radiation

May 26, 2024

1 Overview

For this project, we'll use a dataset that tracks the level of solar radiation. The dataset is openly available from the OpenML Repository: https://www.openml.org/search?type=data&status=active&id=43751.

The objective of this project is to create a model that can forecast the next 24 hours of solar radiation.

[55]:
```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns

from tensorflow.keras import Model, Sequential

from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.losses import MeanSquaredError
from tensorflow.keras.metrics import MeanAbsoluteError

from tensorflow.keras.layers import Dense, Conv1D, LSTM, Lambda, Reshape, RNN,␣
↪LSTMCell

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

Set a random seed to ensure that the results can be reproduced.

[56]:
```
tf.random.set_seed(42)
np.random.seed(42)
```

2 Data Exploring and Preprocessing

Fetch data from OpenML.

[57]:
```python
from sklearn.datasets import fetch_openml

solar = fetch_openml(data_id=43751)
df = solar.frame
df
```

[57]:
```
         UNIXTime                    Data      Time  Radiation  Temperature  \
0      1475229326  9/29/2016 12:00:00 AM  23:55:26       1.21           48
1      1475229023  9/29/2016 12:00:00 AM  23:50:23       1.21           48
2      1475228726  9/29/2016 12:00:00 AM  23:45:26       1.23           48
3      1475228421  9/29/2016 12:00:00 AM  23:40:21       1.21           48
4      1475228124  9/29/2016 12:00:00 AM  23:35:24       1.17           48
…             …                      …        …        …          …
32681  1480587604  12/1/2016 12:00:00 AM  00:20:04       1.22           44
32682  1480587301  12/1/2016 12:00:00 AM  00:15:01       1.17           44
32683  1480587001  12/1/2016 12:00:00 AM  00:10:01       1.20           44
32684  1480586702  12/1/2016 12:00:00 AM  00:05:02       1.23           44
32685  1480586402  12/1/2016 12:00:00 AM  00:00:02       1.20           44

       Pressure  Humidity  WindDirection(Degrees)  Speed TimeSunRise  \
0         30.46        59                  177.39   5.62    06:13:00
1         30.46        58                  176.78   3.37    06:13:00
2         30.46        57                  158.75   3.37    06:13:00
3         30.46        60                  137.71   3.37    06:13:00
4         30.46        62                  104.95   5.62    06:13:00
…           …       …                      …      …         …
32681     30.43       102                  145.42   6.75    06:41:00
32682     30.42       102                  117.78   6.75    06:41:00
32683     30.42       102                  145.19   9.00    06:41:00
32684     30.42       101                  164.19   7.87    06:41:00
32685     30.43       101                   83.59   3.37    06:41:00

      TimeSunSet
0       18:13:00
1       18:13:00
2       18:13:00
3       18:13:00
4       18:13:00
…          …
32681   17:42:00
32682   17:42:00
32683   17:42:00
32684   17:42:00
32685   17:42:00

[32686 rows x 11 columns]
```

```
[ ]: pred_column = 'Radiation'
```

2.1 Makeing datetime feature

```
[59]: # make a datetime column
df['date'] = pd.to_datetime(df['UNIXTime'] - 10*60*60, unit='s', origin='unix')
df
```

```
[59]:          UNIXTime                    Data      Time  Radiation  Temperature  \
0      1475229326   9/29/2016 12:00:00 AM  23:55:26       1.21           48
1      1475229023   9/29/2016 12:00:00 AM  23:50:23       1.21           48
2      1475228726   9/29/2016 12:00:00 AM  23:45:26       1.23           48
3      1475228421   9/29/2016 12:00:00 AM  23:40:21       1.21           48
4      1475228124   9/29/2016 12:00:00 AM  23:35:24       1.17           48
…             …                       …       …         …           …
32681  1480587604  12/1/2016 12:00:00 AM  00:20:04       1.22           44
32682  1480587301  12/1/2016 12:00:00 AM  00:15:01       1.17           44
32683  1480587001  12/1/2016 12:00:00 AM  00:10:01       1.20           44
32684  1480586702  12/1/2016 12:00:00 AM  00:05:02       1.23           44
32685  1480586402  12/1/2016 12:00:00 AM  00:00:02       1.20           44

       Pressure  Humidity  WindDirection(Degrees)  Speed TimeSunRise  \
0         30.46        59                  177.39   5.62    06:13:00
1         30.46        58                  176.78   3.37    06:13:00
2         30.46        57                  158.75   3.37    06:13:00
3         30.46        60                  137.71   3.37    06:13:00
4         30.46        62                  104.95   5.62    06:13:00
…           …       …                      …    …         …
32681     30.43       102                  145.42   6.75    06:41:00
32682     30.42       102                  117.78   6.75    06:41:00
32683     30.42       102                  145.19   9.00    06:41:00
32684     30.42       101                  164.19   7.87    06:41:00
32685     30.43       101                   83.59   3.37    06:41:00

      TimeSunSet                date
0       18:13:00 2016-09-29 23:55:26
1       18:13:00 2016-09-29 23:50:23
2       18:13:00 2016-09-29 23:45:26
3       18:13:00 2016-09-29 23:40:21
4       18:13:00 2016-09-29 23:35:24
…            …                 …
32681   17:42:00 2016-12-01 00:20:04
32682   17:42:00 2016-12-01 00:15:01
32683   17:42:00 2016-12-01 00:10:01
32684   17:42:00 2016-12-01 00:05:02
32685   17:42:00 2016-12-01 00:00:02
```

```
[32686 rows x 12 columns]
```

Sort the dataset by datetime and reset the index.

```
[60]: # sort by date and re-index the dataset
      df = df.sort_values(by=['date'], ascending=True).reset_index(drop=True)
```

```
[61]: # drop columns don't need them anymore
      df = df.drop(columns=['UNIXTime', 'Data', 'Time'])
      df
```

```
[61]:        Radiation  Temperature  Pressure  Humidity  WindDirection(Degrees)  \
      0           2.58           51     30.43       103                   77.27
      1           2.83           51     30.43       103                  153.44
      2           2.16           51     30.43       103                  142.04
      3           2.21           51     30.43       103                  144.12
      4           2.25           51     30.43       103                   67.42
      …              …            …         …         …                       …
      32681       1.22           41     30.34        83                  238.94
      32682       1.21           41     30.34        82                  236.79
      32683       1.21           42     30.34        81                  218.28
      32684       1.19           41     30.34        80                  215.23
      32685       1.21           41     30.34        81                  215.56

             Speed TimeSunRise TimeSunSet                date
      0      11.25    06:07:00   18:38:00 2016-09-01 00:00:08
      1       9.00    06:07:00   18:38:00 2016-09-01 00:05:10
      2       7.87    06:07:00   18:38:00 2016-09-01 00:20:06
      3      18.00    06:07:00   18:38:00 2016-09-01 00:25:05
      4      11.25    06:07:00   18:38:00 2016-09-01 00:30:09
      …        …         …          …                  …
      32681   6.75    06:57:00   17:54:00 2016-12-31 23:35:02
      32682   5.62    06:57:00   17:54:00 2016-12-31 23:40:01
      32683   7.87    06:57:00   17:54:00 2016-12-31 23:45:04
      32684   7.87    06:57:00   17:54:00 2016-12-31 23:50:03
      32685   9.00    06:57:00   17:54:00 2016-12-31 23:55:01

      [32686 rows x 9 columns]
```

2.2 Handling non-numeric data

```
[62]: # convert "TimeSunRise" and "TimeSunSet" to integer: second of the day
      from datetime import datetime

      pt = [datetime.strptime(x,'%H:%M:%S') for x in df['TimeSunRise']]
      df['TimeSunRise'] = [x.second + x.minute*60 + x.hour*360 for x in pt]
      df
```

[62]:
```
       Radiation  Temperature  Pressure  Humidity  WindDirection(Degrees)  \
0           2.58           51     30.43       103                   77.27
1           2.83           51     30.43       103                  153.44
2           2.16           51     30.43       103                  142.04
3           2.21           51     30.43       103                  144.12
4           2.25           51     30.43       103                   67.42
…            …            …      …       …                    …
32681       1.22           41     30.34        83                  238.94
32682       1.21           41     30.34        82                  236.79
32683       1.21           42     30.34        81                  218.28
32684       1.19           41     30.34        80                  215.23
32685       1.21           41     30.34        81                  215.56

       Speed  TimeSunRise TimeSunSet                 date
0      11.25         2580   18:38:00 2016-09-01 00:00:08
1       9.00         2580   18:38:00 2016-09-01 00:05:10
2       7.87         2580   18:38:00 2016-09-01 00:20:06
3      18.00         2580   18:38:00 2016-09-01 00:25:05
4      11.25         2580   18:38:00 2016-09-01 00:30:09
…        …          …        …                   …
32681   6.75         5580   17:54:00 2016-12-31 23:35:02
32682   5.62         5580   17:54:00 2016-12-31 23:40:01
32683   7.87         5580   17:54:00 2016-12-31 23:45:04
32684   7.87         5580   17:54:00 2016-12-31 23:50:03
32685   9.00         5580   17:54:00 2016-12-31 23:55:01

[32686 rows x 9 columns]
```

[63]:
```python
pt = [datetime.strptime(x, '%H:%M:%S') for x in df['TimeSunSet']]
df['TimeSunSet'] = [x.second + x.minute * 60 + x.hour * 360 for x in pt]
df
```

[63]:
```
       Radiation  Temperature  Pressure  Humidity  WindDirection(Degrees)  \
0           2.58           51     30.43       103                   77.27
1           2.83           51     30.43       103                  153.44
2           2.16           51     30.43       103                  142.04
3           2.21           51     30.43       103                  144.12
4           2.25           51     30.43       103                   67.42
…            …            …      …       …                    …
32681       1.22           41     30.34        83                  238.94
32682       1.21           41     30.34        82                  236.79
32683       1.21           42     30.34        81                  218.28
32684       1.19           41     30.34        80                  215.23
32685       1.21           41     30.34        81                  215.56

       Speed  TimeSunRise  TimeSunSet                 date
0      11.25         2580        8760 2016-09-01 00:00:08
```

```
1        9.00          2580         8760 2016-09-01 00:05:10
2        7.87          2580         8760 2016-09-01 00:20:06
3       18.00          2580         8760 2016-09-01 00:25:05
4       11.25          2580         8760 2016-09-01 00:30:09
…       …           …          …                    …
32681    6.75          5580         9360 2016-12-31 23:35:02
32682    5.62          5580         9360 2016-12-31 23:40:01
32683    7.87          5580         9360 2016-12-31 23:45:04
32684    7.87          5580         9360 2016-12-31 23:50:03
32685    9.00          5580         9360 2016-12-31 23:55:01

[32686 rows x 9 columns]
```

2.3 Resampling

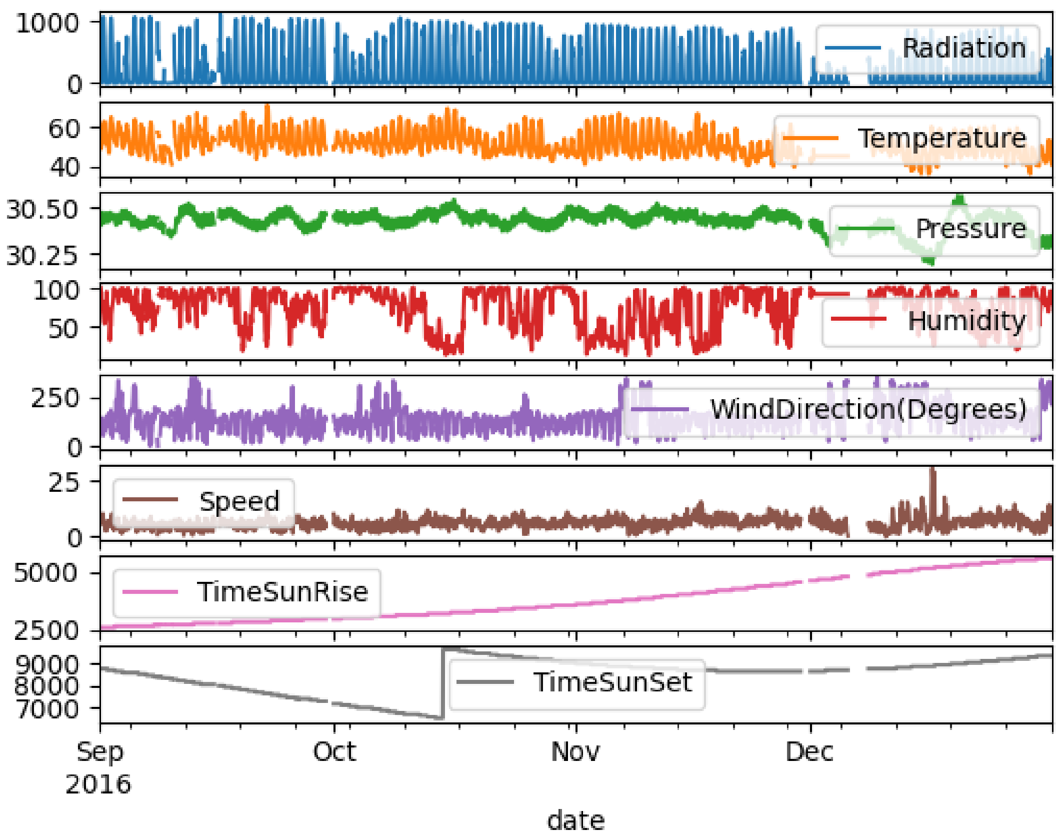
[64]:
```
# resample the dataset to 1 hour interval
df = df.resample('1h', on='date').mean()
df
```

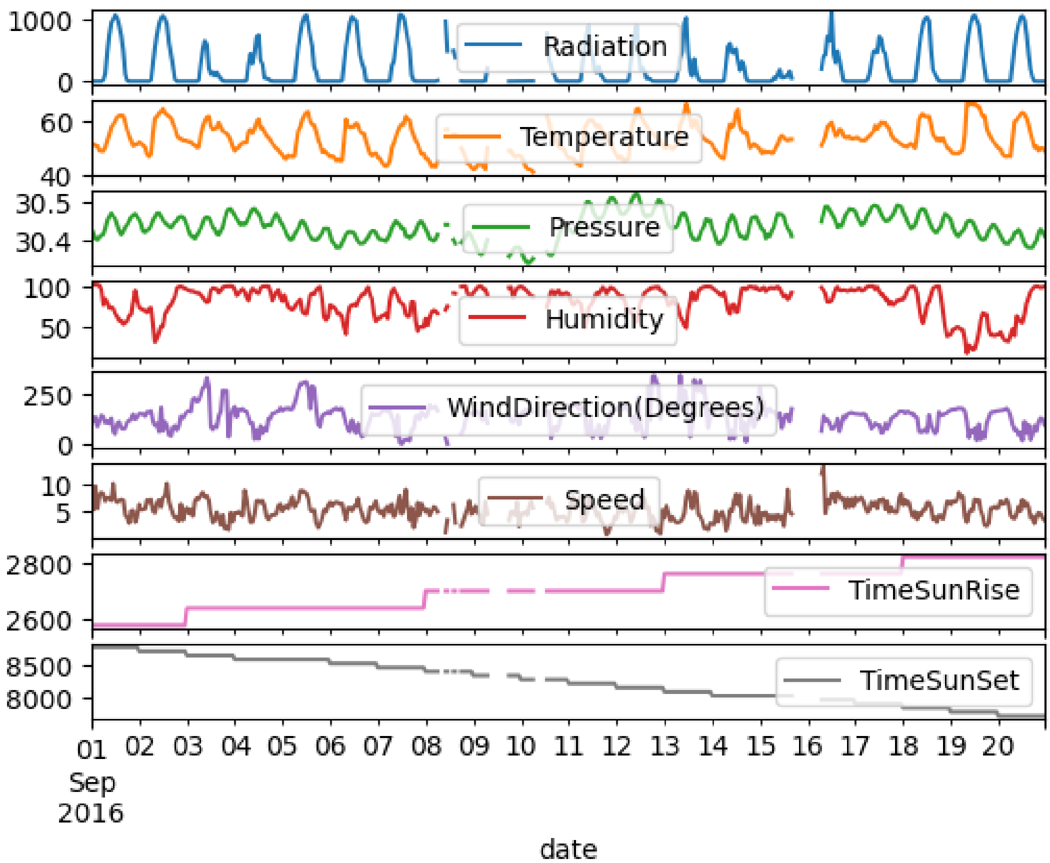
[64]:
```
                     Radiation  Temperature   Pressure    Humidity  \
date
2016-09-01 00:00:00   2.288750    51.125000  30.430000  103.000000
2016-09-01 01:00:00   2.943333    51.500000  30.417500  103.000000
2016-09-01 02:00:00   2.733333    51.000000  30.404167  103.000000
2016-09-01 03:00:00   2.344545    50.818182  30.400000  102.636364
2016-09-01 04:00:00   2.607500    49.083333  30.407500  102.000000
…                          …          …          …        …
2016-12-31 19:00:00   1.221667    46.166667  30.327500   93.666667
2016-12-31 20:00:00   1.216667    44.166667  30.337500   87.083333
2016-12-31 21:00:00   1.225833    41.833333  30.343333   83.333333
2016-12-31 22:00:00   1.207500    40.833333  30.345000   80.166667
2016-12-31 23:00:00   1.204167    40.666667  30.340000   82.000000

                     WindDirection(Degrees)     Speed  TimeSunRise  TimeSunSet
date
2016-09-01 00:00:00              109.837500  8.857500       2580.0      8760.0
2016-09-01 01:00:00              121.345833  5.246667       2580.0      8760.0
2016-09-01 02:00:00              136.402500  9.653333       2580.0      8760.0
2016-09-01 03:00:00               89.257273  5.520909       2580.0      8760.0
2016-09-01 04:00:00              118.165833  7.965833       2580.0      8760.0
…                                     …        …           …          …
2016-12-31 19:00:00              285.428333  5.997500       5580.0      9360.0
2016-12-31 20:00:00              231.082500  5.435000       5580.0      9360.0
2016-12-31 21:00:00              213.929167  7.404167       5580.0      9360.0
2016-12-31 22:00:00              213.836667  8.341667       5580.0      9360.0
2016-12-31 23:00:00              228.371667  6.746667       5580.0      9360.0
```

```
[2928 rows x 8 columns]
```

[65]:
```python
# Define a function for ploting the dataset
def plot_data(df, plot_cols = ['Radiation', 'Temperature', 'Pressure', 
 ↪'Humidity', 'WindDirection(Degrees)', 'Speed', 'TimeSunRise', 'TimeSunSet']):
    plot_features = df[plot_cols]
    plot_features.index = df.index
    _ = plot_features.plot(subplots=True)

    plot_features = df[plot_cols][:480]
    plot_features.index = df.index[:480]
    _ = plot_features.plot(subplots=True)
```

[66]:
```python
plot_data(df)
```





2.4 Dealing with missing data

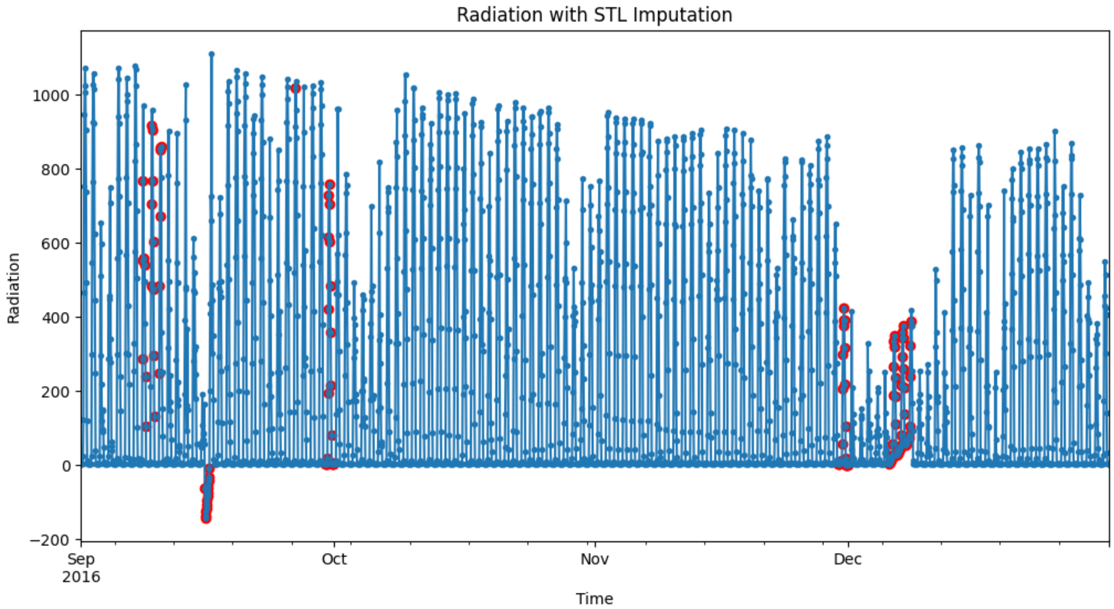
[67]:
```
df.isna().sum()
```

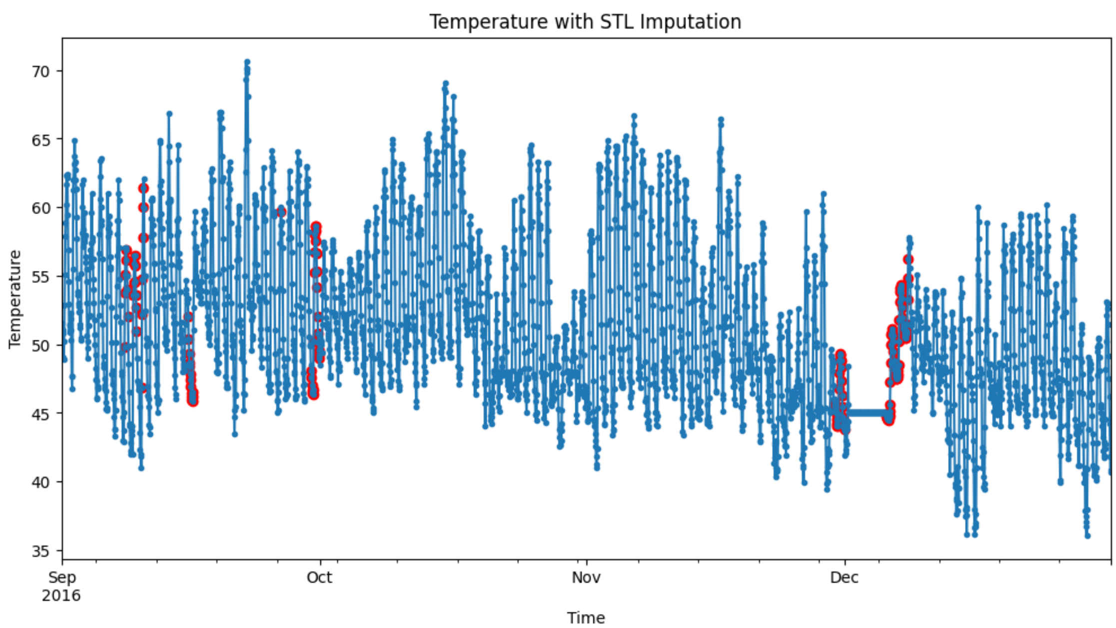
[67]:
```
Radiation                 151
Temperature               151
Pressure                  151
Humidity                  151
WindDirection(Degrees)    151
Speed                     151
TimeSunRise               151
TimeSunSet                151
dtype: int64
```

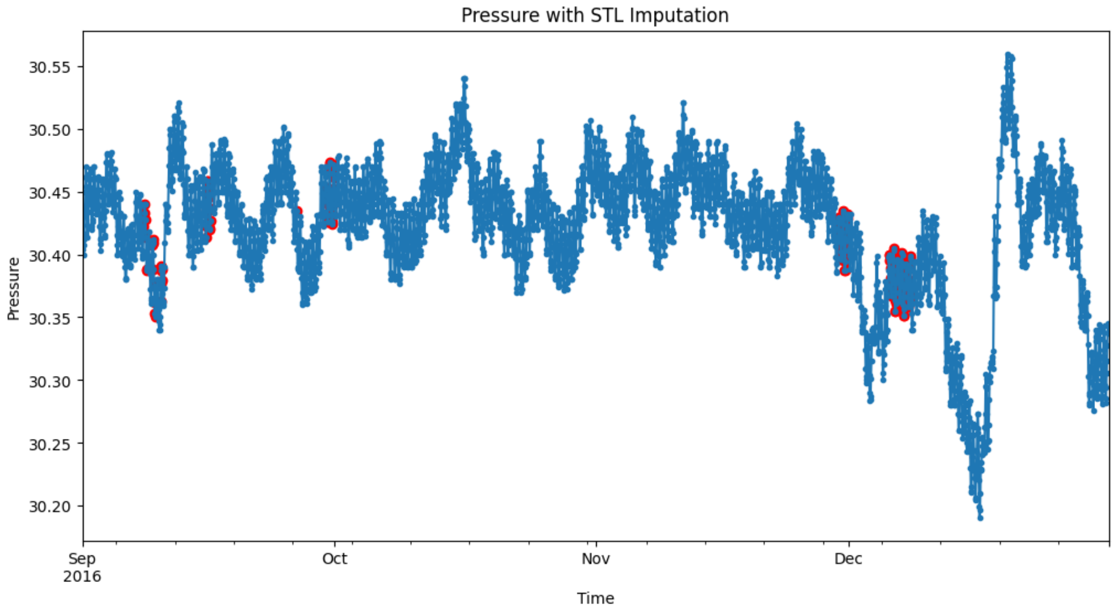
STL Decomposition for Time Series: This method breaks down the time series into trend, seasonality, and residuals, then imputes missing values in the residuals before reassembling the components. This could prove useful for time series data with a distinct trend and seasonality.

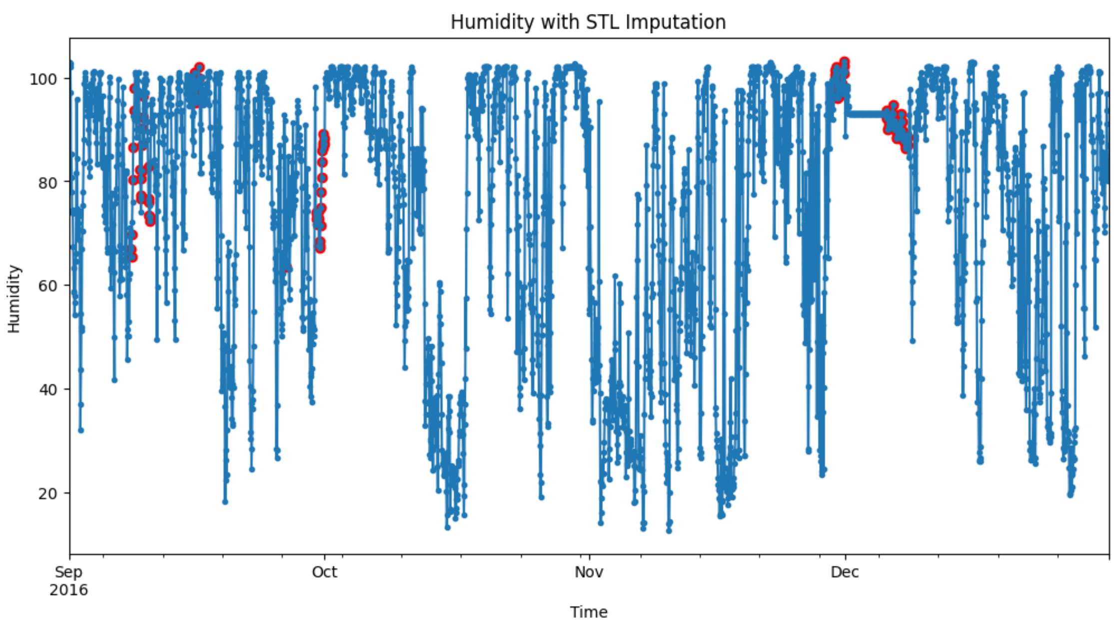
[68]:
```
# function for imputing seasonal data
from statsmodels.tsa.seasonal import STL

def impute_missing_seasonal(df, columns):
    df_copy = df.copy()
    for c in columns:
        # Fill missing values in the time series
        imputed_indices = df[df[c].isnull()].index
        # Apply STL decompostion
        stl = STL(df_copy[c].interpolate(), seasonal=31)
        res = stl.fit()

        # Extract the seasonal and trend components
        seasonal_component = res.seasonal

        # Create the deseasonalised series
        df_deseasonalised = df_copy[c] - seasonal_component

        # Interpolate missing values in the deseasonalised series
        df_deseasonalised_imputed = df_deseasonalised.
↪interpolate(method="linear")

        # Add the seasonal component back to create the final imputed series
        df_imputed = df_deseasonalised_imputed + seasonal_component

        # Update the original dataframe with the imputed values
        df_copy.loc[imputed_indices, c] = df_imputed[imputed_indices]

        # Plot the series using pandas
        plt.figure(figsize=[12, 6])
        df_copy[c].plot(style='.-',  label=c)
        plt.scatter(imputed_indices, df_copy.loc[imputed_indices, c],␣
↪color='red')

        plt.title("{0} with STL Imputation".format(c))
        plt.ylabel(c)
        plt.xlabel("Time")
        plt.show()
    return df_copy
```

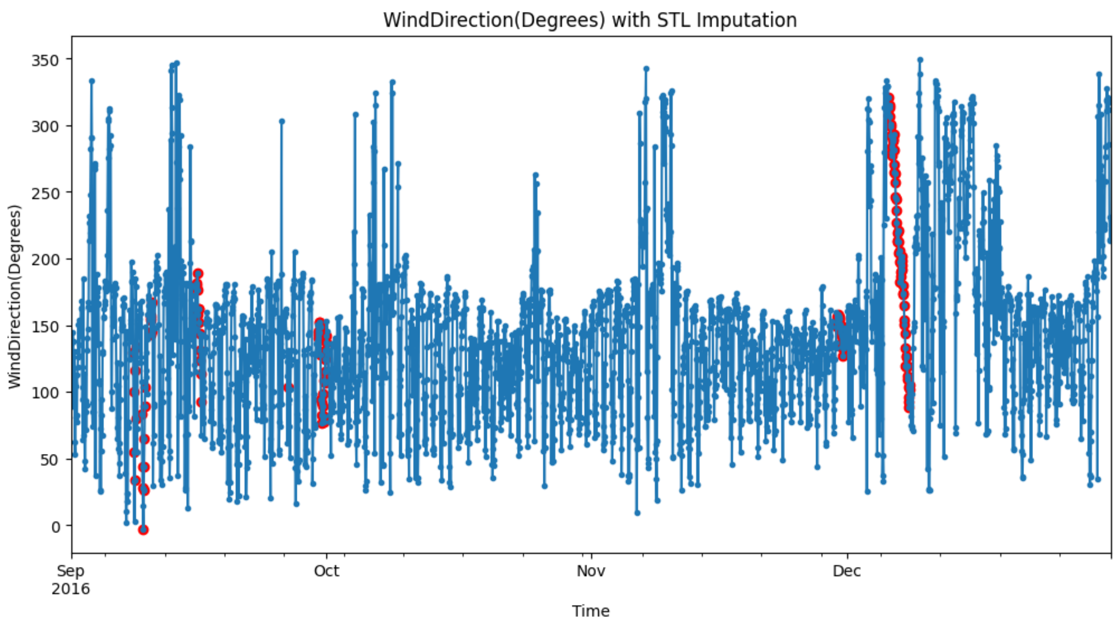
[69]:
```
df = impute_missing_seasonal(df, columns=['Radiation', 'Temperature',␣
↪'Pressure', 'Humidity', 'WindDirection(Degrees)', 'Speed'])
```

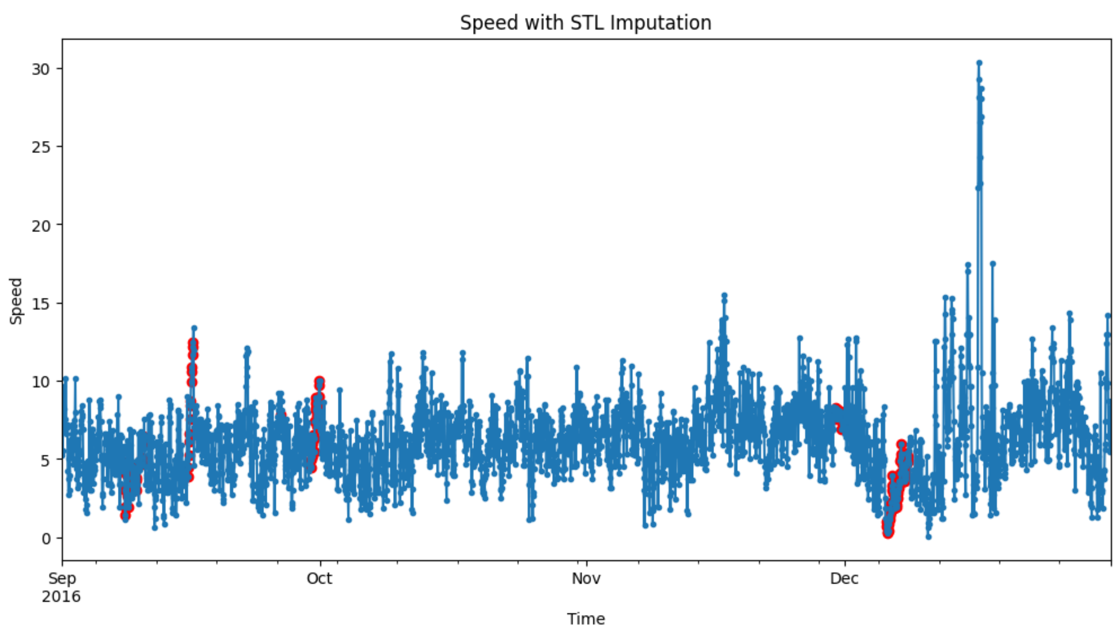

Radiation with STL Imputation
Radiation
1000
800
600
400
200
0
−200
Sep
2016
Oct
Nov
Dec
Time


Temperature with STL Imputation
Temperature
70
65
60
55
50
45
40
35
Sep
2016
Oct
Nov
Dec
Time

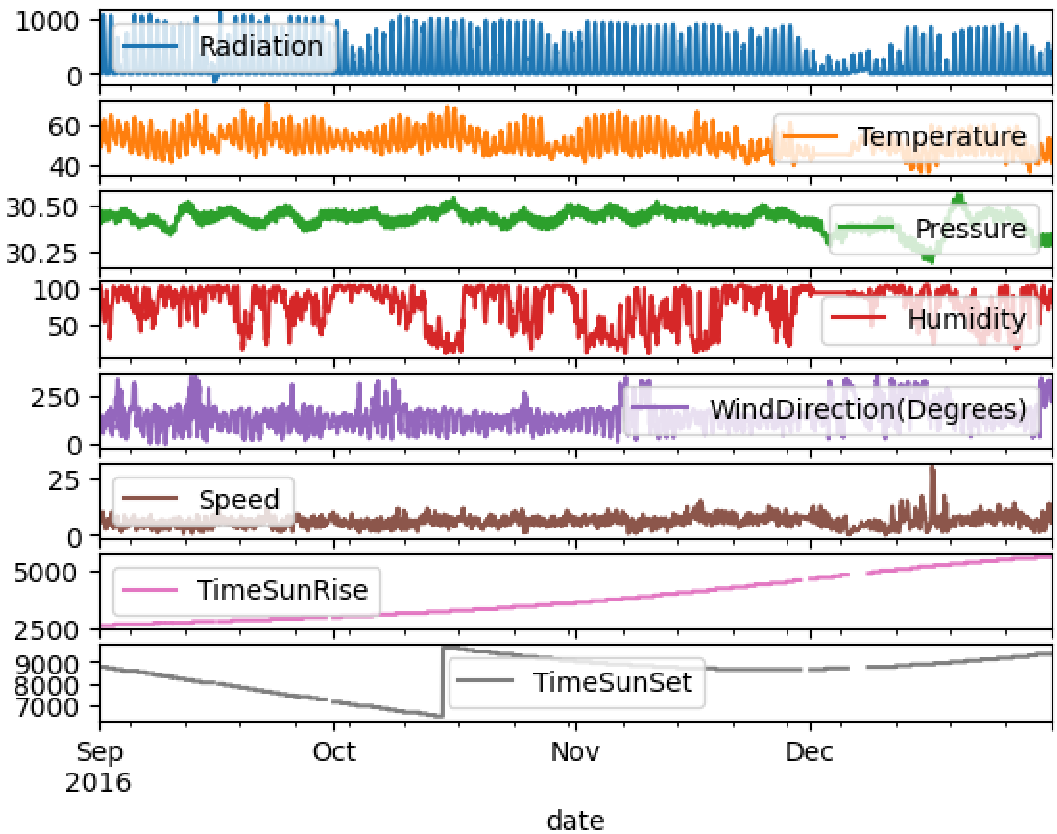

Pressure with STL Imputation
30.55
30.50
30.45
30.40
30.35
30.30
30.25
30.20
Pressure
Sep
2016
Oct
Nov
Dec
Time

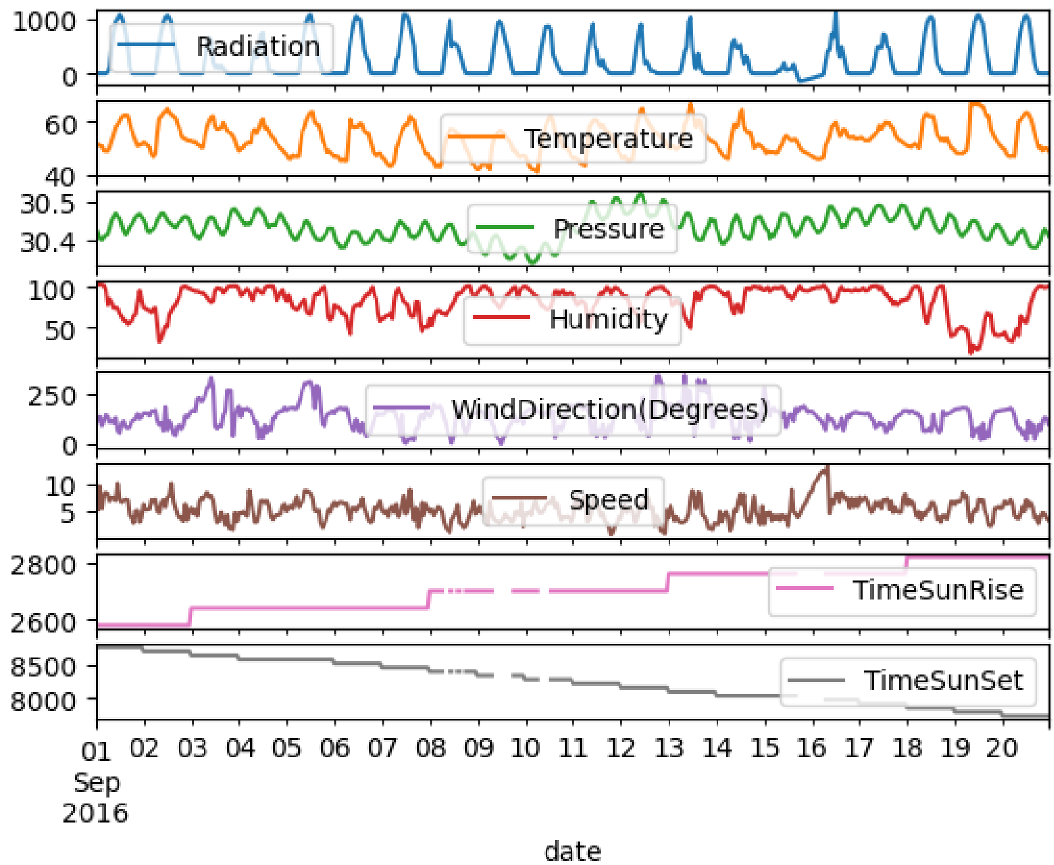

Humidity with STL Imputation
100
80
60
40
20
Humidity
Sep
2016
Oct
Nov
Dec
Time





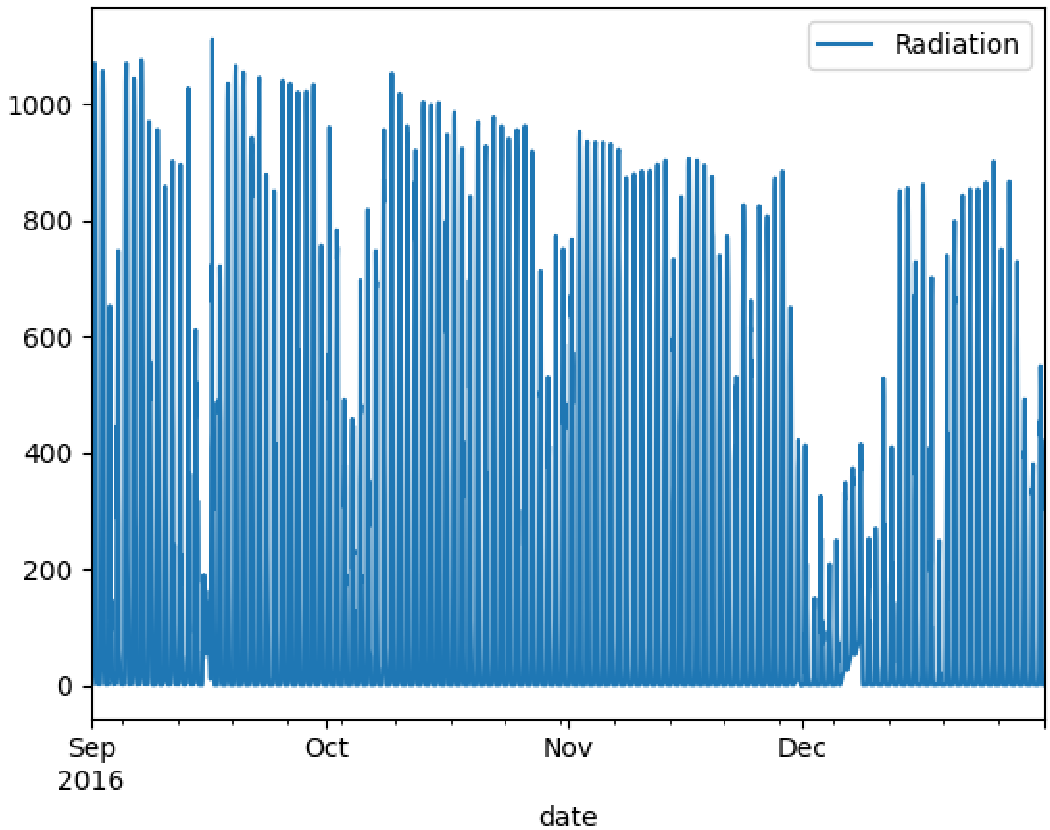
[70]:
```
plot_data(df)
```


Radiation
Temperature
Pressure
Humidity
WindDirection(Degrees)
Speed
TimeSunRise
TimeSunSet
1000
0
60
40
30.50
30.25
100
50
250
0
25
0
5000
2500
9000
8000
7000
Sep
2016
Oct
Nov
Dec
date

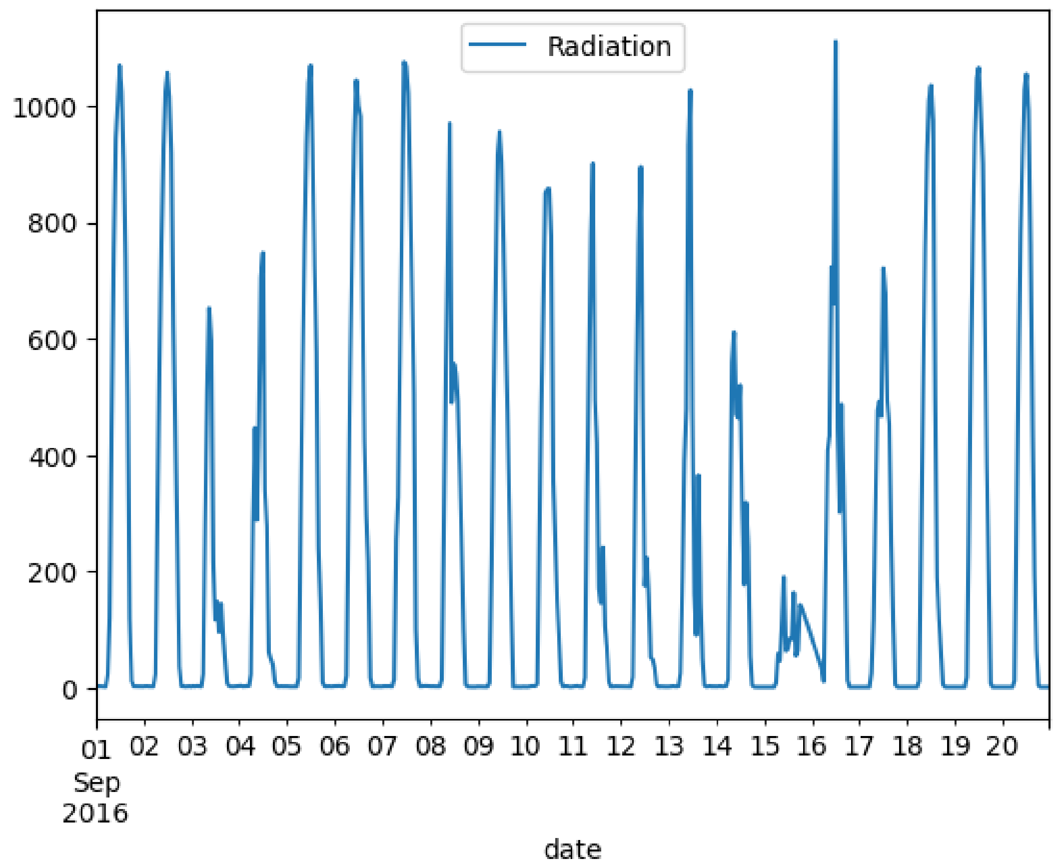


All columns look good except "Radiation". Because Radiation must be >=0, so we take the absolute values of Radiation to make sure they all >= 0.

[71]:
```
# convert all "Radiation" to positive values
df['Radiation'] = df['Radiation'].abs()
plot_data(df, plot_cols=['Radiation'])
```



Radiation
1000
800
600
400
200
0
Sep
2016
Oct
Nov
Dec
date



[72]:
```
df.describe().transpose()
```

[72]:

	count	mean	std	min	\
Radiation	2928.0	207.766305	304.949617	0.130143	
Temperature	2928.0	51.066114	6.060896	36.083333	
Pressure	2928.0	30.421961	0.053459	30.190833	
Humidity	2928.0	75.834242	25.160059	12.666667	
WindDirection(Degrees)	2928.0	143.994081	61.972130	-2.778259	
Speed	2928.0	6.181699	2.622149	0.093333	
TimeSunRise	2777.0	3801.001080	933.493334	2580.000000	
TimeSunSet	2777.0	8477.954627	809.787299	6480.000000	

	25%	50%	75%	max
Radiation	1.228333	7.026975	368.770625	1111.011667
Temperature	46.500000	50.000000	55.000000	70.666667
Pressure	30.400000	30.430000	30.456667	30.560000
Humidity	57.979167	86.000000	96.666667	103.274082
WindDirection(Degrees)	103.003333	141.590417	170.417292	349.908333
Speed	4.591667	5.999583	7.497500	30.370833

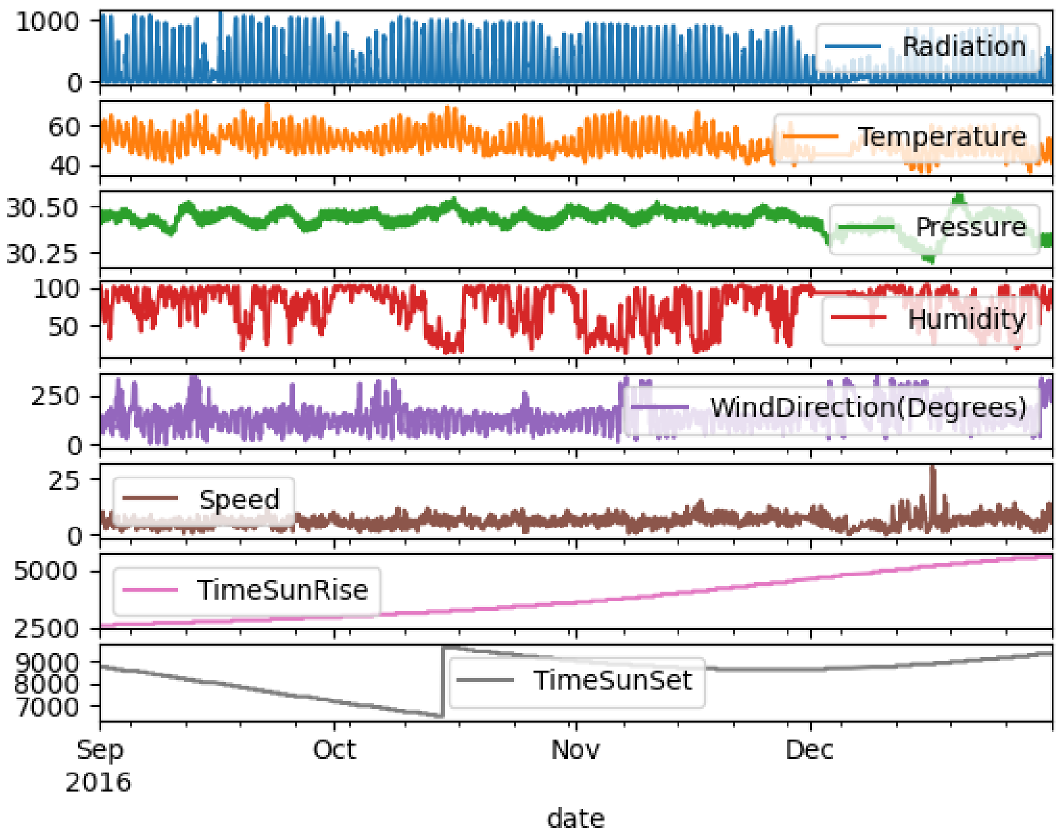
```
TimeSunRise             3000.000000  3540.000000  4560.000000  5580.000000
TimeSunSet              8040.000000  8700.000000  9000.000000  9660.000000
```

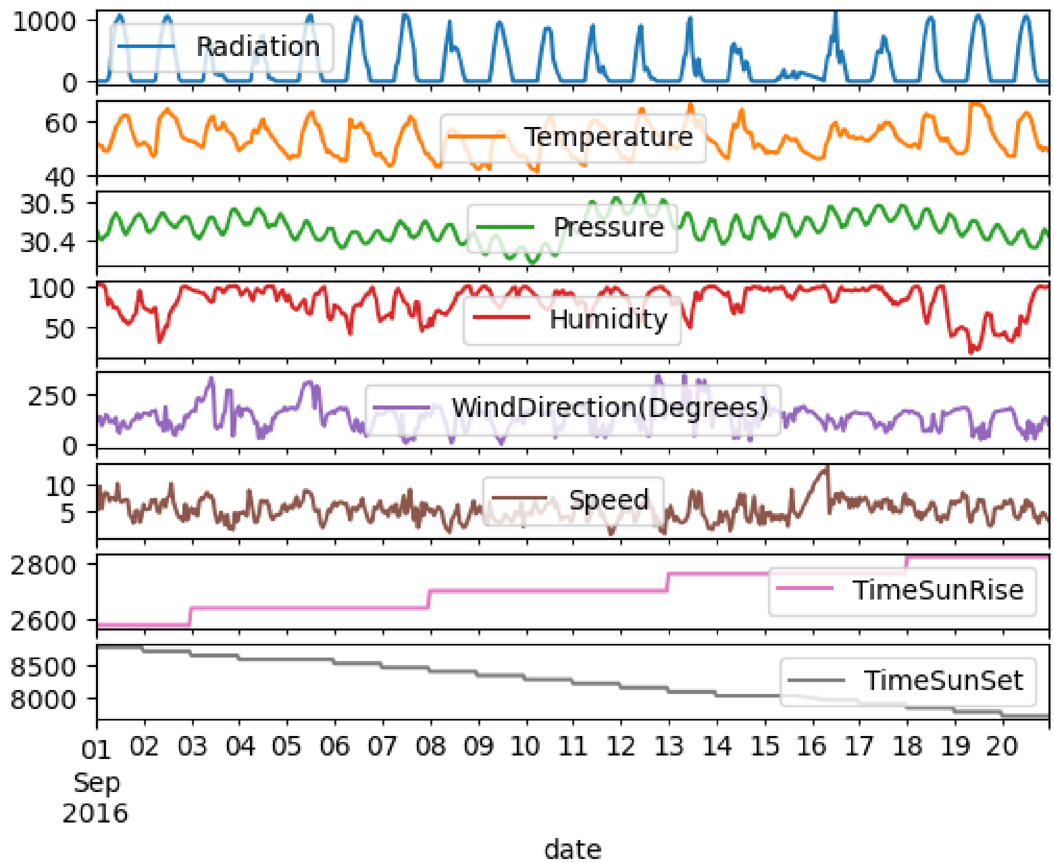
Fill NaN of “TimeSunRise” and “TimeSunSet” by linear method.

[73]:
```python
# Fill missing values of "TimeSunRise" and "TimeSunSet" by linear method
df = df.interpolate()
df.describe().transpose()
```

[73]:
```
                         count         mean         std          min  \
Radiation               2928.0   207.766305  304.949617     0.130143
Temperature             2928.0    51.066114    6.060896    36.083333
Pressure                2928.0    30.421961    0.053459    30.190833
Humidity                2928.0    75.834242   25.160059    12.666667
WindDirection(Degrees)  2928.0   143.994081   61.972130    -2.778259
Speed                   2928.0     6.181699    2.622149     0.093333
TimeSunRise             2928.0  3809.713115  935.646387  2580.000000
TimeSunSet              2928.0  8470.737705  799.056275  6480.000000

                                25%          50%          75%          max
Radiation                  1.228333     7.026975   368.770625  1111.011667
Temperature               46.500000    50.000000    55.000000    70.666667
Pressure                  30.400000    30.430000    30.456667    30.560000
Humidity                  57.979167    86.000000    96.666667   103.274082
WindDirection(Degrees)   103.003333   141.590417   170.417292   349.908333
Speed                      4.591667     5.999583     7.497500    30.370833
TimeSunRise             2940.000000  3570.000000  4620.000000  5580.000000
TimeSunSet              8040.000000  8700.000000  9000.000000  9660.000000
```

[74]:
```python
plot_data(df)
```


Radiation
Temperature
Pressure
Humidity
WindDirection(Degrees)
Speed
TimeSunRise
TimeSunSet
1000
0
60
40
30.50
30.25
100
50
250
0
25
0
5000
2500
9000
8000
7000
Sep
2016
Oct
Nov
Dec
date



2.5 Feature engineering

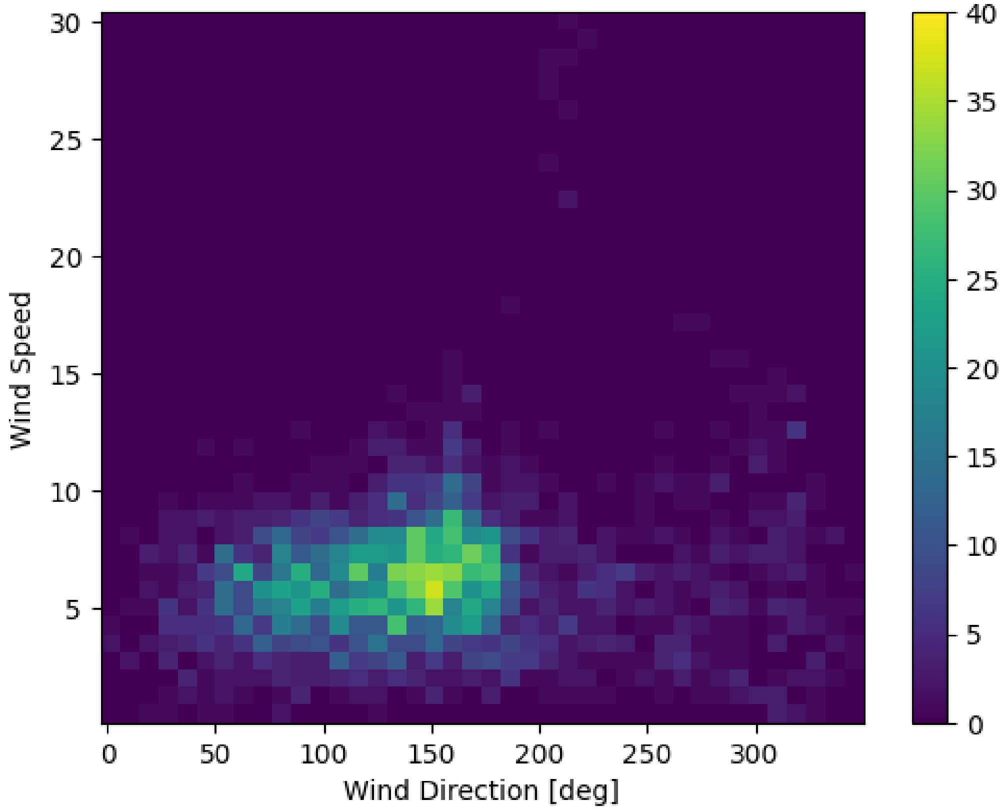
2.5.1 Wind

The feature "WindDirection(Degrees)" gives the wind direction in units of degrees. Angles do not make good model inputs: 360° and 0° should be close to each other and wrap around smoothly. Direction shouldn't matter if the wind is not blowing.

Right now the distribution of wind data looks like this:

[75]:
```
plt.hist2d(df['WindDirection(Degrees)'], df['Speed'], bins=(40, 40), vmax=40)
plt.colorbar()
plt.xlabel('Wind Direction [deg]')
plt.ylabel('Wind Speed')
```

[75]: `Text(0, 0.5, 'Wind Speed')`

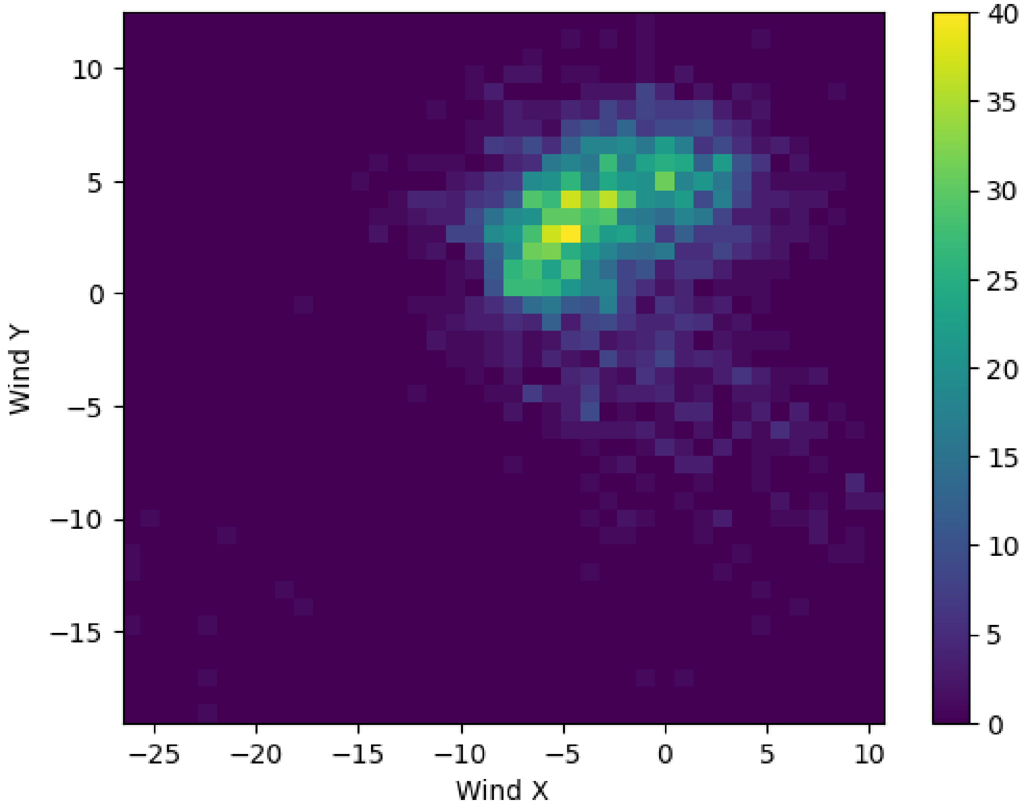


But this will be easier for the model to interpret if you convert the wind direction and velocity columns to a wind vector:

[76]:
```
# convert wind direction and speed to wind vector
speed = df.pop('Speed')

# Convert to radians.
wd_rad = df.pop('WindDirection(Degrees)')*np.pi / 180

# Calculate the wind x and y components.
df['Wx'] = speed*np.cos(wd_rad)
df['Wy'] = speed*np.sin(wd_rad)
```

[77]:
```
plt.hist2d(df['Wx'], df['Wy'], bins=(40, 40), vmax=40)
plt.colorbar()
plt.xlabel('Wind X')
plt.ylabel('Wind Y')
ax = plt.gca()
ax.axis('tight')
```

[77]: (-26.551356904849687,
 10.761976829797197,
 -19.059242496582993,
 12.472954133807654)

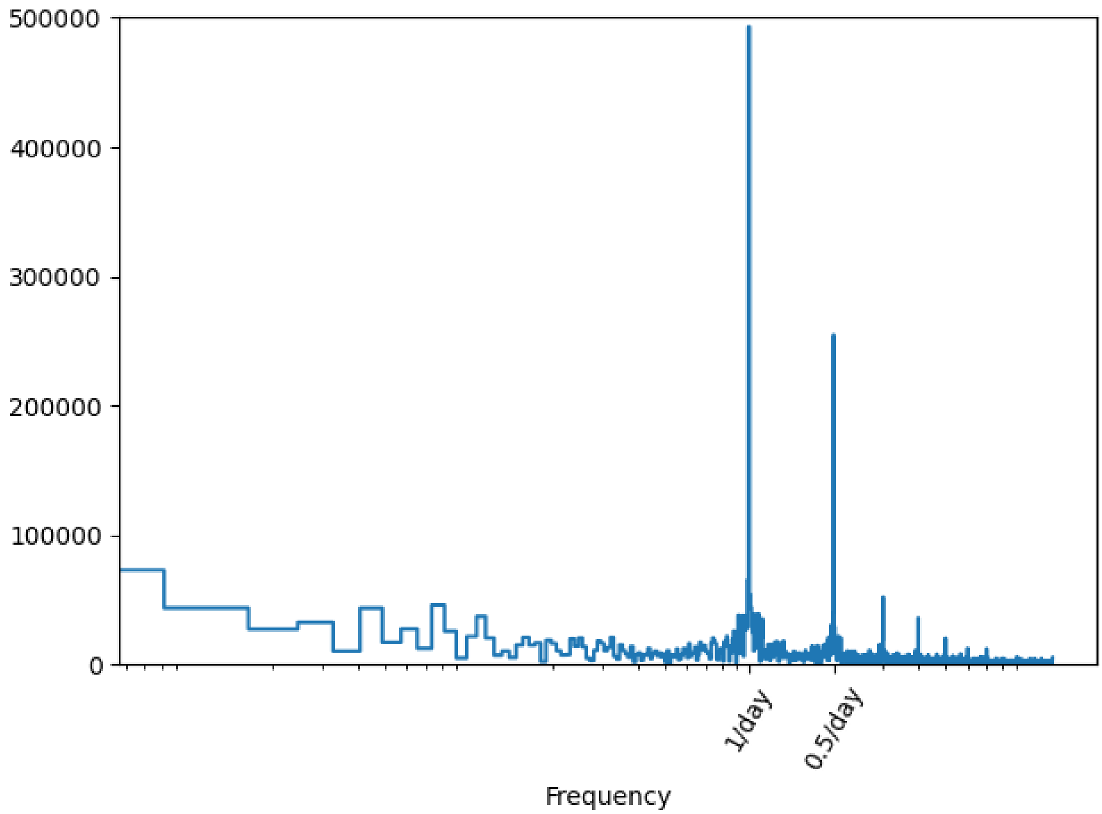


2.5.2 Time

With our target being solar radiation, it is likely that we'll have some seasonality. We can expect that at night, less solar radiation will be detected. Similarly, there may be a peak in the noon. Thus, it is reasonable to assume that there will be some seasonality in our target. We can plot our target to see if we can visually detect the period.

[78]:
```
# Set daily seasonality
fft = tf.signal.rfft(df[pred_column])
f_per_dataset = np.arange(0, len(fft))
n_sample_h = len(df[pred_column])
hours_per_day = 24
day_per_dataset = n_sample_h / hours_per_day
f_per_day = f_per_dataset / day_per_dataset
plt.step(f_per_day, np.abs(fft))
plt.xscale('log')
```

```
plt.ylim(0, 500000)
plt.xticks([1, 2], ['1/day', '0.5/day'], rotation=60)
plt.xlabel('Frequency')
plt.tight_layout()
plt.show()
```



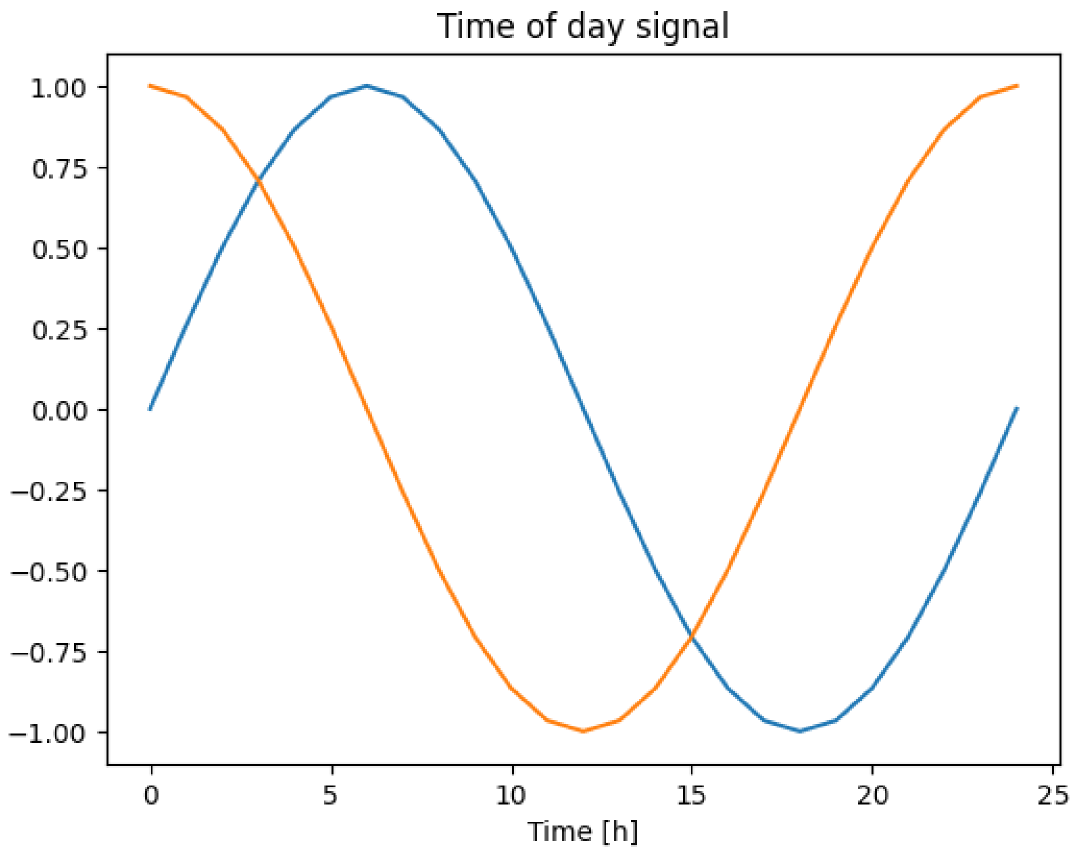
You can see that there is a visible peak for the daily seasonality. This tells us that we indeed have daily seasonality in our data. Thus, we will encode our timestamp using a sine and cosine transformation to express the time while keeping its daily seasonal information.

[79]:
```
# encode timestamp to sine and cosine
from datetime import datetime

timestamp_s = pd.to_datetime(df.index).map(datetime.timestamp)
day = 24 * 60 * 60
df['day_sin'] = (np.sin(timestamp_s * (2*np.pi/day))).values
df['day_cos'] = (np.cos(timestamp_s * (2*np.pi/day))).values
```

[80]:
```
plt.plot(np.array(df['day_sin'])[:25])
plt.plot(np.array(df['day_cos'])[:25])
plt.xlabel('Time [h]')
```

```
plt.title('Time of day signal')
```

[80]: Text(0.5, 1.0, 'Time of day signal')



2.6 Splitting and scaling the data

We'll split the data 70:20:10 for the training, validation, and test sets respectively.

[81]:
```
n = len(df)
# Split 70:20:10 (train:validation:test)
train_df = df[0:int(n*0.7)]
val_df = df[int(n*0.7):int(n*0.9)]
test_df = df[int(n*0.9):]
```

Next, we'll fit the scaler to the training set only, and scale each individual set.

[82]:
```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
scaler.fit(train_df)
```

```
train_df[train_df.columns] = scaler.transform(train_df[train_df.columns])
val_df[val_df.columns] = scaler.transform(val_df[val_df.columns])
test_df[test_df.columns] = scaler.transform(test_df[test_df.columns])
```

3 Preparing for modeling with deep learning

We will build **two baselines**, a **linear model**, a **deep neural network model**, a **long short-term memory (LSTM) model**, a **convolutional neural network (CNN)**, a **combination of CNN and LSTM**, and finally an **autoregressive LSTM**. In the end, we will use the *mean absolute error (MAE)* to determine which model is the best. The one that achieves the lowest MAE on the test set will be the top-performing model.

Note that we'll use the MAE as our evaluation metric and the mean squared error (MSE) as the loss function

3.0.1 Defining the DataWindow class

The DataWindow class allows us to quickly create windows of data for training deep learning models. Each window of data contains a set of inputs and a set of labels. The model is then trained to produce predictions as close as possible to the labels using the inputs.

[83]:
```
class DataWindow():
    def __init__(self, input_width, label_width, shift,
                 train_df=train_df, val_df=val_df, test_df=test_df,
                 label_columns=None):

        self.train_df = train_df
        self.val_df = val_df
        self.test_df = test_df

        self.label_columns = label_columns
        if label_columns is not None:
            self.label_columns_indices = {name: i for i, name in␣
↪enumerate(label_columns)}
        self.column_indices = {name: i for i, name in enumerate(train_df.
↪columns)}

        self.input_width = input_width
        self.label_width = label_width
        self.shift = shift

        self.total_window_size = input_width + shift

        self.input_slice = slice(0, input_width)
        self.input_indices = np.arange(self.total_window_size)[self.input_slice]

        self.label_start = self.total_window_size - self.label_width
```

```
        self.labels_slice = slice(self.label_start, None)
        self.label_indices = np.arange(self.total_window_size)[self.
↪labels_slice]

    def split_to_inputs_labels(self, features):
        inputs = features[:, self.input_slice, :]
        labels = features[:, self.labels_slice, :]
        if self.label_columns is not None:
            labels = tf.stack(
                [labels[:,:,self.column_indices[name]] for name in self.
↪label_columns],
                axis=-1
            )
        inputs.set_shape([None, self.input_width, None])
        labels.set_shape([None, self.label_width, None])

        return inputs, labels

    def plot(self, model=None, plot_col=pred_column, max_subplots=3):
        inputs, labels = self.sample_batch

        plt.figure(figsize=(12, 8))
        plot_col_index = self.column_indices[plot_col]
        max_n = min(max_subplots, len(inputs))

        for n in range(max_n):
            plt.subplot(3, 1, n+1)
            plt.ylabel(f'{plot_col} [scaled]')
            plt.plot(self.input_indices, inputs[n, :, plot_col_index],
                     label='Inputs', marker='.', zorder=-10)

            if self.label_columns:
                label_col_index = self.label_columns_indices.get(plot_col, None)
            else:
                label_col_index = plot_col_index

            if label_col_index is None:
                continue

            plt.scatter(self.label_indices, labels[n, :, label_col_index],
                        edgecolors='k', marker='s', label='Labels', c='green',␣
↪s=64)
            if model is not None:
                predictions = model(inputs)
                plt.scatter(self.label_indices, predictions[n, :,␣
↪label_col_index],
                            marker='X', edgecolors='k', label='Predictions',
```

```
                        c='red', s=64)

            if n == 0:
                plt.legend()

        plt.xlabel('Time (h)')

    def make_dataset(self, data):
        data = np.array(data, dtype=np.float32)
        ds = tf.keras.preprocessing.timeseries_dataset_from_array(
            data=data,
            targets=None,
            sequence_length=self.total_window_size,
            sequence_stride=1,
            shuffle=True,
            batch_size=32
        )

        ds = ds.map(self.split_to_inputs_labels)
        return ds

    @property
    def train(self):
        return self.make_dataset(self.train_df)

    @property
    def val(self):
        return self.make_dataset(self.val_df)

    @property
    def test(self):
        return self.make_dataset(self.test_df)

    @property
    def sample_batch(self):
        result = getattr(self, '_sample_batch', None)
        if result is None:
            result = next(iter(self.train))
            self._sample_batch = result
        return result
```

3.0.2 Utility function to train our models

[84]:
```
# function that automates the training process
def compile_and_fit(model, window, patience=3, max_epochs=50):
    early_stopping = EarlyStopping(monitor='val_loss',
                                   patience=patience,
```

```
                                            mode='min')

    model.compile(loss=MeanSquaredError(),
                  optimizer=Adam(),
                  metrics=[MeanAbsoluteError()])

    history = model.fit(window.train,
                        epochs=max_epochs,
                        validation_data=window.val,
                        callbacks=[early_stopping])

    return history
```

[85]:
```
# dictionary to store the column names and their corresponding indexes.
column_indices = {name: i for i, name in enumerate(train_df.columns)}
```

4 Modeling with deep learning

4.1 Baseline models

Every forecasting project must start with a baseline model. Baselines serve as a benchmark for our more sophisticated models, as they can only be better in comparison to a certain benchmark. Building baseline models also allows us to assess whether the added complexity of a model really generates a significant benefit. It is possible that a complex model does not perform much better than a baseline, in which case implementing a complex model is hard to justify. In this case, we'll build two baseline models: one that **repeats the last known value** and another that **repeats the last 24 hours of data**.

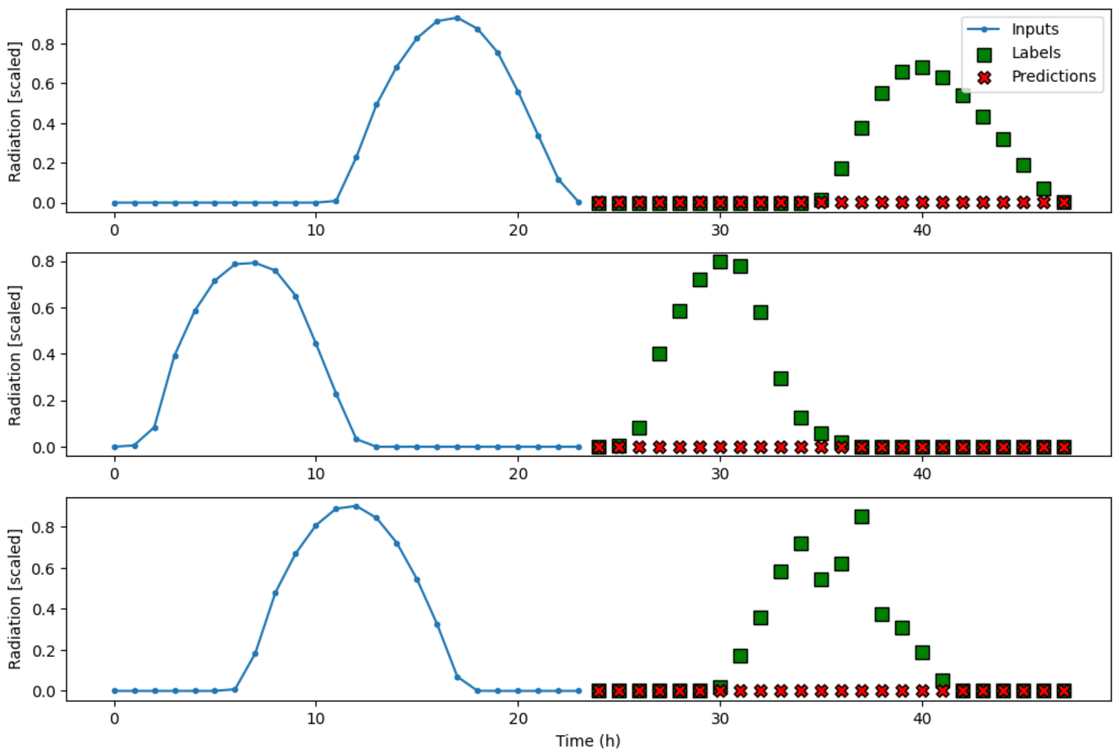
[86]:
```
# the window of data that will be used
# the length of our label sequence is 24 timesteps, and the shift will also be␣
 ↪24 timesteps.
# We'll also use an input length of 24
multi_window = DataWindow(input_width=24, label_width=24, shift=24,␣
 ↪label_columns=[pred_column])
```

[87]:
```
# repeat the last known value of the input sequence as a prediction for the␣
 ↪next 24 hours
class MultiStepLastBaseline(Model):
    def __init__(self, label_index=None):
        super().__init__()
        self.label_index = label_index

    def call(self, inputs):
        if self.label_index is None:
            return tf.tile(inputs[:, -1:, :], [1, 24, 1])
        return tf.tile(inputs[:, -1:, self.label_index:], [1, 24, 1])
```

[88]:
```
baseline_last = MultiStepLastBaseline(label_index=column_indices[pred_column])

baseline_last.compile(loss=MeanSquaredError(), metrics=[MeanAbsoluteError()])

val_performance = {}
performance = {}

val_performance['Baseline - Last'] = baseline_last.evaluate(multi_window.val)
performance['Baseline - Last'] = baseline_last.evaluate(multi_window.test,␣
  ↪verbose=0)
```

```
17/17              0s 2ms/step - loss:
0.3709 - mean_absolute_error: 0.4881
```

[89]:
```
multi_window.plot(baseline_last)
```



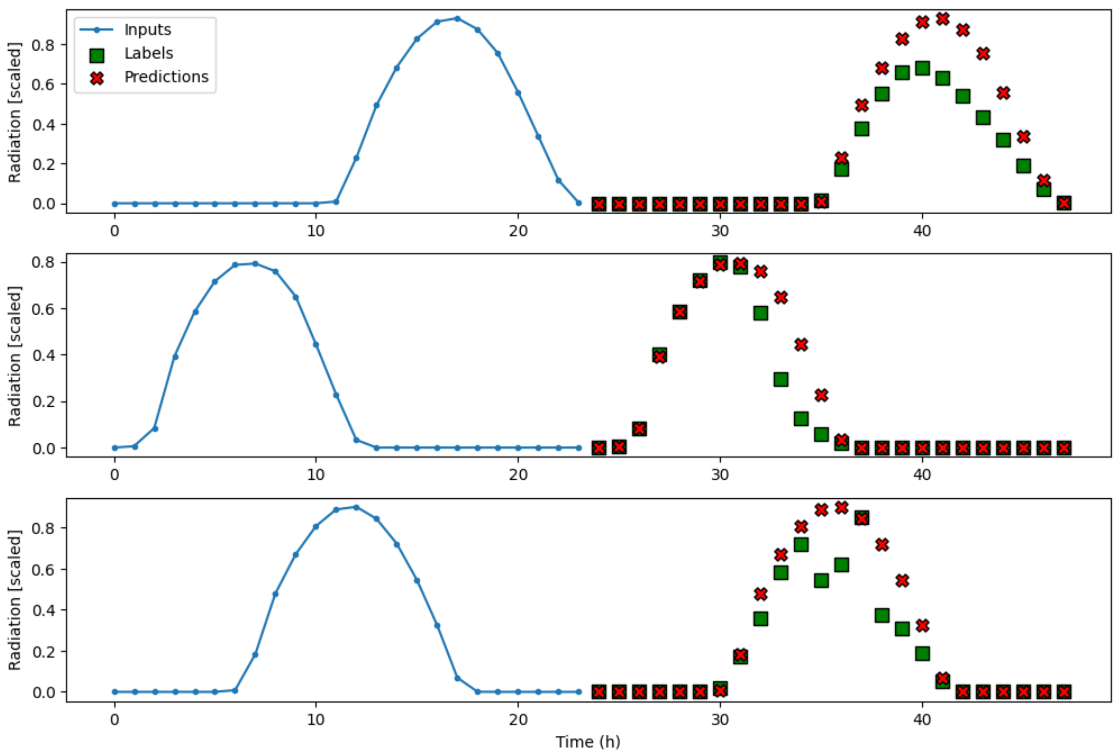
[90]:
```
# a baseline model that repeats the input sequence
class RepeatBaseline(Model):
    def __init__(self, label_index=None):
        super().__init__()
        self.label_index = label_index

    def call(self, inputs):
```

```
        return inputs[:, :, self.label_index:]
```

[91]:
```
baseline_repeat = RepeatBaseline(label_index=column_indices[pred_column])

baseline_repeat.compile(loss=MeanSquaredError(), metrics=[MeanAbsoluteError()])

val_performance['Baseline - Repeat'] = baseline_repeat.evaluate(multi_window.
 ↪val)
performance['Baseline - Repeat'] = baseline_repeat.evaluate(multi_window.test,␣
 ↪verbose=0)
```

```
17/17              0s 2ms/step - loss:
0.3650 - mean_absolute_error: 0.4789
```

[92]:
```
multi_window.plot(baseline_repeat)
```



You'll see that the predictions are equal to the input sequence, which is the expected behavior for this baseline model.

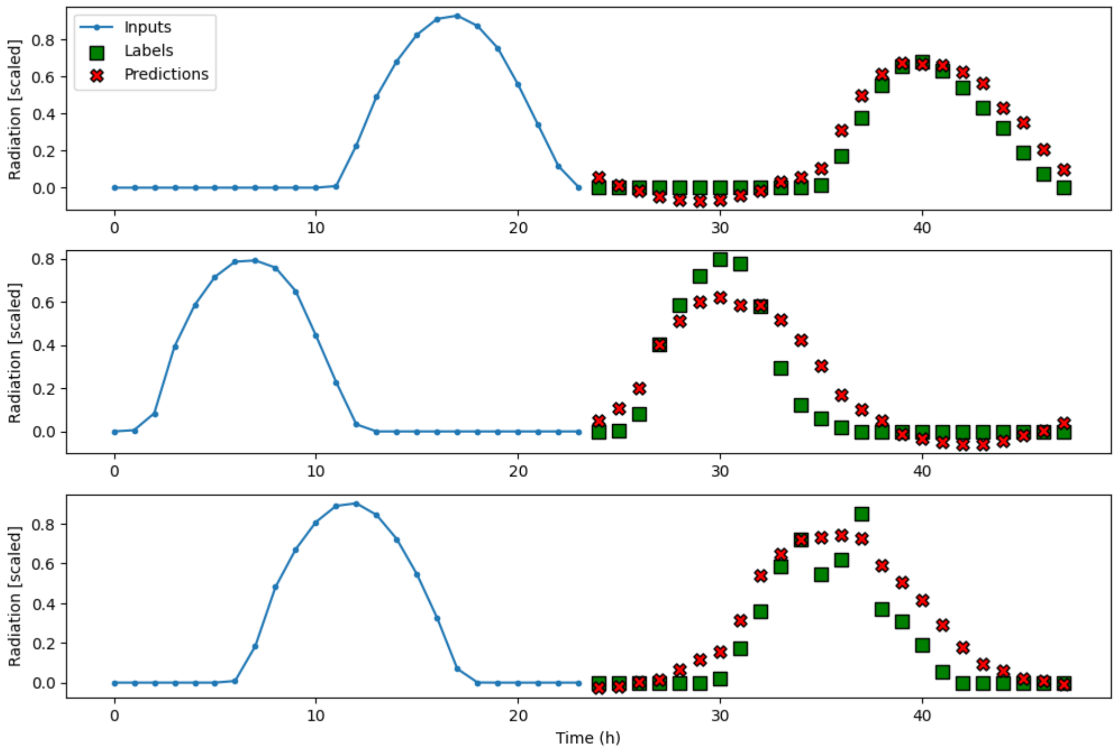
4.2 Linear model

This model consists of only an input layer and an output layer. Thus, only a sequence of weights is computed to generate predictions that are as close as possible to the labels.

[93]:
```python
#  build a model with one Dense output layer that has only one neuron, since we␣
↪are predicting only one target
linear = Sequential([
    Dense(1, kernel_initializer=tf.initializers.zeros)
])

history = compile_and_fit(linear, multi_window)

val_performance['Linear'] = linear.evaluate(multi_window.val)
performance['Linear'] = linear.evaluate(multi_window.test, verbose=0)
```

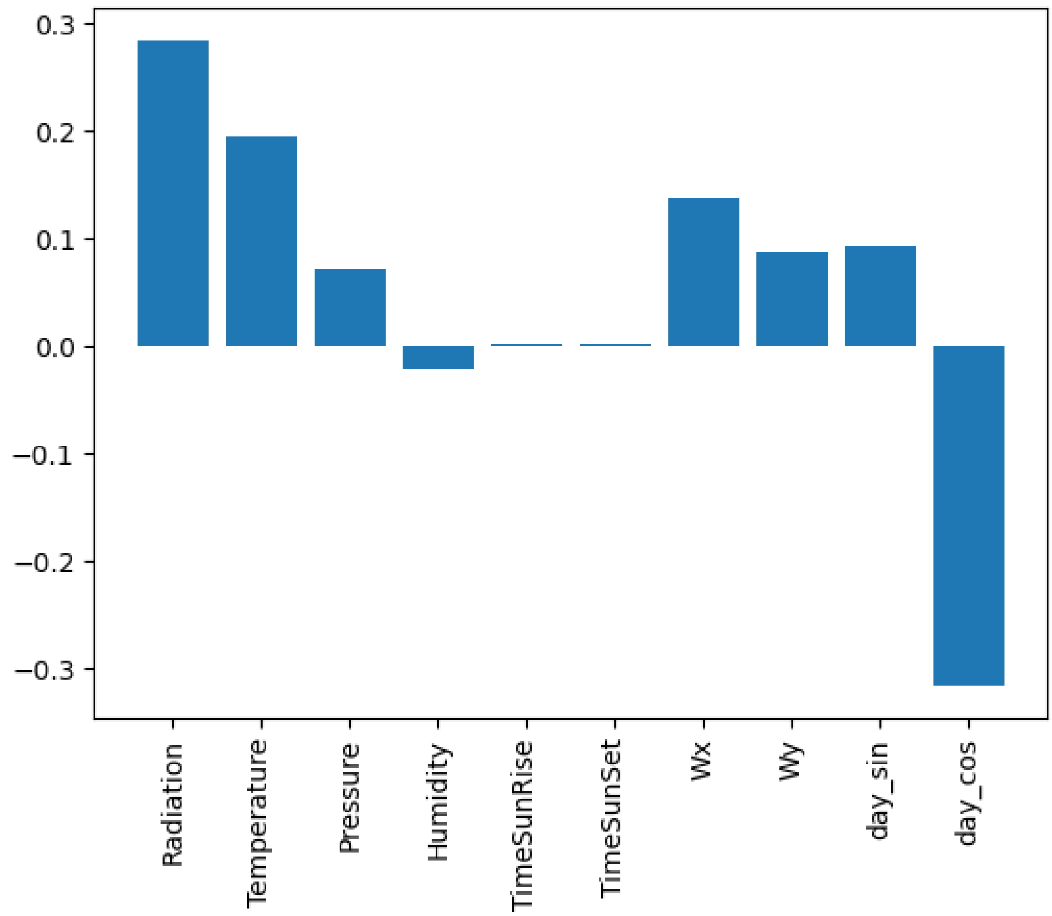
```
Epoch 1/50
63/63              1s 6ms/step - loss:
0.1027 - mean_absolute_error: 0.2111 - val_loss: 0.0370 -
val_mean_absolute_error: 0.1728
Epoch 2/50
63/63              0s 4ms/step - loss:
0.0582 - mean_absolute_error: 0.1992 - val_loss: 0.0274 -
val_mean_absolute_error: 0.1447
Epoch 3/50
63/63              0s 5ms/step - loss:
0.0419 - mean_absolute_error: 0.1651 - val_loss: 0.0212 -
val_mean_absolute_error: 0.1189
Epoch 4/50
63/63              1s 3ms/step - loss:
0.0310 - mean_absolute_error: 0.1379 - val_loss: 0.0179 -
val_mean_absolute_error: 0.1007
Epoch 5/50
63/63              0s 3ms/step - loss:
0.0249 - mean_absolute_error: 0.1182 - val_loss: 0.0167 -
val_mean_absolute_error: 0.0953
Epoch 6/50
63/63              0s 3ms/step - loss:
0.0207 - mean_absolute_error: 0.1050 - val_loss: 0.0167 -
val_mean_absolute_error: 0.0969
Epoch 7/50
63/63              0s 3ms/step - loss:
0.0186 - mean_absolute_error: 0.1000 - val_loss: 0.0173 -
val_mean_absolute_error: 0.1002
Epoch 8/50
63/63              0s 5ms/step - loss:
0.0173 - mean_absolute_error: 0.0974 - val_loss: 0.0178 -
val_mean_absolute_error: 0.1031
17/17              0s 2ms/step - loss:
0.0177 - mean_absolute_error: 0.1024
```

[94]:
```python
multi_window.plot(linear)
```



One advantage to linear models is that they're relatively simple to interpret. You can pull out the layer's weights and visualize the weight assigned to each input:

[95]:
```
# plot layer's weight for each feature
plt.bar(x = range(len(train_df.columns)),
        height=linear.layers[0].kernel[:,0].numpy())
axis = plt.gca()
axis.set_xticks(range(len(train_df.columns)))
_ = axis.set_xticklabels(train_df.columns, rotation=90)
```



4.3 Deep neural network

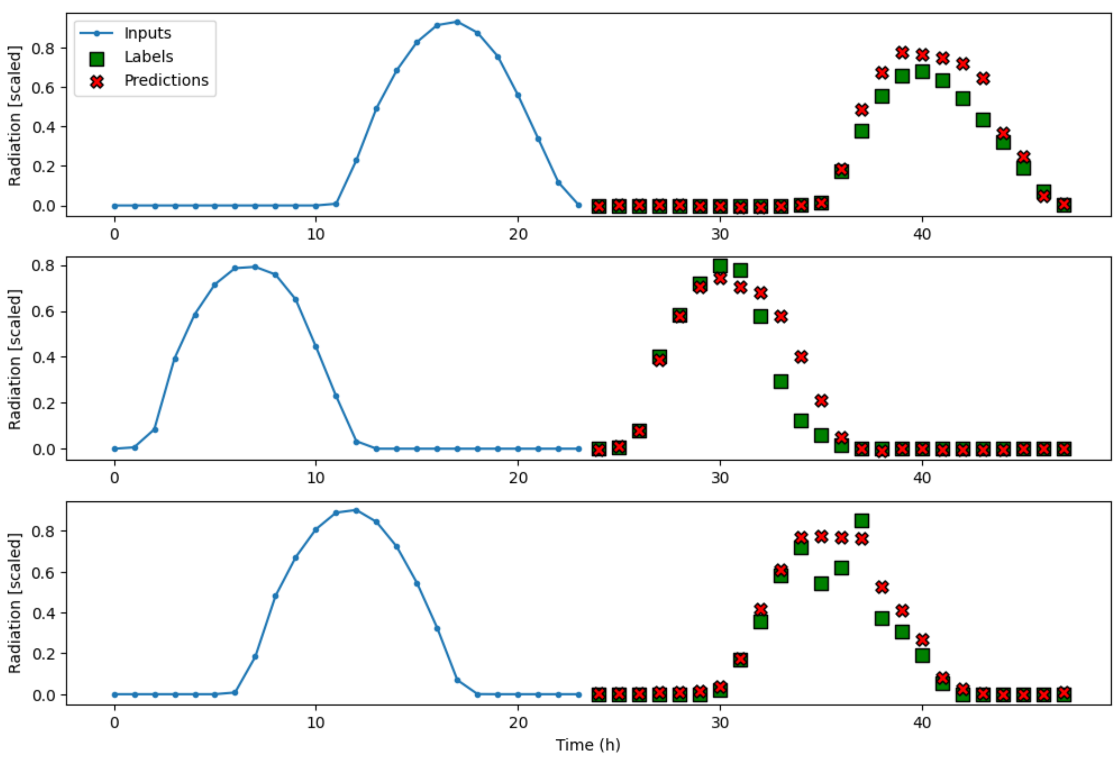
Here we'll stack two Dense layers with 64 neurons and use ReLU as the activation function. Then we'll train the model and store its performance for comparison.

[96]:
```
# stack 2 Dense layers with 64 neurons
dense = Sequential([
    Dense(64, activation='relu'),
    Dense(64, activation='relu'),
    Dense(1, kernel_initializer=tf.initializers.zeros),
])

history = compile_and_fit(dense, multi_window)

val_performance['Dense'] = dense.evaluate(multi_window.val)
performance['Dense'] = dense.evaluate(multi_window.test, verbose=0)
```

```
Epoch 1/50
63/63                 2s 7ms/step - loss:
```

```
0.0835 - mean_absolute_error: 0.1932 - val_loss: 0.0134 -
val_mean_absolute_error: 0.0752
Epoch 2/50
63/63              1s 6ms/step - loss:
0.0128 - mean_absolute_error: 0.0690 - val_loss: 0.0133 -
val_mean_absolute_error: 0.0700
Epoch 3/50
63/63              0s 5ms/step - loss:
0.0112 - mean_absolute_error: 0.0595 - val_loss: 0.0124 -
val_mean_absolute_error: 0.0640
Epoch 4/50
63/63              1s 6ms/step - loss:
0.0109 - mean_absolute_error: 0.0575 - val_loss: 0.0144 -
val_mean_absolute_error: 0.0707
Epoch 5/50
63/63              1s 4ms/step - loss:
0.0106 - mean_absolute_error: 0.0555 - val_loss: 0.0135 -
val_mean_absolute_error: 0.0684
Epoch 6/50
63/63              0s 5ms/step - loss:
0.0103 - mean_absolute_error: 0.0544 - val_loss: 0.0136 -
val_mean_absolute_error: 0.0684
17/17              0s 2ms/step - loss:
0.0132 - mean_absolute_error: 0.0677
```

[97]:
```
multi_window.plot(dense)
```



4.4 Long short-term memory (LSTM) model

The main advantage of the long short-term memory (LSTM) model is that it keeps information from the past in memory. This makes it especially suitable for treating sequences of data, like time series. It allows us to combine information from the present and the past to produce a prediction.

We'll feed the input sequence through an LSTM layer before sending it to the output layer, which remains a Dense layer with one neuron. We'll then train the model and store its performance in the dictionary for comparison at the end.

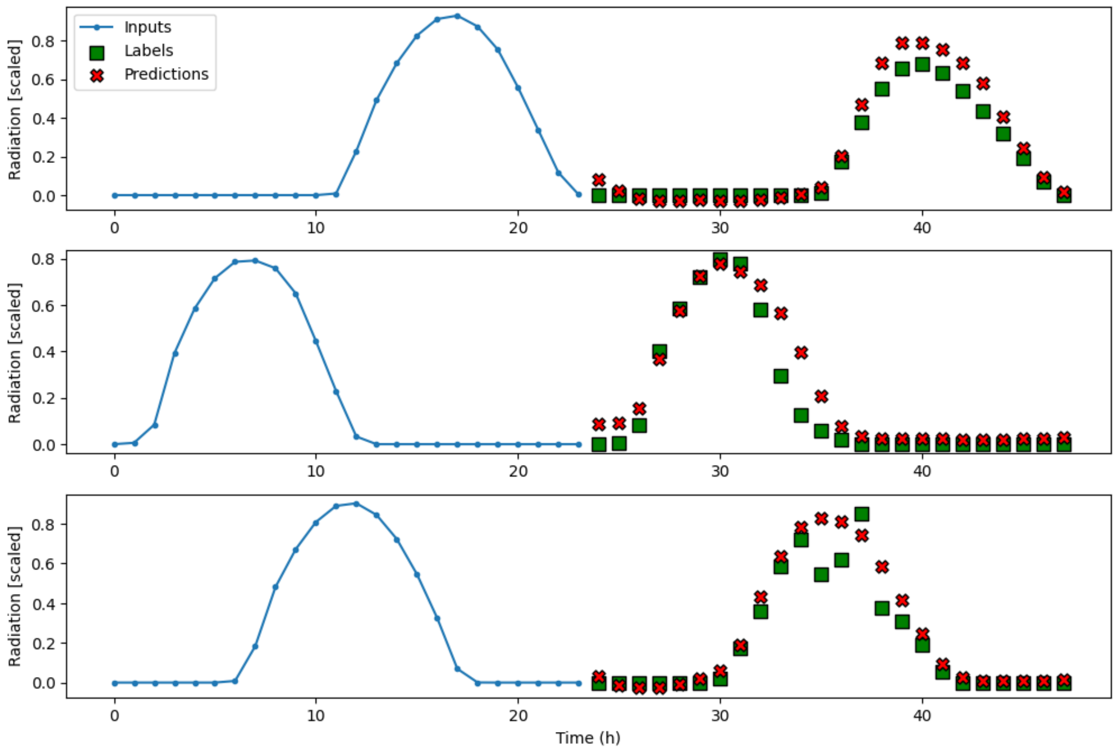
[98]:
```
# add a LSTM layer
lstm_model = Sequential([
    LSTM(32, return_sequences=True),
    Dense(1, kernel_initializer=tf.initializers.zeros),
])

history = compile_and_fit(lstm_model, multi_window)

val_performance['LSTM'] = lstm_model.evaluate(multi_window.val)
performance['LSTM'] = lstm_model.evaluate(multi_window.test, verbose=0)
```

```
Epoch 1/50
63/63           3s 15ms/step -
loss: 0.0965 - mean_absolute_error: 0.2132 - val_loss: 0.0176 -
val_mean_absolute_error: 0.1005
Epoch 2/50
63/63           1s 14ms/step -
loss: 0.0220 - mean_absolute_error: 0.1090 - val_loss: 0.0153 -
val_mean_absolute_error: 0.0880
Epoch 3/50
63/63           1s 12ms/step -
loss: 0.0154 - mean_absolute_error: 0.0829 - val_loss: 0.0152 -
val_mean_absolute_error: 0.0850
Epoch 4/50
63/63           1s 11ms/step -
loss: 0.0134 - mean_absolute_error: 0.0742 - val_loss: 0.0138 -
val_mean_absolute_error: 0.0802
Epoch 5/50
63/63           1s 14ms/step -
loss: 0.0130 - mean_absolute_error: 0.0721 - val_loss: 0.0135 -
val_mean_absolute_error: 0.0780
Epoch 6/50
63/63           1s 11ms/step -
loss: 0.0122 - mean_absolute_error: 0.0690 - val_loss: 0.0151 -
val_mean_absolute_error: 0.0818
Epoch 7/50
```

```
63/63              1s 13ms/step -
loss: 0.0116 - mean_absolute_error: 0.0665 - val_loss: 0.0131 -
val_mean_absolute_error: 0.0754
Epoch 8/50
63/63              1s 12ms/step -
loss: 0.0115 - mean_absolute_error: 0.0656 - val_loss: 0.0136 -
val_mean_absolute_error: 0.0770
Epoch 9/50
63/63              1s 11ms/step -
loss: 0.0112 - mean_absolute_error: 0.0644 - val_loss: 0.0123 -
val_mean_absolute_error: 0.0724
Epoch 10/50
63/63              1s 14ms/step -
loss: 0.0109 - mean_absolute_error: 0.0628 - val_loss: 0.0134 -
val_mean_absolute_error: 0.0767
Epoch 11/50
63/63              1s 11ms/step -
loss: 0.0107 - mean_absolute_error: 0.0615 - val_loss: 0.0131 -
val_mean_absolute_error: 0.0764
Epoch 12/50
63/63              1s 11ms/step -
loss: 0.0106 - mean_absolute_error: 0.0604 - val_loss: 0.0145 -
val_mean_absolute_error: 0.0826
17/17              0s 4ms/step - loss:
0.0144 - mean_absolute_error: 0.0827
```

[99]:

```
multi_window.plot(lstm_model)
```



4.5 Convolutional neural network (CNN)

A convolutional neural network (CNN) uses the convolution function to reduce the feature space. This effectively filters our time series and performs feature selection. Furthermore, a CNN is faster to train than an LSTM since the operations are parallelized, whereas the LSTM must treat one element of the sequence at a time.

Because the convolution operation reduces the feature space, we must provide a slightly longer input sequence to make sure that the output sequence contains 24 timesteps. How much longer it needs to be depends on the length of the kernel that performs the convolution operation. In this case, we'll use a kernel length of 3.

[100]:
```
# Define CNN related variables
KERNEL_WIDTH = 3
LABEL_WIDTH = 24
INPUT_WIDTH = LABEL_WIDTH + KERNEL_WIDTH - 1

# Define window object for CNN
cnn_multi_window = DataWindow(input_width=INPUT_WIDTH, label_width=LABEL_WIDTH,␣
↪shift=24, label_columns=[pred_column])
```

Next, we'll send the input through a Conv1D layer, which filters the input sequence. Then it is fed to a Dense layer with 32 neurons for learning before going to the output layer.

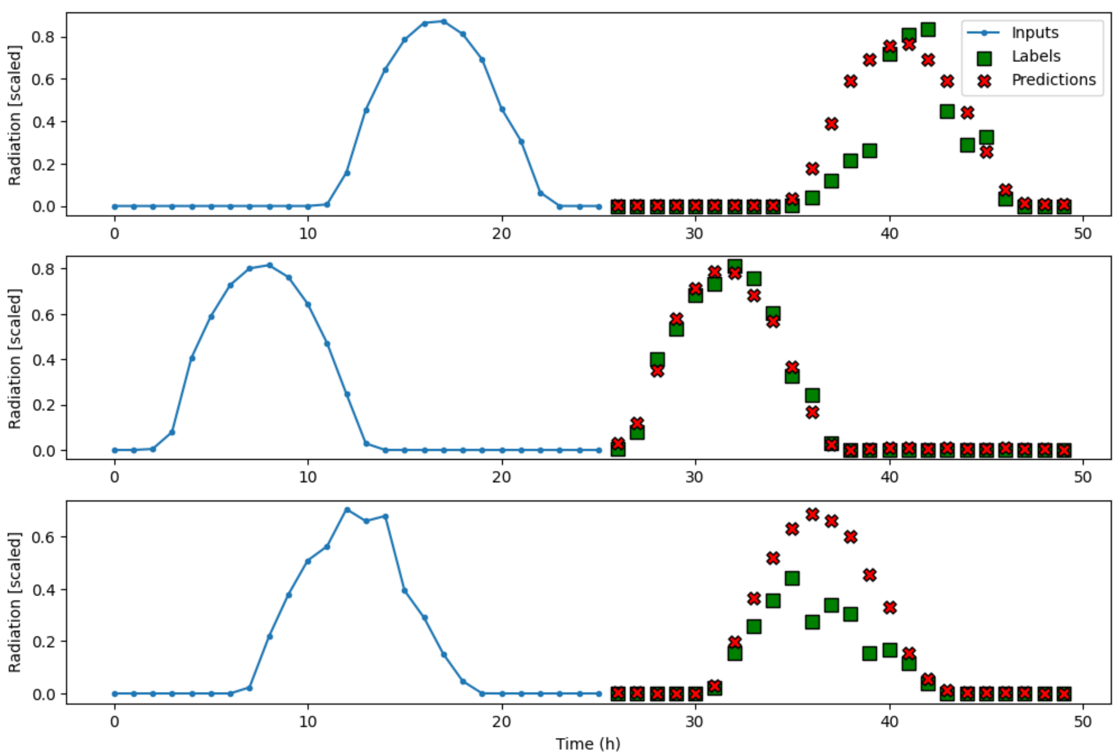
[101]:

```python
# stack a Conv2D layer
cnn_model = Sequential([
    Conv1D(32, activation='relu', kernel_size=(KERNEL_WIDTH)),
    Dense(units=32, activation='relu'),
    Dense(1, kernel_initializer=tf.initializers.zeros),
])

history = compile_and_fit(cnn_model, cnn_multi_window)

val_performance['CNN'] = cnn_model.evaluate(cnn_multi_window.val)
performance['CNN'] = cnn_model.evaluate(cnn_multi_window.test, verbose=0)
```

```
Epoch 1/50
63/63            2s 6ms/step - loss:
0.0866 - mean_absolute_error: 0.2084 - val_loss: 0.0138 -
val_mean_absolute_error: 0.0847
Epoch 2/50
63/63            0s 4ms/step - loss:
0.0134 - mean_absolute_error: 0.0776 - val_loss: 0.0121 -
val_mean_absolute_error: 0.0701
Epoch 3/50
63/63            0s 5ms/step - loss:
0.0108 - mean_absolute_error: 0.0597 - val_loss: 0.0130 -
val_mean_absolute_error: 0.0711
Epoch 4/50
63/63            0s 4ms/step - loss:
0.0106 - mean_absolute_error: 0.0576 - val_loss: 0.0154 -
val_mean_absolute_error: 0.0758
Epoch 5/50
63/63            0s 4ms/step - loss:
0.0106 - mean_absolute_error: 0.0575 - val_loss: 0.0116 -
val_mean_absolute_error: 0.0649
Epoch 6/50
63/63            0s 4ms/step - loss:
0.0105 - mean_absolute_error: 0.0560 - val_loss: 0.0125 -
val_mean_absolute_error: 0.0666
Epoch 7/50
63/63            0s 4ms/step - loss:
0.0101 - mean_absolute_error: 0.0547 - val_loss: 0.0142 -
val_mean_absolute_error: 0.0693
Epoch 8/50
63/63            0s 4ms/step - loss:
0.0101 - mean_absolute_error: 0.0543 - val_loss: 0.0116 -
val_mean_absolute_error: 0.0620
Epoch 9/50
63/63            0s 5ms/step - loss:
0.0099 - mean_absolute_error: 0.0533 - val_loss: 0.0122 -
```

```
val_mean_absolute_error: 0.0630
Epoch 10/50
63/63            1s 4ms/step - loss:
0.0097 - mean_absolute_error: 0.0529 - val_loss: 0.0133 -
val_mean_absolute_error: 0.0665
Epoch 11/50
63/63            0s 4ms/step - loss:
0.0096 - mean_absolute_error: 0.0521 - val_loss: 0.0113 -
val_mean_absolute_error: 0.0608
Epoch 12/50
63/63            0s 4ms/step - loss:
0.0097 - mean_absolute_error: 0.0525 - val_loss: 0.0114 -
val_mean_absolute_error: 0.0617
Epoch 13/50
63/63            1s 5ms/step - loss:
0.0097 - mean_absolute_error: 0.0530 - val_loss: 0.0113 -
val_mean_absolute_error: 0.0610
Epoch 14/50
63/63            0s 5ms/step - loss:
0.0092 - mean_absolute_error: 0.0511 - val_loss: 0.0117 -
val_mean_absolute_error: 0.0633
17/17            0s 2ms/step - loss:
0.0118 - mean_absolute_error: 0.0635
```

[102]:
```
cnn_multi_window.plot(cnn_model)
```



4.6 Combining a CNN with an LSTM

We know that LSTM is good at treating sequences of data, while CNN can filter a sequence of data. Therefore, it is interesting to test whether filtering a sequence before feeding it to an LSTM can result in a better-performing model.

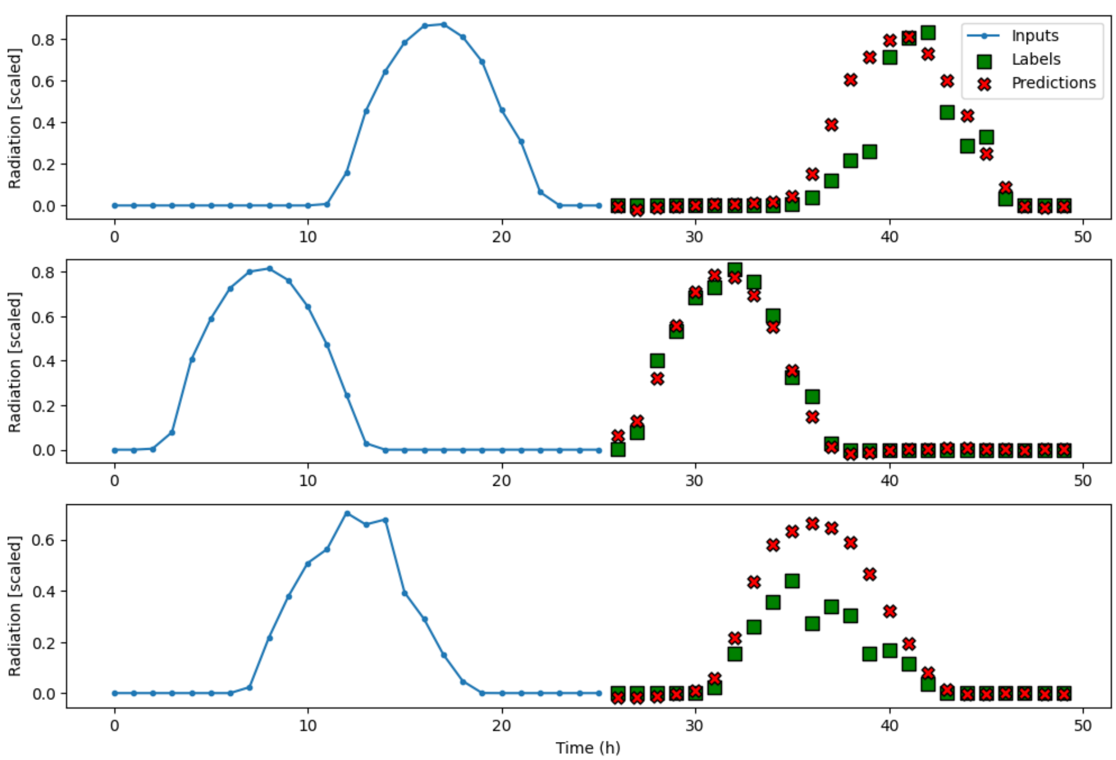
We'll feed the input sequence to a Conv1D layer, but use an LSTM layer for learning this time. Then we'll send the information to the output layer. Again, we'll train the model and store its performance.

[103]:
```
# add Conv1D + LSTM layers
cnn_lstm_model = Sequential([
    Conv1D(32, activation='relu', kernel_size=(KERNEL_WIDTH)),
    LSTM(32, return_sequences=True),
    Dense(1, kernel_initializer=tf.initializers.zeros),
])

history = compile_and_fit(cnn_lstm_model, cnn_multi_window)

val_performance['CNN + LSTM'] = cnn_lstm_model.evaluate(cnn_multi_window.val)
performance['CNN + LSTM'] = cnn_lstm_model.evaluate(cnn_multi_window.test,␣
 ↪verbose=0)
```

```
Epoch 1/50
63/63              4s 17ms/step -
loss: 0.0914 - mean_absolute_error: 0.2097 - val_loss: 0.0308 -
val_mean_absolute_error: 0.1293
Epoch 2/50
63/63              1s 12ms/step -
loss: 0.0164 - mean_absolute_error: 0.0899 - val_loss: 0.0195 -
val_mean_absolute_error: 0.0973
Epoch 3/50
63/63              1s 12ms/step -
loss: 0.0123 - mean_absolute_error: 0.0704 - val_loss: 0.0205 -
val_mean_absolute_error: 0.0972
Epoch 4/50
63/63              1s 13ms/step -
loss: 0.0112 - mean_absolute_error: 0.0653 - val_loss: 0.0176 -
val_mean_absolute_error: 0.0891
Epoch 5/50
63/63              1s 13ms/step -
loss: 0.0109 - mean_absolute_error: 0.0633 - val_loss: 0.0174 -
val_mean_absolute_error: 0.0867
Epoch 6/50
63/63              1s 13ms/step -
loss: 0.0105 - mean_absolute_error: 0.0612 - val_loss: 0.0160 -
```

```
val_mean_absolute_error: 0.0811
Epoch 7/50
63/63             1s 12ms/step -
loss: 0.0101 - mean_absolute_error: 0.0592 - val_loss: 0.0195 -
val_mean_absolute_error: 0.0874
Epoch 8/50
63/63             1s 14ms/step -
loss: 0.0107 - mean_absolute_error: 0.0603 - val_loss: 0.0132 -
val_mean_absolute_error: 0.0758
Epoch 9/50
63/63             1s 12ms/step -
loss: 0.0104 - mean_absolute_error: 0.0603 - val_loss: 0.0196 -
val_mean_absolute_error: 0.0856
Epoch 10/50
63/63             1s 12ms/step -
loss: 0.0099 - mean_absolute_error: 0.0568 - val_loss: 0.0167 -
val_mean_absolute_error: 0.0784
Epoch 11/50
63/63             1s 13ms/step -
loss: 0.0096 - mean_absolute_error: 0.0545 - val_loss: 0.0149 -
val_mean_absolute_error: 0.0733
17/17             0s 4ms/step - loss:
0.0147 - mean_absolute_error: 0.0726
```

[104]:
```
cnn_multi_window.plot(cnn_lstm_model)
```



4.7 The autoregressive LSTM model

The final model that we'll implement is an autoregressive LSTM (ARLSTM) model. Instead of generating the entire output sequence in a single shot, the autoregressive model will generate one prediction at a time and use that prediction as an input to generate the next one. This kind of architecture is present in state-of-the-art forecasting models, but it comes with a caveat.

If the model generates a very bad first prediction, this mistake will be carried on to the next predictions, which will magnify the errors. Nevertheless, it is worth testing this model to see if it works well in our situation.

[105]:
```python
# implement ARLSTM model
class AutoRegressive(Model):
    def __init__(self, units, out_steps):
        super().__init__()
        self.out_steps = out_steps
        self.units = units
        self.lstm_cell = LSTMCell(units)
        self.lstm_rnn = RNN(self.lstm_cell, return_state=True)
        self.dense = Dense(train_df.shape[1])

    def warmup(self, inputs):
        x, *state = self.lstm_rnn(inputs)
        prediction = self.dense(x)

        return prediction, state

    def call(self, inputs, training=None):
        predictions = []
        prediction, state = self.warmup(inputs)

        predictions.append(prediction)

        for n in range(1, self.out_steps):
            x = prediction
            x, state = self.lstm_cell(x, states=state, training=training)

            prediction = self.dense(x)
            predictions.append(prediction)

        predictions = tf.stack(predictions)
        predictions = tf.transpose(predictions, [1, 0, 2])

        return predictions
```

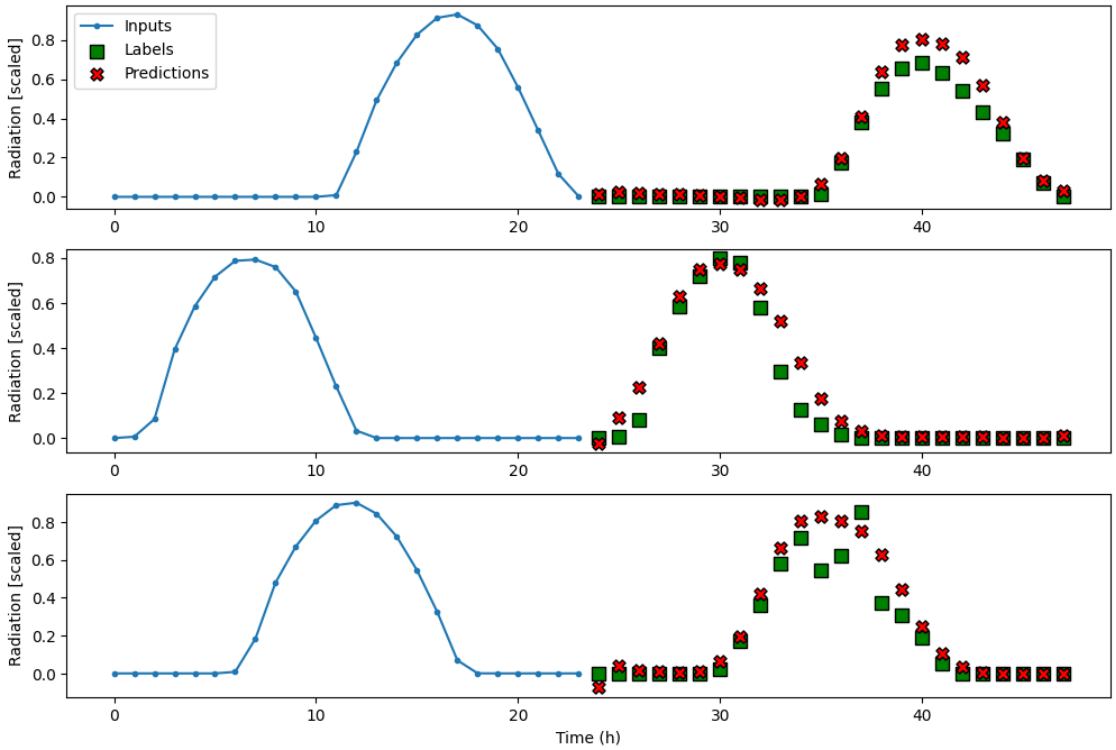
[106]:

```
AR_LSTM = AutoRegressive(units=32, out_steps=24)

history = compile_and_fit(AR_LSTM, multi_window)

val_performance['AR - LSTM'] = AR_LSTM.evaluate(multi_window.val)
performance['AR - LSTM'] = AR_LSTM.evaluate(multi_window.test, verbose=0)
```

```
Epoch 1/50
63/63            10s 29ms/step -
loss: 0.0992 - mean_absolute_error: 0.2379 - val_loss: 0.0435 -
val_mean_absolute_error: 0.1753
Epoch 2/50
63/63            1s 16ms/step -
loss: 0.0559 - mean_absolute_error: 0.1870 - val_loss: 0.0364 -
val_mean_absolute_error: 0.1412
Epoch 3/50
63/63            1s 16ms/step -
loss: 0.0176 - mean_absolute_error: 0.0984 - val_loss: 0.0339 -
val_mean_absolute_error: 0.1280
Epoch 4/50
63/63            1s 16ms/step -
loss: 0.0149 - mean_absolute_error: 0.0868 - val_loss: 0.0371 -
val_mean_absolute_error: 0.1304
Epoch 5/50
63/63            1s 17ms/step -
loss: 0.0144 - mean_absolute_error: 0.0834 - val_loss: 0.0284 -
val_mean_absolute_error: 0.1160
Epoch 6/50
63/63            1s 16ms/step -
loss: 0.0133 - mean_absolute_error: 0.0789 - val_loss: 0.0301 -
val_mean_absolute_error: 0.1139
Epoch 7/50
63/63            1s 17ms/step -
loss: 0.0127 - mean_absolute_error: 0.0758 - val_loss: 0.0239 -
val_mean_absolute_error: 0.1046
Epoch 8/50
63/63            1s 17ms/step -
loss: 0.0121 - mean_absolute_error: 0.0729 - val_loss: 0.0298 -
val_mean_absolute_error: 0.1115
Epoch 9/50
63/63            1s 17ms/step -
loss: 0.0119 - mean_absolute_error: 0.0712 - val_loss: 0.0227 -
val_mean_absolute_error: 0.1000
Epoch 10/50
63/63            1s 17ms/step -
loss: 0.0116 - mean_absolute_error: 0.0694 - val_loss: 0.0237 -
val_mean_absolute_error: 0.1016
```

```
Epoch 11/50
63/63              1s 17ms/step -
loss: 0.0113 - mean_absolute_error: 0.0677 - val_loss: 0.0274 -
val_mean_absolute_error: 0.1058
Epoch 12/50
63/63              1s 16ms/step -
loss: 0.0110 - mean_absolute_error: 0.0659 - val_loss: 0.0235 -
val_mean_absolute_error: 0.1000
17/17              0s 6ms/step - loss:
0.0242 - mean_absolute_error: 0.1012
```

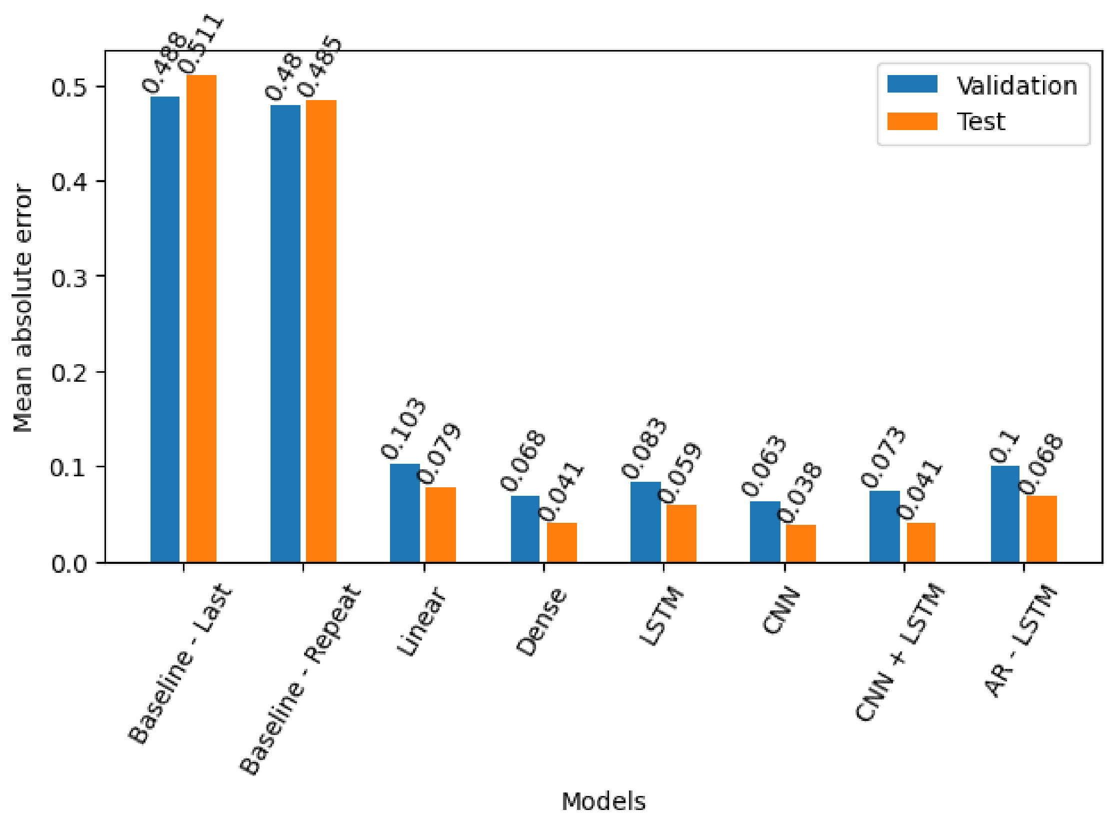
[107]:
```
multi_window.plot(AR_LSTM)
```



4.8 Selecting the best model

[108]:
```
# plot the MAE on both the validation and test sets
mae_val = [v[1] for v in val_performance.values()]
mae_test = [v[1] for v in performance.values()]

x = np.arange(len(performance))

fig, ax = plt.subplots()
ax.bar(x - 0.15, mae_val, width=0.25, label='Validation')
```

```
ax.bar(x + 0.15, mae_test, width=0.25, label='Test')
ax.set_ylabel('Mean absolute error')
ax.set_xlabel('Models')

for index, value in enumerate(mae_val):
    plt.text(x=index - 0.15, y=value+0.005, s=str(round(value, 3)),␣
 ↪ha='center', rotation=60)

for index, value in enumerate(mae_test):
    plt.text(x=index + 0.15, y=value+0.0025, s=str(round(value, 3)),␣
 ↪ha='center', rotation=60)

plt.xticks(ticks=x, labels=performance.keys(), rotation=60)
plt.legend(loc='best')
plt.tight_layout()
```



5 Summary

- In this project, we compare serveral models to predict the solar radiation in the next 24 hours. Based on our simulation, **CNN** with 1 convolution layers has the best performance.

We should choose this model to predict the upcoming radiation.

- Sometimes it will have different result, but usually the best preformance will be one of the followings: **Deep neural network**, **CNN**, and **CNN with LSTM**.
- Linear model is simple and fast. When it is only necessary to understand future trends in the data without requiring precise numbers, a linear model is a good choice.
- Handling missing values in time series data is very tricky. If the feature(like radiation) has the trend seasonality, we need to impute missing values using some special techniques such as STL Decomposition to try to simulate the trend seasonality.
- Scaling the data is also very important. Using different scaling method will get different preformance. Here we use MinMaxScaler because we want to keep the sample distribution after scaling the data.
- In feature engineering, we transform some features to a more meaningful values. We combine *wind degree* and *wind speed* to **wind vector**, to make their values are more mingingful. We also put time features like sine and cosine of day wave to make daily trend is displaying in the dataset.

6 Reference

https://www.openml.org/search?type=data&status=active&id=43751

https://www.tensorflow.org/tutorials/structured_data/time_series

https://medium.com/@aaabulkhair/data-imputation-demystified-time-series-data-69bc9c798cb7

Marco Peixeiro, “Time Series Forecasting in Python”, Manning Publications, 2022