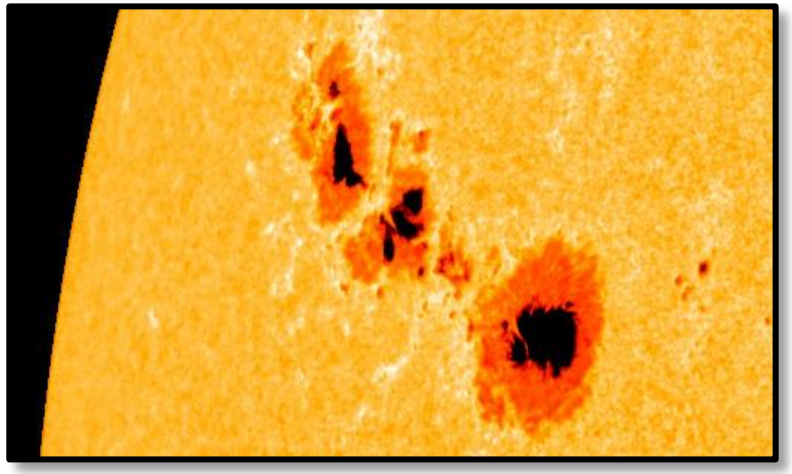
RNN Examples

Problem description

In this tutorial, we will introduce a problem and solve it using Recurrent Neural Networks (RNNs). In the previous presentation, we studied different types of sequential problems, including sequence-to-number, sequence classification, and sequence-to-sequence generation.

The problem we will address in this presentation is of the sequence-to-number type. We will predict the average number of sunspots per day for the upcoming month. To tackle this problem, we have a dataset containing the average number of sunspots per day for each month from 1749 to 1983.

Before proceeding to solve this problem, it is essential to understand what a sunspot is and its significance in solar activity.

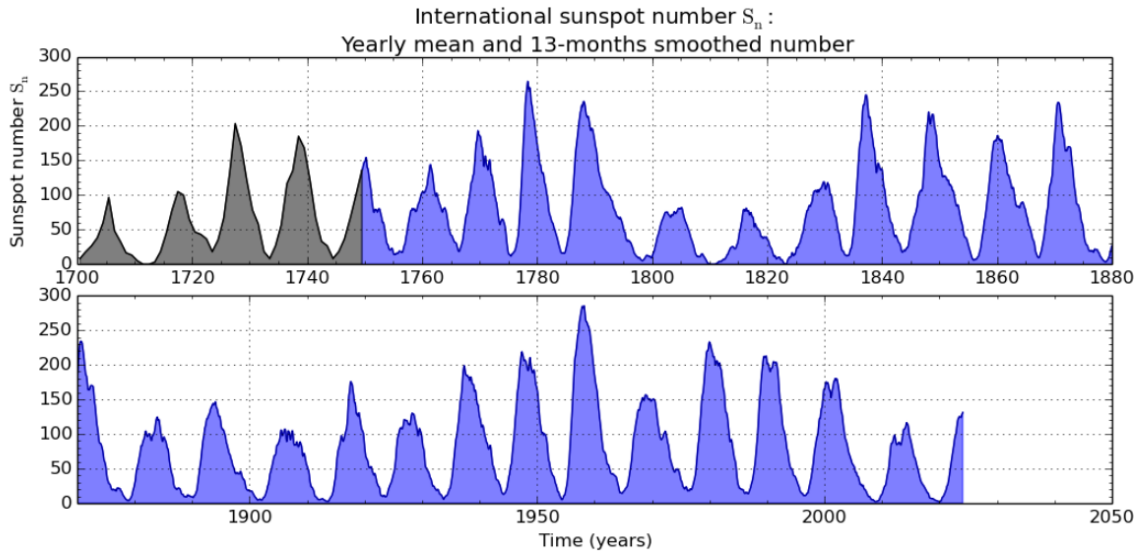


Source: www.wikipedia.org

RNN Examples

Problem description

Sunspots are temporary, dark regions on the Sun's surface caused by intense magnetic activity that inhibits convection, leading to lower temperatures compared to the surrounding areas.
The daily average number of sunspots each month is considered sequential data because it reflects a time series that captures the cyclical nature of solar activity. This data exhibits patterns over time, making it suitable for prediction using Recurrent Neural Networks (RNNs), which are designed to work with sequences and can learn from past observations to predict future values effectively



Source: www.sidc.be

Import the modules

Like any deep learning project, we will first include the essential modules.
We will use Pandas to read the dataset, which is provided in CSV format. NumPy will be utilized to reshape our inputs and perform some preprocessing tasks. We will define our model using the Keras Sequential API, incorporating Dense layers and SimpleRNN layers.
Additionally, we will use MinMaxScaler to scale the input data. To evaluate our model, we will plot the predicted outputs against the actual expected values, so we will also import Matplotlib.

```
1 from pandas import read_csv
2 import numpy as np
3 from keras.models import Sequential
4 from keras.layers import Dense, SimpleRNN
5 from sklearn.preprocessing import MinMaxScaler
6 from sklearn.metrics import mean_squared_error
7 import math
8 import matplotlib.pyplot as plt
```

Source: www.machinelearningmastery.com

Preparing the dataset

In this example, we will define a function that encompasses each step, from importing and preprocessing the data to evaluating the model.
Within this function, we will prepare our dataset. First, we will import it using the read_csv function from Pandas. Next, we will convert the input data type to float. We will then scale the data so that all numbers are between 0 and 1.
We will use the first 80 percent of the data for training, while the remaining 20 percent will be used for testing. This percentage can be modified when we call the function by adjusting the split_percent value.

```
10  # Parameter split_percent defines the ratio of training examples
11  def get_train_test(url, split_percent=0.8):
12      df = read_csv(url, usecols=[1], engine='python')
13      data = np.array(df.values.astype('float32'))
14      scaler = MinMaxScaler(feature_range=(0, 1))
15      data = scaler.fit_transform(data).flatten()
16      n = len(data)
17      # Point for splitting data into train and test
18      split = int(n*split_percent)
19      train_data = data[range(split)]
20      test_data = data[split:]
21      return train_data, test_data, data
22
```

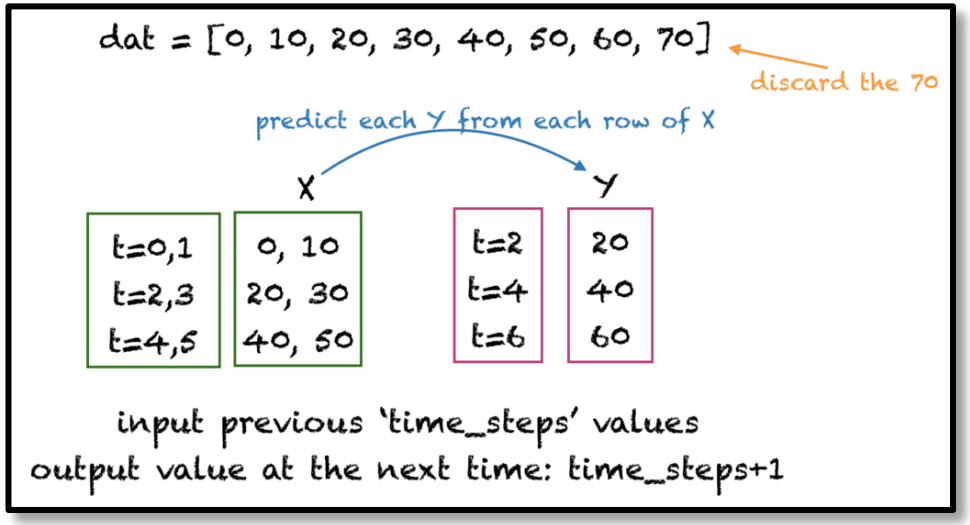
Source: www.machinelearningmastery.com

Preparing the dataset

Within the function, we identify specific indices in the dataset to extract target values. These indices are spaced according to the number of time steps, ensuring that each target corresponds to the end of a sequence of input data.

The input sequences are then constructed based on the identified target values. We calculate how many sequences can be created. Then, we reshape the data into a three-dimensional array. This format is essential for RNNs, as they require inputs to be structured as sequences of observations.

Finally, the function returns two arrays: one containing the input sequences and another containing the corresponding target values.




```python
# Prepare the input X and target Y
def get_XY(data, time_steps):
    Y_ind = np.arange(time_steps, len(data), time_steps)
    Y = data[Y_ind]
    rows_x = len(Y)
    X = data[range(time_steps*rows_x)]
    X = np.reshape(X, (rows_x, time_steps, 1))
    return X, Y

```

Source: www.machinelearningmastery.com

Define the Network

The most challenging aspect of solving a problem using RNNs is preparing the data. Defining the model is relatively straightforward, especially when using the Keras Sequential API. We follow the same steps as we would for creating an artificial neural network (ANN); however, when adding an RNN layer, we use the SimpleRNN class in Keras. We specify the input size and the activation function.

In this example, we will use one Dense layer on top of the RNN layer with a single neuron. This layer is responsible for generating our output.

Finally, we compile the model using the Mean Squared Error (MSE) loss function and the Adam optimizer.

```
32  def create_RNN(hidden_units, dense_units, input_shape, activation):
33      model = Sequential()
34      model.add(SimpleRNN(hidden_units, input_shape=input_shape, activation=activation[0]))
35      model.add(Dense(units=dense_units, activation=activation[1]))
36      model.compile(loss='mean_squared_error', optimizer='adam')
37      return model
38
```

Source: www.machinelearningmastery.com

Define the metrics

To assess the effectiveness of our trained model, we need to calculate various metrics that quantify its performance on the regression task. We will evaluate our model in two ways. First, we will print some metrics like MSE and RMSE. We will also use a visual method, which we will cover in the next slide.
To calculate the MSE of the model on the training and testing datasets, we will use the mean squared error function from the Keras metrics. To convert the calculated MSE to RMSE, we will use the sqrt function from the math module. In the end, we will print the results of our calculations.

```
39  def print_error(trainY, testY, train_predict, test_predict):
40      # Error of predictions
41      train_rmse = math.sqrt(mean_squared_error(trainY, train_predict))
42      test_rmse = math.sqrt(mean_squared_error(testY, test_predict))
43      # Print RMSE
44      print('Train RMSE: %.3f RMSE' % (train_rmse))
45      print('Test RMSE: %.3f RMSE' % (test_rmse))
46
```

Source: www.machinelearningmastery.com

Visual comparison

Another commonly used method to evaluate a regression model is to plot the model's predictions alongside the actual values on the same graph. This allows us to visually assess how well the model has performed on both the testing and training datasets.

First, we create a NumPy array containing both the actual values and the model predictions. Next, we define a figure and plot both arrays on it. To differentiate between the testing and training values, we add a vertical line at the boundary of the two sets.

Finally, we will add a legend and labels to make the plot easier to understand.

```
47  # Plot the result
48  def plot_result(trainY, testY, train_predict, test_predict):
49      actual = np.append(trainY, testY)
50      predictions = np.append(train_predict, test_predict)
51      rows = len(actual)
52      plt.figure(figsize=(15, 6), dpi=80)
53      plt.plot(range(rows), actual)
54      plt.plot(range(rows), predictions)
55      plt.axvline(x=len(trainY), color='r')
56      plt.legend(['Actual', 'Predictions'])
57      plt.xlabel('Observation number after given time steps')
58      plt.ylabel('Sunspots scaled')
59      plt.title('Actual and Predicted Values. The Red Line Separates The Training And Test Examples')
```

Source: www.machinelearningmastery.com

Final part

Now that we have defined the essential functions, it is time to put everything together. We will first define some variables, such as the URL to the dataset and the number of time steps.
Next, we will prepare the testing and training datasets using the functions we created earlier. After that, we will define the model, incorporating the RNN architecture and Dense layers.
We will then train the model using the training dataset. Finally, we will ask the model to predict the training and testing datasets, print the evaluation metrics, and plot the graph we defined earlier to compare the actual values with the predictions.

```
61  sunspots_url = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/monthly-sunspots.csv'
62  time_steps = 12
63  train_data, test_data, data = get_train_test(sunspots_url)
64  trainX, trainY = get_XY(train_data, time_steps)
65  testX, testY = get_XY(test_data, time_steps)
66
67  # Create model and train
68  model = create_RNN(hidden_units=3, dense_units=1, input_shape=(time_steps,1),
69                     activation=['tanh', 'tanh'])
70  model.fit(trainX, trainY, epochs=20, batch_size=1, verbose=2)
71
72  # make predictions
73  train_predict = model.predict(trainX)
74  test_predict = model.predict(testX)
75
76  # Print error
77  print_error(trainY, testY, train_predict, test_predict)
78
79  #Plot result
80  plot_result(trainY, testY, train_predict, test_predict)
```

Source: www.machinelearningmastery.com

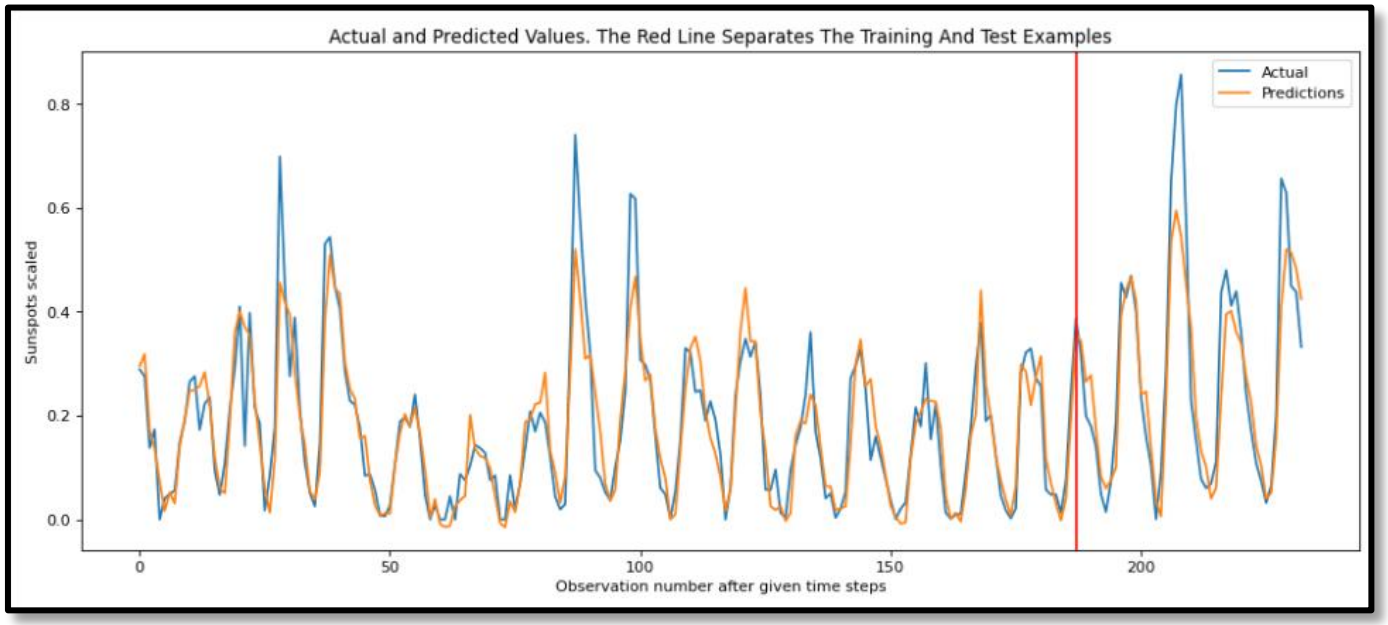
Outputs

When we run the code, we will see the final graph along with the RMSE for both the testing and training datasets.

Let's start with the RMSE. We can observe that both the testing and training RMSE values are quite low, indicating that the model has effectively learned from the training dataset. Additionally, we can be confident that no overfitting has occurred, as the testing RMSE is also low.

We can draw similar conclusions from the graph. The orange and blue lines, which represent the actual and predicted values, are very close to each other at every point on the graph. This proximity holds true on both sides of the vertical line, indicating that the model has learned effectively and has not overfitted the training data.

```
Train RMSE: 0.058 RMSE
Test RMSE: 0.077 RMSE
```



Source: www.machinelearningmastery.com

Summary

In this lecture, we focused on solving a regression problem using Recurrent Neural Networks (RNNs). We chose this structure because we were dealing with sequential data.
The main difference between building a simple artificial neural network (ANN) model and an RNN model lies in the data preparation process. We needed to reshape our data to fit the RNN structure. Another key difference we covered was the RNN cell itself. We used the Keras Sequential API to add these cells to our model.
Finally, we evaluated our model using the Root Mean Squared Error (RMSE) and a comparison graph. In the next lecture, we will explore the limitations of RNNs and their solutions.