Name : Ajay Shitkar
Roll No : 58
Experiment No. 8
Design and implement RNN for classification of temporal data , sequence to sequence data modelling etc.
Date of Performance: 26/03/24
Date of Submission: 02/04/24

Title: Design and implement RNN for classification of temporal data , sequence to sequence data modelling etc.

Aim: To design and implement RNN for classification of temporal data , sequence to sequence data modelling etc

Objective: To design deep learning models for supervised, unsupervised and sequence learning

Theory:

Recurrent Neural Networks (RNN) model the temporal dependencies present in the data as it contains an implicit memory of previous inputs. Hence, time series data being sequential in nature is often used in RNN. For working with time series data in RNNs, TensorFlow provides a number of APIs and tools, like tf.keras.layers. RNN API, which allows to create of unique RNN cell classes and use them with data. Several RNN cell types are also supported by this API, including Basic RNN, LSTM, and GRU.

Time Series Data: Each data point in a time series is linked to a timestamp, which shows the exact time when the data was observed or recorded. Many fields, including finance, economics, weather forecasting, and machine learning, frequently employ this kind of data.

The fact that time series data frequently display patterns or trends across time, such as seasonality or cyclical patterns, is an essential feature associated with it. To make predictions or learn more about the underlying processes or occurrences being observed, these patterns can be analyzed and modeled.

Code and Output :

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
df = pd.read_csv('/content/monthly_milk_production.csv',index_col='Date',parse_dates=True)
df.index.freq='MS'
```

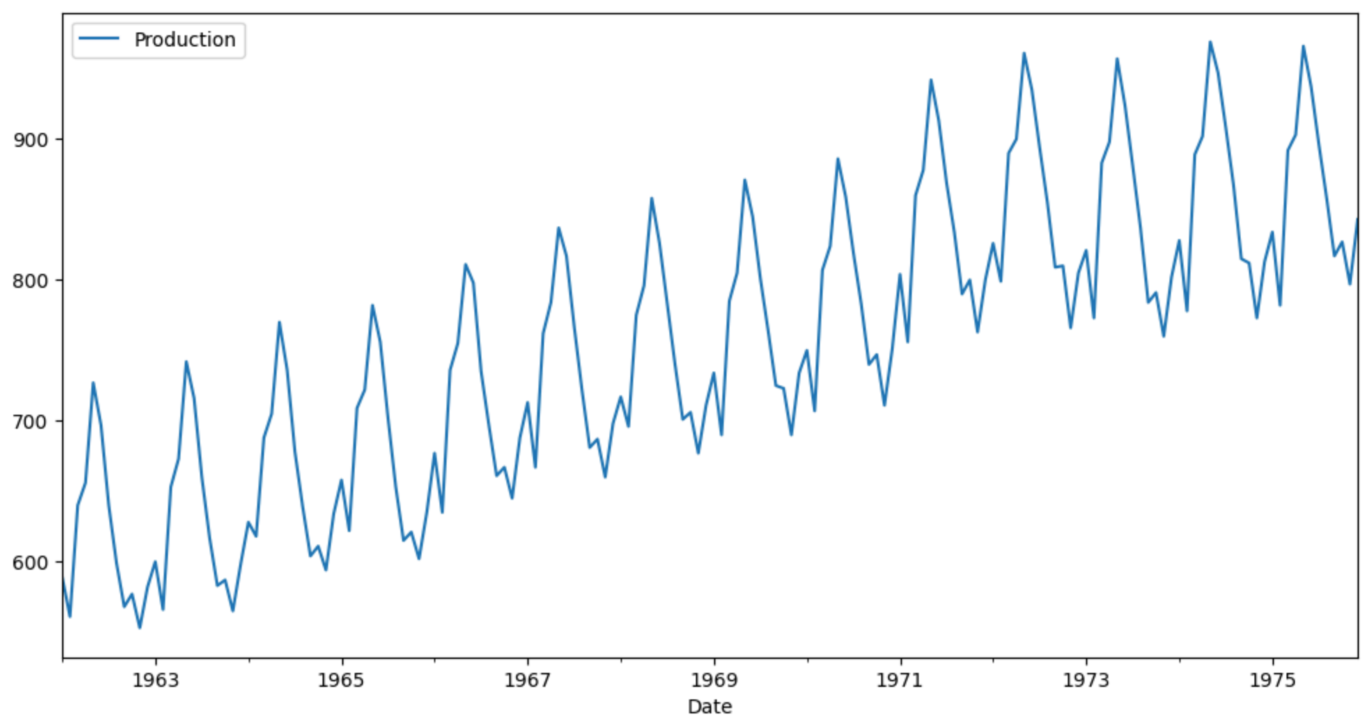
```
df.head()
```

Date	Production
1962-01-01	589
1962-02-01	561
1962-03-01	640
1962-04-01	656
1962-05-01	727

Next steps: Generate code with df View recommended plots

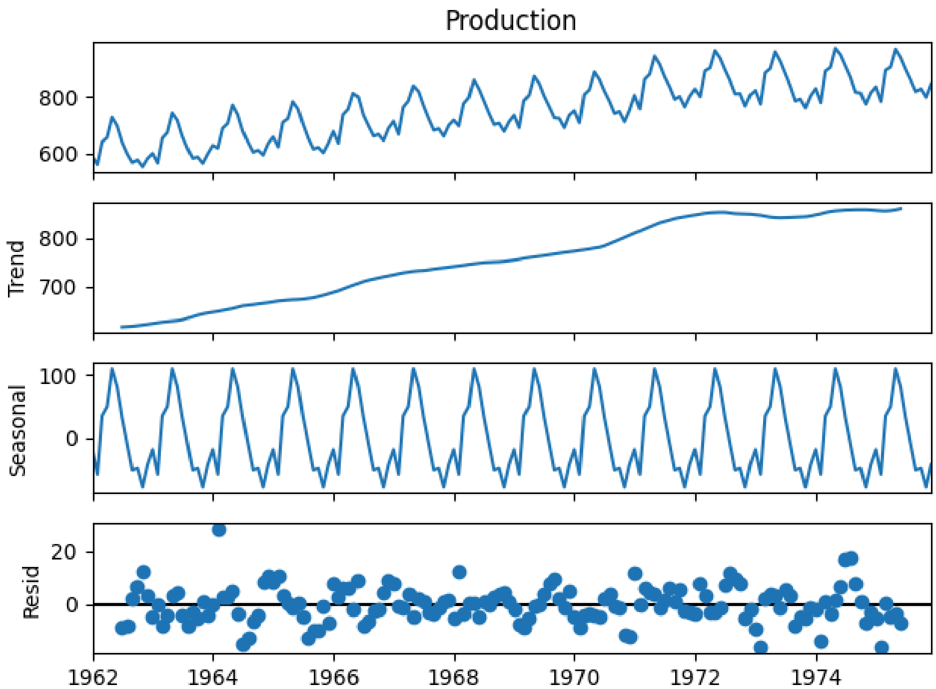
```
df.plot(figsize=(12,6))
```

<Axes: xlabel='Date'>




```
from statsmodels.tsa.seasonal import seasonal_decompose
```

```
results = seasonal_decompose(df['Production'])
results.plot();
```




```
len(df)
```

```
168
```

```
train = df.iloc[:156]
test = df.iloc[156:]
```

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
```

```
df.head(),df.tail()
```

```
(            Production
 Date
 1962-01-01         589
 1962-02-01         561
 1962-03-01         640
 1962-04-01         656
 1962-05-01         727,
             Production
 Date
 1975-08-01         858
 1975-09-01         817
 1975-10-01         827
 1975-11-01         797
 1975-12-01         843)
```

```
scaler.fit(train)
scaled_train = scaler.transform(train)
scaled_test = scaler.transform(test)
```

```
scaled_train[:10]
```

```
array([[0.08653846],
       [0.01923077],
       [0.20913462],
```

```
       [0.24759615],
       [0.41826923],
       [0.34615385],
       [0.20913462],
       [0.11057692],
       [0.03605769],
       [0.05769231]])
```

```
from keras.preprocessing.sequence import TimeseriesGenerator
```

```
# define generator
n_input = 3
n_features = 1
generator = TimeseriesGenerator(scaled_train, scaled_train, length=n_input, batch_size=1)
```

```
X,y = generator[1]
print(f'Given the Array: \n{X.flatten()}')
print(f'Predict this y: \n {y}')
```

```
Given the Array:
[0.01923077 0.20913462 0.24759615]
Predict this y:
 [[0.41826923]]
```

```
X.shape
```

```
(1, 3, 1)
```

```
# We do the same thing, but now instead for 12 months
n_input = 12

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

Start coding or generate with AI.

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

```
# define model
model = Sequential()
model.add(LSTM(100, activation='relu', input_shape=(n_input, n_features)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

```
model.summary()
```

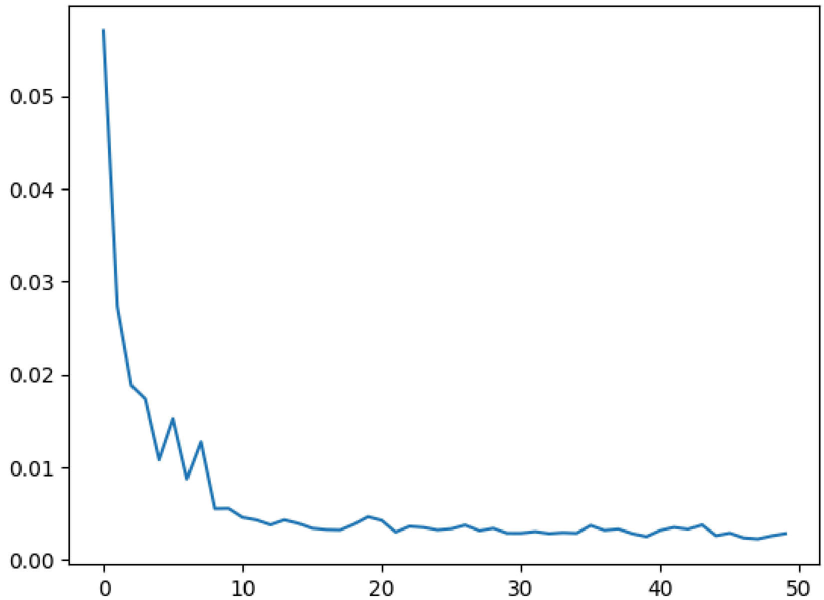
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 100)               40800

 dense (Dense)               (None, 1)                 101

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

```
# fit model
model.fit(generator,epochs=50)
```

```
144/144 [==============================] - 1s 8ms/step - loss: 0.0030
Epoch 23/50
144/144 [==============================] - 1s 8ms/step - loss: 0.0036
Epoch 24/50
144/144 [==============================] - 1s 9ms/step - loss: 0.0035
Epoch 25/50
144/144 [==============================] - 1s 9ms/step - loss: 0.0032
Epoch 26/50
144/144 [==============================] - 1s 8ms/step - loss: 0.0033
Epoch 27/50
144/144 [==============================] - 1s 8ms/step - loss: 0.0037
Epoch 28/50
144/144 [==============================] - 1s 8ms/step - loss: 0.0031
Epoch 29/50
144/144 [==============================] - 1s 8ms/step - loss: 0.0034
Epoch 30/50
144/144 [==============================] - 1s 8ms/step - loss: 0.0029
Epoch 31/50
144/144 [==============================] - 1s 8ms/step - loss: 0.0028
Epoch 32/50
144/144 [==============================] - 1s 9ms/step - loss: 0.0030
Epoch 33/50
144/144 [==============================] - 2s 14ms/step - loss: 0.0028
Epoch 34/50
144/144 [==============================] - 1s 8ms/step - loss: 0.0029
Epoch 35/50
144/144 [==============================] - 1s 8ms/step - loss: 0.0028
Epoch 36/50
144/144 [==============================] - 1s 8ms/step - loss: 0.0037
Epoch 37/50
144/144 [==============================] - 1s 8ms/step - loss: 0.0032
Epoch 38/50
144/144 [==============================] - 1s 8ms/step - loss: 0.0033
Epoch 39/50
144/144 [==============================] - 1s 8ms/step - loss: 0.0028
Epoch 40/50
144/144 [==============================] - 1s 8ms/step - loss: 0.0025
Epoch 41/50
144/144 [==============================] - 1s 8ms/step - loss: 0.0032
Epoch 42/50
144/144 [==============================] - 2s 13ms/step - loss: 0.0035
Epoch 43/50
144/144 [==============================] - 1s 9ms/step - loss: 0.0033
Epoch 44/50
144/144 [==============================] - 1s 8ms/step - loss: 0.0038
Epoch 45/50
144/144 [==============================] - 1s 8ms/step - loss: 0.0026
Epoch 46/50
144/144 [==============================] - 1s 8ms/step - loss: 0.0029
Epoch 47/50
144/144 [==============================] - 1s 8ms/step - loss: 0.0024
Epoch 48/50
144/144 [==============================] - 1s 8ms/step - loss: 0.0022
Epoch 49/50
144/144 [==============================] - 1s 8ms/step - loss: 0.0026
Epoch 50/50
144/144 [==============================] - 1s 8ms/step - loss: 0.0028
<keras.src.callbacks.History at 0x7eda88444f40>
```

```
loss_per_epoch = model.history.history['loss']
plt.plot(range(len(loss_per_epoch)),loss_per_epoch)
```

```
[<matplotlib.lines.Line2D at 0x7eda816e15a0>]
```




```
last_train_batch = scaled_train[-12:]
```

```
last_train_batch = last_train_batch.reshape((1, n_input, n_features))
```

```
model.predict(last_train_batch)
```

```
1/1 [==============================] - 0s 294ms/step
array([[0.6327976]], dtype=float32)
```

```
scaled_test[0]
```

```
array([0.67548077])
```

```
test_predictions = []

first_eval_batch = scaled_train[-n_input:]
current_batch = first_eval_batch.reshape((1, n_input, n_features))

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 34ms/step
1/1 [==============================] - 0s 30ms/step
1/1 [==============================] - 0s 32ms/step
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 38ms/step
1/1 [==============================] - 0s 30ms/step
1/1 [==============================] - 0s 30ms/step
1/1 [==============================] - 0s 34ms/step
1/1 [==============================] - 0s 37ms/step
1/1 [==============================] - 0s 33ms/step
```

```
1/1 [==============================] - 0s 30ms/step
1/1 [==============================] - 0s 32ms/step
```

```
test_predictions
```

```
[array([0.6327976], dtype=float32),
 array([0.6445972], dtype=float32),
 array([0.80612993], dtype=float32),
 array([0.878759], dtype=float32),
 array([0.94510865], dtype=float32),
 array([0.94375926], dtype=float32),
 array([0.89151114], dtype=float32),
 array([0.7969227], dtype=float32),
 array([0.6947224], dtype=float32),
 array([0.6393323], dtype=float32),
 array([0.5977034], dtype=float32),
 array([0.6235139], dtype=float32)]
```

```
test
```

Date	Production
1975-01-01	834
1975-02-01	782
1975-03-01	892
1975-04-01	903
1975-05-01	966
1975-06-01	937
1975-07-01	896
1975-08-01	858
1975-09-01	817
1975-10-01	827
1975-11-01	797
1975-12-01	843

Next steps: Generate code with `test` View recommended plots

Conclusion: Through the design and implementation of Recurrent Neural Networks (RNNs) for classification of temporal data and sequence-to-sequence modeling, it's evident that RNNs excel in capturing temporal dependencies and sequential patterns. The versatility of RNN architectures allows for effective modeling of diverse data types, from time series to natural language sequences. By leveraging techniques such as gated recurrent units (GRUs) or long short-term memory (LSTM) cells, RNNs can mitigate the vanishing gradient problem and retain information over long sequences. The successful application of RNNs in tasks like sentiment analysis, speech recognition, and machine translation underscores their utility in handling sequential data across various domains. As research continues to enhance RNN architectures and training methodologies, their role in advancing sequence modeling and classification tasks remains pivotal, promising continued breakthroughs in data-driven applications.