Time Series Forecasting Using Recurrent Neural Network

This is a time series forecasting problem that I want to solve it using RNN. Different types of RNN are used and their performances are compared.

Import required libraries and Load data from local drive and load dataset

```
#Load data from local drive
from google.colab import files
uploaded = files.upload()
```

```
Choose Files  monthly_mi...duction.csv
• monthly_milk_production.csv(text/csv) - 2201 bytes, last modified: 4/15/2022 - 100% done
Saving monthly_milk_production.csv to monthly_milk_production (1).csv
```

```
#Load data set
df = pd.read_csv('monthly_milk_production.csv', index_col='Date',parse_dates=True)
df.index.freq = 'MS'
```

See how the first five rows of data:

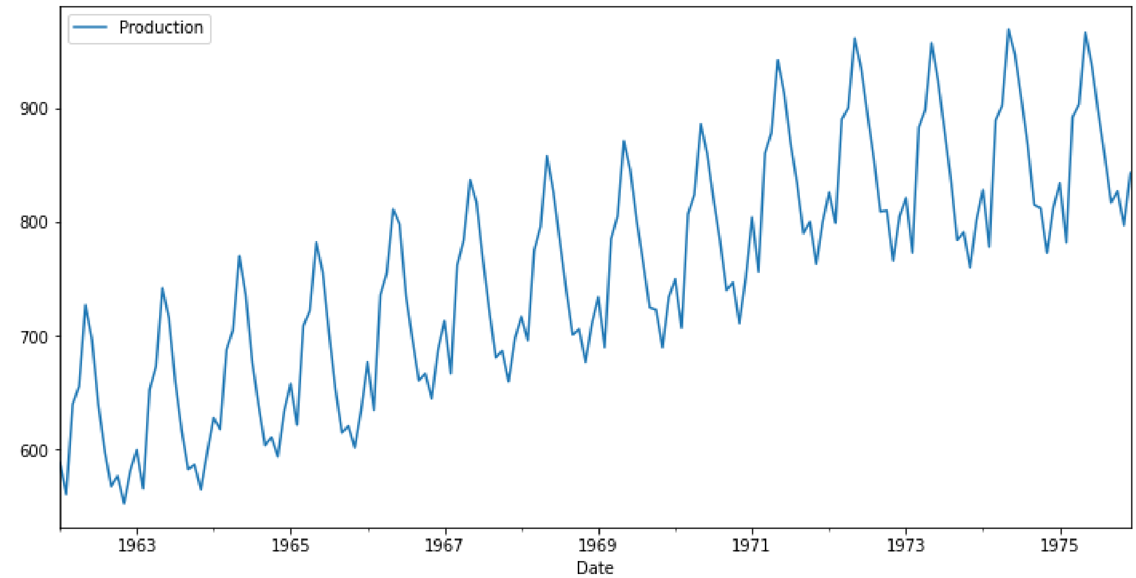
```
df.head()
```

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

Check the seasonality of data:

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

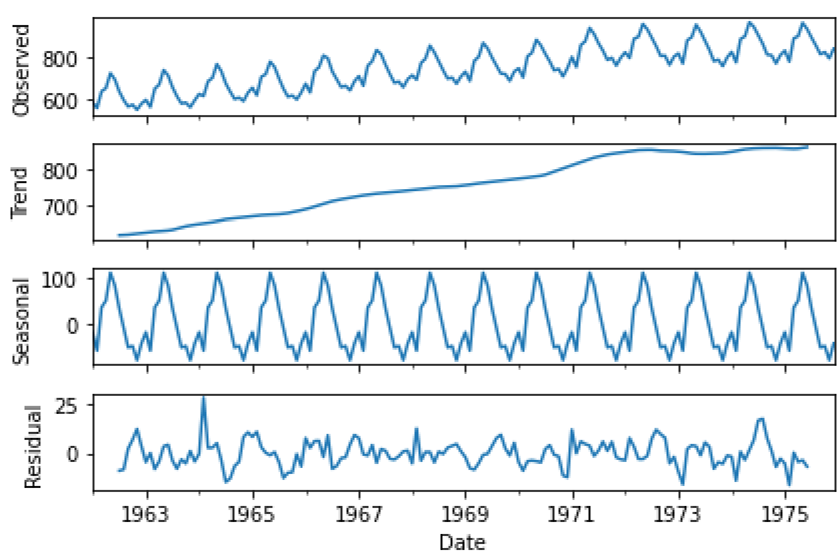
```
<AxesSubplot:xlabel='Date'>
```



We see that there is some sort of seasonality repeating pattern and the general trend increasing with time.

Decompose the feature vector:

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



The above figure shows us different components of the previous main graph. The first graph shows the similar graph as before. The second one shows the trend by isolating the trend and removing seasonality part. We see there is generally increasing trend with time. The season part of the graph, the third graph, shows just the seasonality by subtracting and removing the trend from the original graph and we can see a clear seasonal pattern over here. The fourth graph is the residual part, basically there is what cannot be explained by the trend or seasonality. It is the noise part of the dataset.

Check the length of the dataset to decide on splitting it to train and test parts:

```
len(df)
```

```
168
```

Consider the last 12 data of the whole dataset as the testing dataset. It is the last 12 months. The remaining 156 months are considered as a training dataset.

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

Preprocess the data:

```
[ ] 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)
```

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

```
scaled_train[:10] #All values are in the range of 0 to 1
```

```
array([[0.08653846],
       [0.01923077],
       [0.20913462],
       [0.24759615],
       [0.41826923],
       [0.34615385],
       [0.20913462],
       [0.11057692],
       [0.03605769],
       [0.05769231]])
```

Let's format the model to give it to neural network.

First, assume we have the data of the first three months and we want to predict the fourth month and define generator to generate time series

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

```
[ ] # define generator to generate time series
    n_input = 3
    n_features = 1 #One feature means univariate problem. Since we have only one column of timeseries. If we have multiple c
    generator = TimeseriesGenerator(scaled_train, scaled_train, length=n_input, batch_size=1) #the first three values are ta
    #to predict the fourth value
```

```
[ ] X,y = generator[0] #See the first value of generator and extract input X(three value) and output y(one value)
    print(f'Given the Array: \n {X.flatten()}')
    print(f'Predict this: \n {y}')
```

```
Given the Array:
 [0.08653846 0.01923077 0.20913462]
Predict this:
 [[0.24759615]]
```

We do the same thing, but now instead of 3, we solve for 12 months:

```
[ ] # We do the same thing, but now instead of 3, we solve for 12 months
    n_input = 12
    generator = TimeseriesGenerator(scaled_train, scaled_train, length=n_input, batch_size=1)
```

Call the sequential, the dense and the LSTM classes and make the model:

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

Make the model:

```
[ ] # define model
    model = Sequential() #define model sequential to have layer by layer
    model.add(LSTM(100, activation='relu', input_shape=(n_input, n_features))) #add LSTM layer with 100 nodes
    model.add(Dense(1)) #final output layer and make the final prediction
    model.compile(optimizer='adam', loss='mse') #compile the model
```

Let's see a summary of the model:

```
[ ] #print the model
    model.summary()
```

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

 dense (Dense)               (None, 1)                 101

=================================================================
Total params: 40,901
Trainable params: 40,901
Non-trainable params: 0
_________________________________________________________________
```

The model has 40,901 trainable and 0 non-trainable parameters.

Train the model on the training data using model.fit() syntax. We see loss decreases in each epoch:

```
# fit model
model.fit(generator, epochs=30)
```

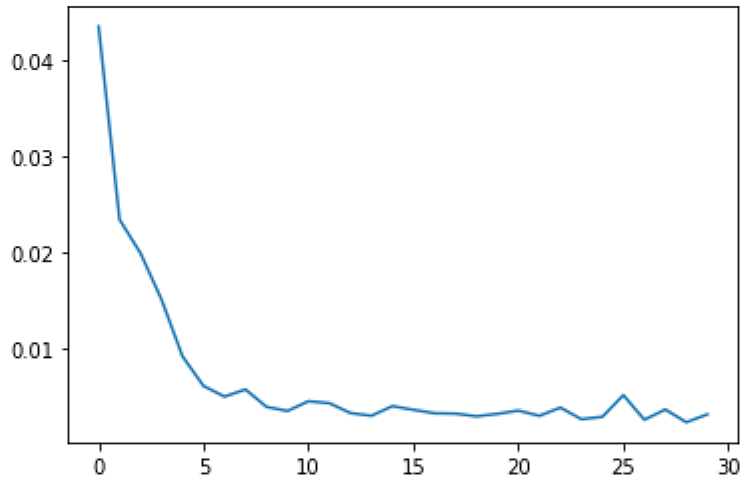
```
Epoch 1/30
144/144 [==============================] - 3s 7ms/step - loss: 0.0436
Epoch 2/30
144/144 [==============================] - 1s 7ms/step - loss: 0.0235
Epoch 3/30
144/144 [==============================] - 1s 7ms/step - loss: 0.0200
Epoch 4/30
144/144 [==============================] - 1s 7ms/step - loss: 0.0152
Epoch 5/30
144/144 [==============================] - 1s 7ms/step - loss: 0.0093
Epoch 6/30
144/144 [==============================] - 1s 7ms/step - loss: 0.0062
Epoch 7/30
144/144 [==============================] - 1s 7ms/step - loss: 0.0051
Epoch 8/30
144/144 [==============================] - 1s 7ms/step - loss: 0.0058
Epoch 9/30
144/144 [==============================] - 1s 7ms/step - loss: 0.0040
Epoch 10/30
144/144 [==============================] - 1s 7ms/step - loss: 0.0036
Epoch 11/30
144/144 [==============================] - 1s 7ms/step - loss: 0.0046
Epoch 12/30
```

Plot the loss per epoch:

We see loss decreases per epoch.

```
loss_per_epoch = model.history.history['loss'] #Display Deep Lea
plt.plot(range(len(loss_per_epoch)), loss_per_epoch) #We can use
```

```
[<matplotlib.lines.Line2D at 0x7f241ca746d0>]
```



The number of epochs are 30 but we see in the last epochs, that loss variances are low.

Lets make the predictions:

taking the last 12 months values in the training set to make the prediction for the first value in th e test set:

```
last_train_batch = scaled_train[-12:] # taking the last 12 months values in the tr
```

```
[ ] last_train_batch = last_train_batch.reshape((1, n_input, n_features)) # reshape it
    #Why we do reshape? becuase that is how the model has been trained on
```

```
model.predict(last_train_batch) # predict the first value in the testing set
```

```
array([[0.68895966]], dtype=float32)
```

Now, we can predict the testing set

Here, we take the 12 values to make the next prediction. Then, using that prediction, it's getting a new input of 12 values, and using that it's again making a prediction

```
[ ] test_predictions = []

    first_eval_batch = scaled_train[-n_input:] # get the last 12 values in the training set
    current_batch = first_eval_batch.reshape((1, n_input, n_features)) # the current input or the current batch of 12 values

    for i in range(len(test)):
      #print(i)
      # get the prediction value for the first batch
      current_pred = model.predict(current_batch)[0] # use the last 12 values in order to make the prediction. 0 means towards row r

      # 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) #taking the current batch while dropping its first :
      # current_batch[:,1:,:] means dropping the first input. axis=1 means appending it along the column
    # Ex: if current batch is [1,2,3], then predictionis [4]. Then next batch will be [2,3,4] to predict [5]. So, [1] was dropping
```

```
test_predictions
```

```
[array([0.68895966], dtype=float32),
 array([0.71596587], dtype=float32),
 array([0.90997845], dtype=float32),
 array([1.0114869], dtype=float32),
 array([1.1111249], dtype=float32),
 array([1.1083733], dtype=float32),
 array([1.0353343], dtype=float32),
 array([0.9079941], dtype=float32),
 array([0.7726714], dtype=float32),
 array([0.7056668], dtype=float32),
 array([0.67349875], dtype=float32),
 array([0.7023535], dtype=float32)]
```

One problem is that the range of test predictions values is that these values are in ranges of 0 to 1. So, we must transform it back to the original scale because the original test values are about 900:

```
[ ] true_predictions = scaler.inverse_transform(test_predictions) #tranfrom test values back to
```

```
test['Predictions'] = true_predictions #add Predictions as an np array to test dataframe
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_gui
  """Entry point for launching an IPython kernel.
```

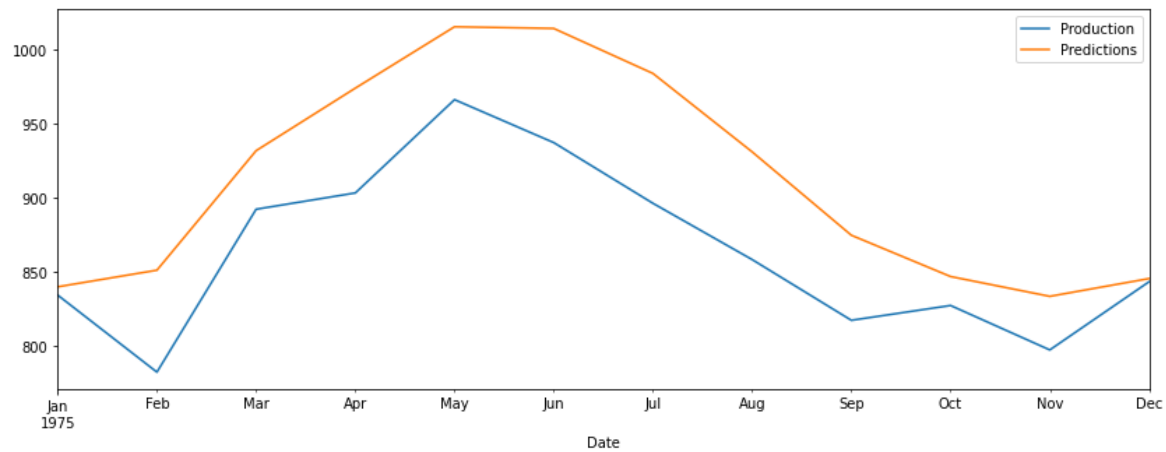
```
test.head()
```

Date	Production	Predictions
1975-01-01	834	839.607218
1975-02-01	782	850.841801
1975-03-01	892	931.551035
1975-04-01	903	973.778545
1975-05-01	966	1015.227947

We see the value of the prediction column is transformed back to the original test value scales.

```
test.plot(figsize=(14,5))
```

```
<AxesSubplot:xlabel='Date'>
```



The above figure shows how the prediction compares to the predictions and we see that they are similar, and the model has done a good job.

If we want to put a number to how good a prediction is, we can simply calculate the root mean square error:

```
[ ]  from sklearn.metrics import mean_squared_error
     from math import sqrt
     rmse = sqrt(mean_squared_error(test['Production'], te
     print(rmse)
```

```
56.03878051682608
```

So, the rmse for LSTM RNN is about 56. Let's perform GRU on our dataset and compare them with each other.

The steps I did to make GRU are as follows:

1. Creating a Simple GRU Neural Network with Keras
    1. Importing the Right Modules
    2. Adding Layers to Your Model
2. Training and Testing our Gated Recurrent Unit RNN on the MNIST Dataset
    1. Load the MNIST dataset
    2. Compile the GRU model
    3. Train and Fit the Model
    4. Test your GRU Model

Using Keras and Tensorflow makes building neural networks much easier to build. It's much easier to build neural networks with these libraries than from scratch.

Creating GRU RNN Model in Keras:

```
# Creating GRU RNN Model in Keras
model_GRU = Sequential() #define model sequential to have layer by lay
model_GRU.add(GRU(100, input_shape=(n_input, n_features)))
model_GRU.add(Dense(1))
model_GRU.compile(optimizer='adam', loss='mse') #compile the model
```

```
model_GRU.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 gru_4 (GRU)                 (None, 100)               30900

 dense_5 (Dense)             (None, 1)                 101

=================================================================
Total params: 31,001
Trainable params: 31,001
Non-trainable params: 0
_________________________________________________________________
```

Fit the model using on training data for GRU RNN using model_GRU.fit() and see the loss values per epochs:

```
# fit model
model_GRU.fit(generator, epochs=30)
```

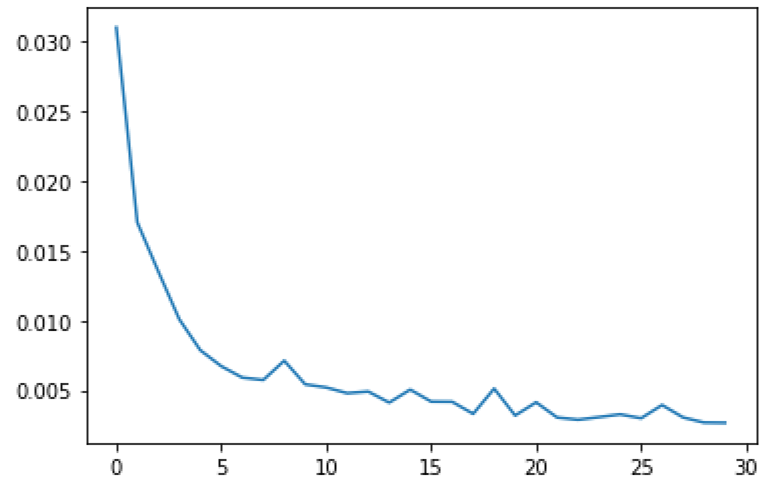
```
144/144 [==============================] - 2s 13ms/step - loss: 0.0170
Epoch 3/30
144/144 [==============================] - 2s 12ms/step - loss: 0.0135
Epoch 4/30
144/144 [==============================] - 2s 14ms/step - loss: 0.0101
Epoch 5/30
144/144 [==============================] - 2s 15ms/step - loss: 0.0079
Epoch 6/30
144/144 [==============================] - 2s 15ms/step - loss: 0.0067
Epoch 7/30
144/144 [==============================] - 2s 15ms/step - loss: 0.0059
Epoch 8/30
144/144 [==============================] - 2s 13ms/step - loss: 0.0057
Epoch 9/30
144/144 [==============================] - 2s 16ms/step - loss: 0.0071
Epoch 10/30
144/144 [==============================] - 2s 15ms/step - loss: 0.0054
Epoch 11/30
144/144 [==============================] - 2s 15ms/step - loss: 0.0052
Epoch 12/30
144/144 [==============================] - 2s 13ms/step - loss: 0.0048
Epoch 13/30
144/144 [==============================] - 2s 14ms/step - loss: 0.0049
Epoch 14/30
144/144 [==============================] - 2s 13ms/step - loss: 0.0041
Epoch 15/30
```

As we see loss decreases in each epoch. To see its trend better, I will plot loss per epoch.

We can use the data collected in the history object to create plots. I will get the last 12 values in the training set as testing dataset.

```
loss_per_epoch = model_GRU.history.history['loss'] #Display D
plt.plot(range(len(loss_per_epoch)), loss_per_epoch) #We can
```

```
[<matplotlib.lines.Line2D at 0x7f241baf93d0>]
```



We can use the data collected in the history object to create plots. I will get the last 12 values in the training set as testing dataset.

```
test_predictions = []

first_eval_batch = scaled_train[-n_input:] # get the last 12 values in the training set
current_batch = first_eval_batch.reshape((1, n_input, n_features)) # the current input or the current batch of 12 values

for i in range(len(test)):
  #print(i)
  # get the prediction value for the first batch
  current_pred = model_GRU.predict(current_batch)[0] # use the last 12 values in order to make the prediction. 0 means towards row

  # 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) #taking the current batch while dropping its first inp
  # current_batch[:,1:,:] means dropping the first input. axis=1 means appending it along the column
```

tranfrom test values back to the original scale becuase the original test values:

```
[ ] true_predictions = scaler.inverse_transform(test_predictions) #tranfrom test values back to the origina
```

```
[ ] test['Predictions'] = true_predictions #add Predictions as an np array to test dataframe
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.
  """Entry point for launching an IPython kernel.
```

We will see the prediction values are transformed back to the original testing scales:

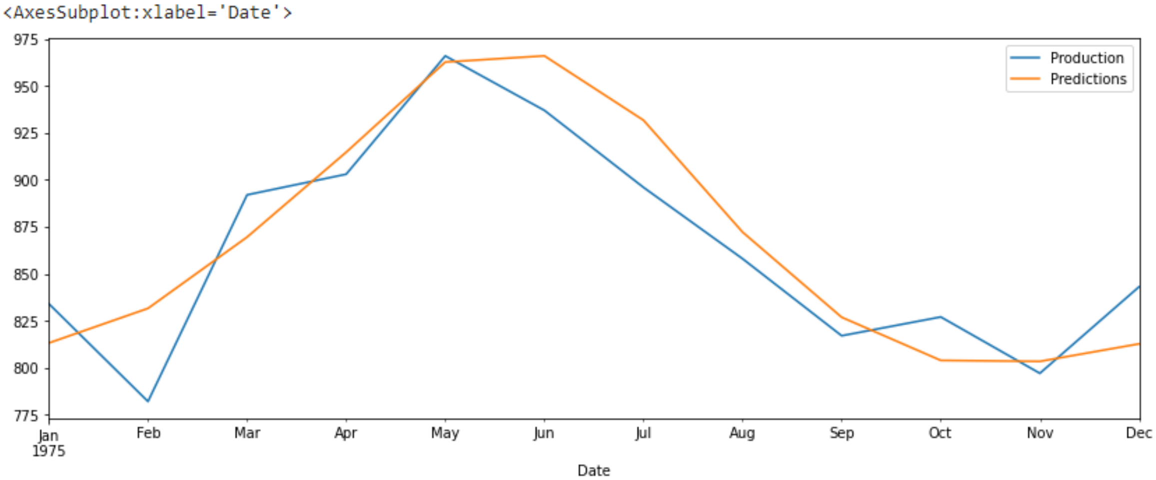
```
test.head()
```

Date	Production	Predictions
1975-01-01	834	813.139921
1975-02-01	782	831.571308
1975-03-01	892	869.535772
1975-04-01	903	914.719852
1975-05-01	966	962.723606

If we plot prediction values along with actual values, we can see:

```
test.plot(figsize=(14,5))
```

```
<AxesSubplot:xlabel='Date'>
```



As we see, these two values, prediction and actual values are more similar to each other when using GRU. To show it better, we can calculate the rmse:

Conclusion:

GRU trains faster than LSTM. A GRU is basically an LSTM without an output gate. They perform similarly to LSTMs for most tasks but do better on smaller datasets and less frequent data as we saw in this codes. We also learned that a GRU is just a fancy RNN with gates. We built a simple sequential GRU with one layer.

```python
from sklearn.metrics import mean_squared_error
from math import sqrt
rmse = sqrt(mean_squared_error(test['Production'], test['Predictions']))
print(rmse)
```

```
24.93011198711782
```

The rmse value for LSTM was about 56 but for GRU it is about 24.

Conclusion:

GRU trains faster than LSTM. A GRU is basically an LSTM without an output gate. They perform similarly to LSTMs for most tasks but do better on smaller datasets and less frequent data as we saw in these codes. We also learned that a GRU is just a fancy RNN with gates. We built a simple sequential GRU with one layer.