

[2]:
```
            IPG2211A2N
DATE
1985-01-01     72.5052
1985-02-01     70.6720
1985-03-01     62.4502
1985-04-01     57.4714
1985-05-01     55.3151
```

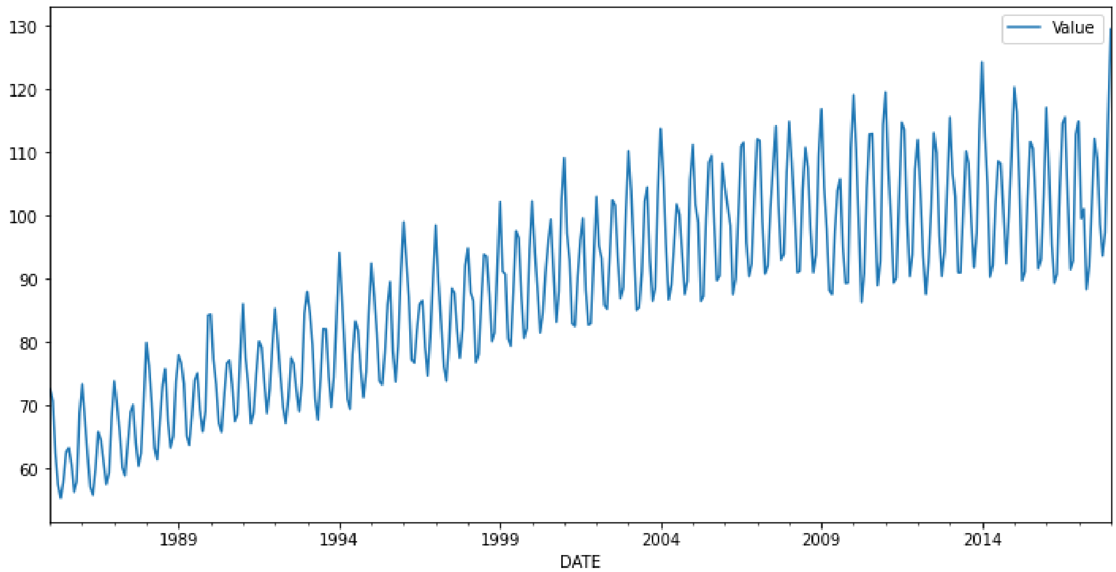
[3]:
```
df.rename(columns={'IPG2211A2N': 'Value'}, inplace=True)
```

[4]:
```
# Print first 5 rows
df.head()
```

[4]:
```
              Value
DATE
1985-01-01  72.5052
1985-02-01  70.6720
1985-03-01  62.4502
1985-04-01  57.4714
1985-05-01  55.3151
```

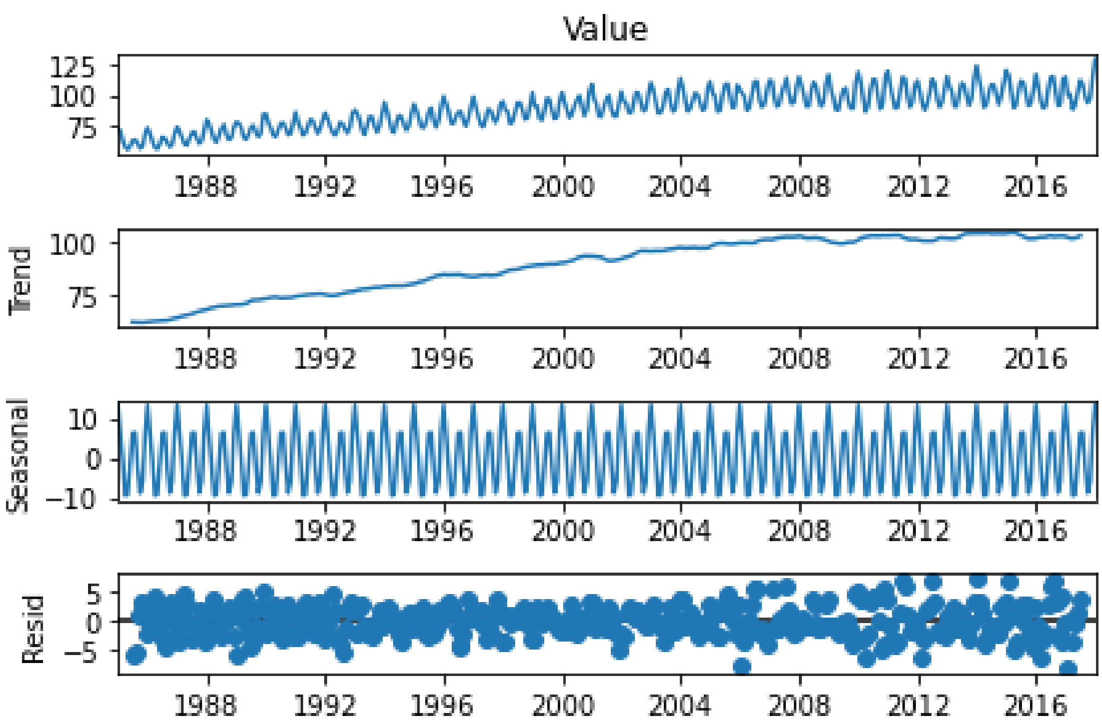
[5]:
```
# Plotting the DataFrame
df.plot(figsize=(12,6))
```

[5]: <Axes: xlabel='DATE'>



Below, The code cell utilizes the seasonal decomposition function from the statsmodels library to decompose the time series data into its three main components: trend, seasonality, and residual.

[6]:
```
from statsmodels.tsa.seasonal import seasonal_decompose
results = seasonal_decompose(df['Value'])
results.plot();
```



NB: RNN(LSTM) is not affected by seasonality

[7]:
```
# Get number of rows in the column
len(df)
```

[7]:
```
397
```

[8]:
```
# Split the data into testing and training sets
train = df.iloc[:300]
test = df.iloc[300:]
```

[ ]:
```

```

Below, we import the MinMaxScaler class from the sklearn.preprocessing module and then create an instance of this scaler. The MinMaxScaler is commonly used for feature scaling, specifically for scaling numerical features to a specified range, usually between 0 and 1

[9]:
```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
```

[10]:
```
df.head(),df.tail()
```

[10]:
```
(              Value
 DATE
 1985-01-01  72.5052
 1985-02-01  70.6720
 1985-03-01  62.4502
 1985-04-01  57.4714
 1985-05-01  55.3151,
               Value
 DATE
 2017-09-01   98.6154
 2017-10-01   93.6137
 2017-11-01   97.3359
 2017-12-01  114.7212
 2018-01-01  129.4048)
```

NB: The purpose of scaling is to normalize the numerical values, typically bringing them within a specific range (commonly between 0 and 1). This can be beneficial for certain machine learning algorithms that are sensitive to the scale of input features.

[11]:
```
scaler.fit(train) # fit the scaler on the training data (train), to determine␣
 ↪the scaling parameters.
scaled_train = scaler.transform(train) # then transform the training data using␣
 ↪the fitted scaler.
```

```
scaled_test = scaler.transform(test) # Also transform the testing data using
 ↪the same scaler
```

[12]:
```
# Display the first 10 elements of the scaled_train array
scaled_train[:10]
```

[12]:
```
array([[0.27943885],
       [0.24963871],
       [0.11598677],
       [0.03505238],
       [0.        ],
       [0.04511473],
       [0.11875025],
       [0.12896377],
       [0.08565994],
       [0.01626068]])
```

[13]:
```
# Importing the TimeseriesGenerator class from the keras.preprocessing.sequence
 ↪module
from keras.preprocessing.sequence import TimeseriesGenerator
```

```
WARNING:tensorflow:From C:\Users\christopher.wachira_\anaconda3\lib\site-
packages\keras\src\losses.py:2976: The name
tf.losses.sparse_softmax_cross_entropy is deprecated. Please use
tf.compat.v1.losses.sparse_softmax_cross_entropy instead.
```

[ ]:
```
```

Here's a breakdown of the cell below: This code block is defining a TimeseriesGenerator instance named generator.

n_input: This variable is set to 3, indicating that the generator will consider 3 time steps for each input sample. In time series data, this typically means using the previous 3 data points to predict the next data point.

n_features: This is set to 1, suggesting that each data point has one feature.

TimeseriesGenerator: This is a class from the Keras library used for generating batches of temporal data. It takes the scaled training data (scaled_train) as input for both x (input sequences) and y (target values). The length parameter specifies the length of the input sequences, and batch_size determines the number of samples in each batch.

This generator is used to create input-output pairs for training a time series model, considering the specified number of time steps and features.

[14]:
```
# define generator
n_input = 3
n_features = 1
```

```
generator = TimeseriesGenerator(scaled_train, scaled_train, length=n_input,␣
  ↪batch_size=1)
```

[ ]:

[15]:
```
# Extract the first batch of input-output pairs from the generator
X, y = generator[0]

# Display the input sequence as a flattened array
print(f'Given the Array: \n{X.flatten()}')

# Display the corresponding target value for prediction
print(f'Predict this y: \n {y}')
```

```
Given the Array:
[0.27943885 0.24963871 0.11598677]
Predict this y:
 [[0.03505238]]
```

[ ]:

[16]:
```
# retrieve the shape of the variable X
X.shape
```

[16]:
```
(1, 3, 1)
```

In the cell above, X is the input sequence obtained from the first batch of input-output pairs generated by the TimeseriesGenerator. The result (1, 3, 1) indicates the shape of X:

The first dimension has a size of 1, representing the batch size. The second dimension has a size of 3, which corresponds to the number of time steps (n_input) considered for each input sample. The third dimension has a size of 1, indicating the number of features (n_features) for each data point. Thus, the shape (1, 3, 1) signifies that we have a batch of one input sequence, with each sequence consisting of 3 time

[ ]:

[17]:
```
# We do the same thing, but now instead for 12 months
# Set the number of time steps to 12 for each input sample
n_input = 12

# Create a new TimeseriesGenerator instance with updated parameters
generator = TimeseriesGenerator(scaled_train, scaled_train, length=n_input,␣
  ↪batch_size=1)
```

[ ]:

[18]:
```python
# # Import necessary modules from the Keras library for constructing a neural
↪network model
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
```

[19]:
```python
# Define the neural network model
model = Sequential()  # Create a sequential model

# Add a Long Short-Term Memory (LSTM) layer with 100 units, using ReLU
↪activation function,
# and specifying the input shape as (n_input, n_features)
model.add(LSTM(100, activation='relu', input_shape=(n_input, n_features)))

# Add a Dense layer with 1 unit (for output) to the model
model.add(Dense(1))

# Compile the model using the Adam optimizer and mean squared error loss
↪function
model.compile(optimizer='adam', loss='mse')
```

```
WARNING:tensorflow:From C:\Users\christopher.wachira_\anaconda3\lib\site-
packages\keras\src\backend.py:873: The name tf.get_default_graph is deprecated.
Please use tf.compat.v1.get_default_graph instead.

WARNING:tensorflow:From C:\Users\christopher.wachira_\anaconda3\lib\site-
packages\keras\src\optimizers\__init__.py:309: The name tf.train.Optimizer is
deprecated. Please use tf.compat.v1.train.Optimizer instead.
```

[20]:
```python
# Display a summary of the neural network model architecture
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 100)               40800

 dense (Dense)               (None, 1)                 101

=================================================================
Total params: 40901 (159.77 KB)
Trainable params: 40901 (159.77 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

From the structure, The model is of type Sequential, which means it is a linear stack of layers. The first layer is an LSTM layer named lstm_1 with 100 units, using the ReLU activation function. It has an input shape of (n_input, n_features). The second layer is a Dense layer named dense_1 with 1 unit for output. The total number of parameters in the model is 40,901, and all of them are trainable. The summary also provides the output shape of each layer.

[ ]:

[21]:
```
# fit model
model.fit(generator,epochs=50)
```

```
Epoch 1/50
WARNING:tensorflow:From C:\Users\christopher.wachira_\anaconda3\lib\site-
packages\keras\src\utils\tf_utils.py:492: The name tf.ragged.RaggedTensorValue
is deprecated. Please use tf.compat.v1.ragged.RaggedTensorValue instead.

288/288 [==============================] - 4s 8ms/step - loss: 0.0315
Epoch 2/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0188
Epoch 3/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0144
Epoch 4/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0119
Epoch 5/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0089
Epoch 6/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0071
Epoch 7/50
288/288 [==============================] - 3s 9ms/step - loss: 0.0055
Epoch 8/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0050
Epoch 9/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0039
Epoch 10/50
288/288 [==============================] - 3s 9ms/step - loss: 0.0037
Epoch 11/50
288/288 [==============================] - 3s 10ms/step - loss: 0.0035
Epoch 12/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0033
Epoch 13/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0033
Epoch 14/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0028
Epoch 15/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0033
Epoch 16/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0029
Epoch 17/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0027
```

```
Epoch 18/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0027
Epoch 19/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0029
Epoch 20/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0027
Epoch 21/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0025
Epoch 22/50
288/288 [==============================] - 3s 9ms/step - loss: 0.0029
Epoch 23/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0026
Epoch 24/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0024
Epoch 25/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0028
Epoch 26/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0024
Epoch 27/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0024
Epoch 28/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0027
Epoch 29/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0024
Epoch 30/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0024
Epoch 31/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0022
Epoch 32/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0025
Epoch 33/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0021
Epoch 34/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0023
Epoch 35/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0022
Epoch 36/50
288/288 [==============================] - 3s 10ms/step - loss: 0.0024
Epoch 37/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0023
Epoch 38/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0020
Epoch 39/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0022
Epoch 40/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0023
Epoch 41/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0022
```

```
Epoch 42/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0020
Epoch 43/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0021
Epoch 44/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0021
Epoch 45/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0023
Epoch 46/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0021
Epoch 47/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0020
Epoch 48/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0020
Epoch 49/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0021
Epoch 50/50
288/288 [==============================] - 2s 8ms/step - loss: 0.0021
```

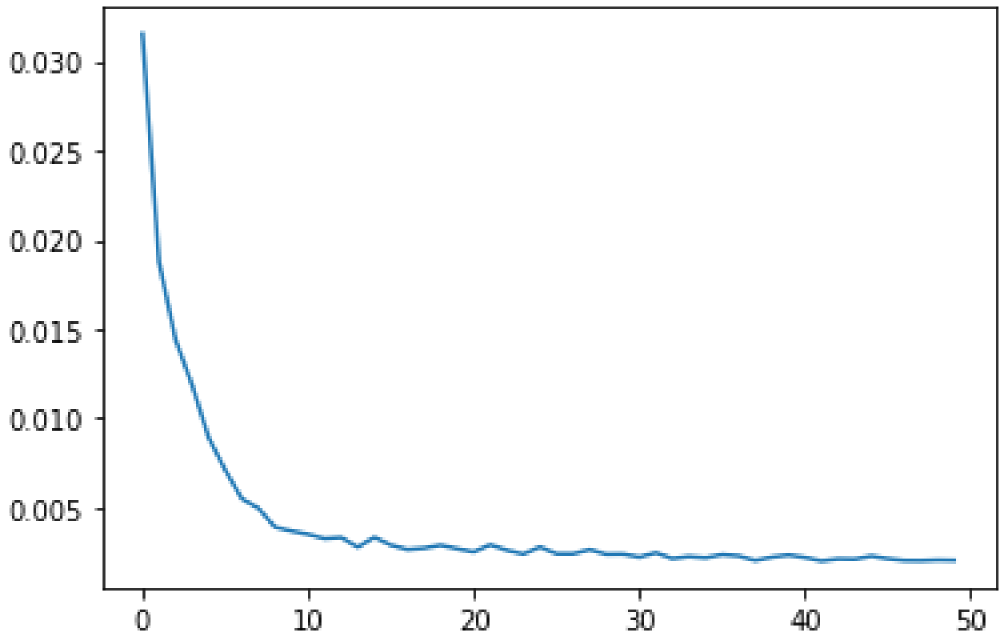
[21]:
```
<keras.src.callbacks.History at 0x1978d0e1d30>
```

Plot change in training loss across epochs.

[22]:
```
# Access the training loss history from the model's history attribute
loss_per_epoch = model.history.history['loss']

# Plot the training loss over epochs
plt.plot(range(len(loss_per_epoch)), loss_per_epoch)
```

[22]:
```
[<matplotlib.lines.Line2D at 0x1978f5ebcd0>]
```



We should've trained around 20 epochs since from 20, the loss is relatively the same

[ ]:

```

```

[23]:

```
# Select the last 12 data points from the scaled training data for further␣
↪processing
last_train_batch = scaled_train[-12:]
```

[24]:

```
# Reshape the last_train_batch array to match the input shape expected by the␣
↪model
last_train_batch = last_train_batch.reshape((1, n_input, n_features))
```

[25]:

```
# Predict using the trained model and the reshaped last_train_batch
model.predict(last_train_batch)
```

```
1/1 [==============================] - 0s 493ms/step
```

[25]:

```
array([[1.0224808]], dtype=float32)
```

[26]:

```
# Access the first element of the 'scaled_test' array and print the result
scaled_test[0]
```

[26]:

```
array([1.03551893])
```

[27]:

```
# Initialize an empty list to store test predictions
test_predictions = []
```

```python
# Take the last 'n_input' data points from the scaled training data as the␣
↪initial batch for evaluation
first_eval_batch = scaled_train[-n_input:]

# Reshape the initial batch to match the input shape expected by the model
current_batch = first_eval_batch.reshape((1, n_input, n_features))

# Iterate over each time step in the test data
for i in range(len(test)):

    # get the prediction value for the first batch
    current_pred = model.predict(current_batch)[0]

    # append the prediction into the array
    test_predictions.append(current_pred)

    # use the prediction to update the batch and remove the first value
    current_batch = np.append(current_batch[:,1:,:],[[current_pred]],axis=1)
```

```
1/1 [==============================] - 0s 58ms/step
1/1 [==============================] - 0s 54ms/step
1/1 [==============================] - 0s 53ms/step
1/1 [==============================] - 0s 51ms/step
1/1 [==============================] - 0s 64ms/step
1/1 [==============================] - 0s 58ms/step
1/1 [==============================] - 0s 53ms/step
1/1 [==============================] - 0s 54ms/step
1/1 [==============================] - 0s 55ms/step
1/1 [==============================] - 0s 57ms/step
1/1 [==============================] - 0s 63ms/step
1/1 [==============================] - 0s 42ms/step
1/1 [==============================] - 0s 62ms/step
1/1 [==============================] - 0s 47ms/step
1/1 [==============================] - 0s 47ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 80ms/step
1/1 [==============================] - 0s 93ms/step
1/1 [==============================] - 0s 77ms/step
1/1 [==============================] - 0s 74ms/step
1/1 [==============================] - 0s 71ms/step
1/1 [==============================] - 0s 66ms/step
1/1 [==============================] - 0s 74ms/step
1/1 [==============================] - 0s 70ms/step
1/1 [==============================] - 0s 83ms/step
1/1 [==============================] - 0s 76ms/step
1/1 [==============================] - 0s 79ms/step
```

```
1/1 [==============================] - 0s 80ms/step
1/1 [==============================] - 0s 77ms/step
1/1 [==============================] - 0s 72ms/step
1/1 [==============================] - 0s 72ms/step
1/1 [==============================] - 0s 70ms/step
1/1 [==============================] - 0s 66ms/step
1/1 [==============================] - 0s 66ms/step
1/1 [==============================] - 0s 74ms/step
1/1 [==============================] - 0s 75ms/step
1/1 [==============================] - 0s 72ms/step
1/1 [==============================] - 0s 83ms/step
1/1 [==============================] - 0s 65ms/step
1/1 [==============================] - 0s 82ms/step
1/1 [==============================] - 0s 61ms/step
1/1 [==============================] - 0s 63ms/step
1/1 [==============================] - 0s 46ms/step
1/1 [==============================] - 0s 63ms/step
1/1 [==============================] - 0s 65ms/step
1/1 [==============================] - 0s 64ms/step
1/1 [==============================] - 0s 49ms/step
1/1 [==============================] - 0s 57ms/step
1/1 [==============================] - 0s 48ms/step
1/1 [==============================] - 0s 42ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 40ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 39ms/step
1/1 [==============================] - 0s 40ms/step
1/1 [==============================] - 0s 47ms/step
1/1 [==============================] - 0s 47ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 47ms/step
1/1 [==============================] - 0s 47ms/step
1/1 [==============================] - 0s 39ms/step
1/1 [==============================] - 0s 56ms/step
1/1 [==============================] - 0s 48ms/step
1/1 [==============================] - 0s 63ms/step
1/1 [==============================] - 0s 25ms/step
1/1 [==============================] - 0s 45ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 48ms/step
1/1 [==============================] - 0s 40ms/step
1/1 [==============================] - 0s 47ms/step
1/1 [==============================] - 0s 40ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 45ms/step
```

```
1/1 [==============================] - 0s 47ms/step
1/1 [==============================] - 0s 41ms/step
1/1 [==============================] - 0s 39ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 49ms/step
1/1 [==============================] - 0s 40ms/step
1/1 [==============================] - 0s 39ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 40ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 40ms/step
1/1 [==============================] - 0s 40ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 31ms/step
1/1 [==============================] - 0s 39ms/step
1/1 [==============================] - 0s 40ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 24ms/step
```

[28]:
```
# show an array of predicted values,
test_predictions
```

[28]:
```
[array([1.0224808], dtype=float32),
 array([0.8436816], dtype=float32),
 array([0.6830243], dtype=float32),
 array([0.5395467], dtype=float32),
 array([0.5326084], dtype=float32),
 array([0.71346927], dtype=float32),
 array([0.8391961], dtype=float32),
 array([0.81650734], dtype=float32),
 array([0.6633301], dtype=float32),
 array([0.54867315], dtype=float32),
 array([0.5886578], dtype=float32),
 array([0.8582326], dtype=float32),
 array([1.0055311], dtype=float32),
 array([0.86855817], dtype=float32),
 array([0.69448555], dtype=float32),
 array([0.5473881], dtype=float32),
 array([0.5427575], dtype=float32),
 array([0.74170506], dtype=float32),
 array([0.8766601], dtype=float32),
 array([0.82620454], dtype=float32),
 array([0.6722152], dtype=float32),
```

```
array([0.5498484], dtype=float32),
array([0.6016797], dtype=float32),
array([0.8486148], dtype=float32),
array([0.98534584], dtype=float32),
array([0.86883676], dtype=float32),
array([0.69618237], dtype=float32),
array([0.5510321], dtype=float32),
array([0.5550811], dtype=float32),
array([0.765368], dtype=float32),
array([0.9016484], dtype=float32),
array([0.83600724], dtype=float32),
array([0.6785346], dtype=float32),
array([0.55214787], dtype=float32),
array([0.60785246], dtype=float32),
array([0.84839535], dtype=float32),
array([0.9735124], dtype=float32),
array([0.86037385], dtype=float32),
array([0.6911311], dtype=float32),
array([0.55096924], dtype=float32),
array([0.5689781], dtype=float32),
array([0.7863301], dtype=float32),
array([0.9179859], dtype=float32),
array([0.8410336], dtype=float32),
array([0.6811929], dtype=float32),
array([0.55474085], dtype=float32),
array([0.6131201], dtype=float32),
array([0.85074055], dtype=float32),
array([0.9670268], dtype=float32),
array([0.8527901], dtype=float32),
array([0.68545914], dtype=float32),
array([0.5526302], dtype=float32),
array([0.58463407], dtype=float32),
array([0.80485415], dtype=float32),
array([0.92799664], dtype=float32),
array([0.84138656], dtype=float32),
array([0.68030715], dtype=float32),
array([0.5575031], dtype=float32),
array([0.6205435], dtype=float32),
array([0.85429037], dtype=float32),
array([0.96224856], dtype=float32),
array([0.84598696], dtype=float32),
array([0.6798699], dtype=float32),
array([0.5551826], dtype=float32),
array([0.60147274], dtype=float32),
array([0.8215239], dtype=float32),
array([0.933468], dtype=float32),
array([0.83784044], dtype=float32),
```

```
 array([0.6763749], dtype=float32),
 array([0.56028205], dtype=float32),
 array([0.6313771], dtype=float32),
 array([0.85944366], dtype=float32),
 array([0.957152], dtype=float32),
 array([0.83830047], dtype=float32),
 array([0.6733545], dtype=float32),
 array([0.55824506], dtype=float32),
 array([0.6200638], dtype=float32),
 array([0.83699095], dtype=float32),
 array([0.9352256], dtype=float32),
 array([0.8309519], dtype=float32),
 array([0.6697056], dtype=float32),
 array([0.5631087], dtype=float32),
 array([0.64639664], dtype=float32),
 array([0.8668232], dtype=float32),
 array([0.95086133], dtype=float32),
 array([0.8283564], dtype=float32),
 array([0.6648166], dtype=float32),
 array([0.5616565], dtype=float32),
 array([0.6417042], dtype=float32),
 array([0.8524966], dtype=float32),
 array([0.93359435], dtype=float32),
 array([0.8205651], dtype=float32),
 array([0.66010654], dtype=float32),
 array([0.5661415], dtype=float32),
 array([0.6665845], dtype=float32),
 array([0.8770554], dtype=float32),
 array([0.9428737], dtype=float32)]
```

[29]:
```
# Display the first few rows of the 'test' DataFrame
test.head()
```

[29]:
```
               Value
DATE
2010-01-01  119.0166
2010-02-01  110.5330
2010-03-01   98.2672
2010-04-01   86.3000
2010-05-01   90.8364
```

[30]:
```
# Transform the scaled predictions back to the original data scale using the
 ↪inverse scaler
true_predictions = scaler.inverse_transform(test_predictions)
```

[31]:
```
# Create a new column 'Predictions' in the 'test' DataFrame and assign the
 ↪predicted values
```

```
test['Predictions'] = true_predictions
```

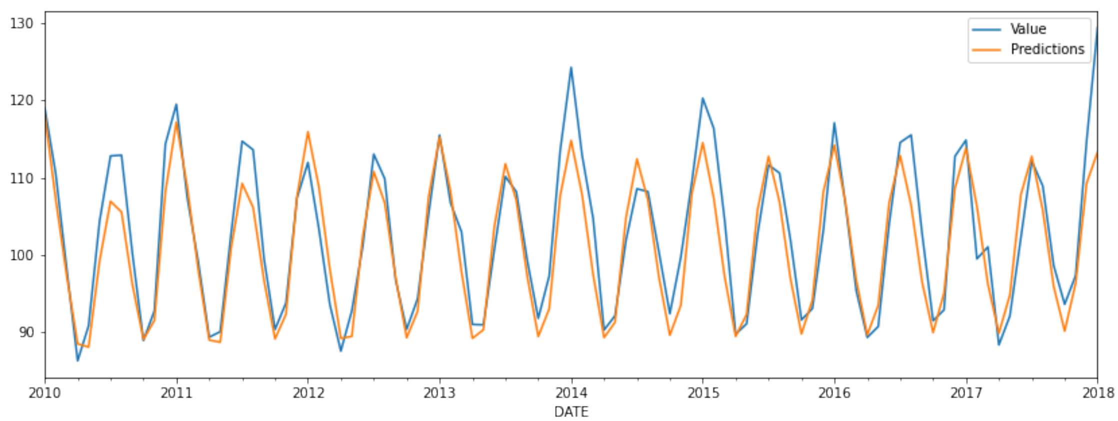
[32]:
```
test.head()
```

[32]:

DATE	Value	Predictions
2010-01-01	119.0166	118.214543
2010-02-01	110.5330	107.215438
2010-03-01	98.2672	97.332364
2010-04-01	86.3000	88.506126
2010-05-01	90.8364	88.079304

[33]:
```
# Display a plot of the 'test' DataFrame with a specified figure size
test.plot(figsize=(14,5))
```

[33]: <Axes: xlabel='DATE'>



[34]:
```
# For accuracy, calculate Root Mean Squared Error (RMSE) between the actual␣
↪production values and the predicted values
from sklearn.metrics import mean_squared_error
from math import sqrt
rmse=sqrt(mean_squared_error(test['Value'],test['Predictions']))
print(rmse)
```

4.09572649017682

[ ]:
```
```