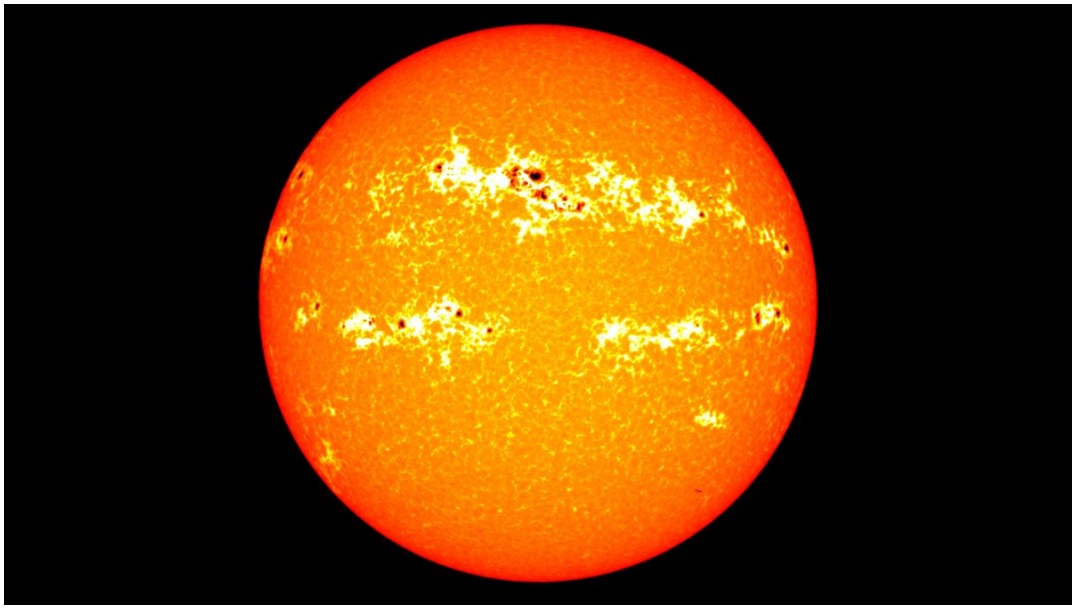


Time series analysis

-Sunspots-

-See appendix for full code-

Section 1: Data

I will use a dataset from the article:

https://machinelearningmastery.com/time-series-datasets-for-machine-learning/ (https://machinelearningmastery.com/time-series-datasets-for-machine-learning/) The data is attributed to Andrews & Herzberg (1985)

Quote from data set description:

This dataset describes a monthly count of the number of observed sunspots for just over 230 years (1749-1983).

Shape of data:

The data has 2820 rows and only date and number of sunspots as columns

Missing data:

There is no missing data

Data types:

Month is a data type object and Sunspots is float64. The month column is afterwards set as datetime and as index.

Data cleaning:

No cleaning required.

Feature engineering:

No feature engineering required.

Other findings:

The data seems stationary, with same mean and variance over the period and shows signs of autocorrelation strucutre.

Section 2: Main objectives of analysis

The goal of this project is to forecast a year into the future. 24 months is set aside, then the model is given the rest to train, and will be given 12 months as input to forecast, and output the next 12 months

---

Section 3: Model

The models tried are variations of a 1 hidden-layered SimpleRNN and 1 hidden-layered LSTM, both from the Keras library. They are used with different number of cells and different number of epochs. Both have a single dense layer as output.

The SimpleRNN was trained with combinations of 50 and 100 epochs, and 10 and 30 cell units.

The LSTM, however, was trained with combinations of 100, 1000 and 5000 epochs and 20, 50 and 100 cell units.

Section 4: Findings

None of the RNN models found the pattern or something close to it. They all predicted the last output as the new prediction. This is due to the low number of epochs and cell units.

The LSTM models began a non-linear prediction once it ran 1000 epochs and more.

The best model was a LSTM of 50 and 100 cell units and 1000 epochs, as expected. Though they look overfitted.

Section 5: Possible flaws and plan to revisit

It would be interesting to run the RNN and LSTM the same way to see how they differ. This is not done due to time constraint.

One could try to predict differnet periods of time of the model to see if it has overfit, which is likely with the high number of epochs, and a thing Deep Learning model are vulnerable to.

Lastly, it would be highly interesting to try adding more layers to the model, and could be done in the future.

Appendix: Workbook

In [2]:

```
# Imports
import os
#os.chdir('data')
import pandas as pd
import numpy as np
from datetime import datetime
from statsmodels.tsa.stattools import adfuller

# Plotting
import matplotlib.pyplot as plt
from statsmodels.tsa.seasonal import seasonal_decompose
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# Network
import keras
import tensorflow as tf
from keras.models import Sequential
from keras.layers import Dense, SimpleRNN, LSTM
from tensorflow.keras import layers
```

```
C:\Users\brosb\Anaconda3\lib\site-packages\statsmodels\tools\_testing.py:19:
FutureWarning: pandas.util.testing is deprecated. Use the functions in the p
ublic API at pandas.testing instead.
  import pandas.util.testing as tm
Using TensorFlow backend.
```

In [5]:

```
# DataAnalysis class.
# For time series: x is dates, and y is the target column. Ensure that the date is properly
class DataAnalysis:
    '''
    The DataAnalysis class is used for objectives such as: 'Time Series', 'Classification'
    '''
    def __init__(self, objective, df, x, y):

        # ----------------------------THIS WORKS-------------------------------------
        # Define objective
        self.objective = objective
        print('The objective of the analysis is: ', objective)
        print('-------------------------------------------------------------------------------
        # ----------------------------THIS WORKS-------------------------------------


        # ----------------------------THIS WORKS-------------------------------------
        # Define dataframe
        if isinstance(df, pd.DataFrame):
            self.df = df
            print('This is the DataFrame to be analyzed: ')
            display(df.head())
        elif isinstance(df, str):
            self.df = pd.read_csv(df)
            print('The DataFrame to be analyzed is: ', df)
        else:
            print('Error, not appropriate input. Input should either be Pandas Dataframe or

        print('-------------------------------------------------------------------------------
        # ----------------------------THIS WORKS-------------------------------------


        # ----------------------------THIS WORKS-------------------------------------
        # Define x and y
        self.x = self.df[x]
        self.y = self.df[y]
        # ----------------------------THIS WORKS-------------------------------------

        # Info
        print('This is the info of the dataframe \n')
        print(self.df.info(), '\n');
        print('-------------------------------------------------------------------------------


        # Plot y against index
        print('This is the plot of the Y variable with the index\n')
        plt.rcParams["figure.figsize"] = (20,3)
        self.y.plot()

        # Time series components
        if self.objective == 'Time Series':
            print('Remember to consider: ')
            print('Does the series contain trend, seasonality or clear noise?')
            print('Does the series seem additivie, multiplicative or psuedo-additive?')
            print('Is the series stationary? Consider constant mean, constant variance, con


    def decompose_time_series(self, decompose_series):
        self.decompose_series = decompose_series
```

```
        # Decompose
        ss_decomposition = seasonal_decompose(x=self.decompose_series)
        estimated_trend = ss_decomposition.trend
        estimated_seasonal = ss_decomposition.seasonal
        estimated_residual = ss_decomposition.resid

        # Plot
        plt.plot(estimated_trend)
        plt.title('Estimated trend')
        plt.show()
        plt.plot(estimated_seasonal, label = 'Estimated ')
        plt.title('Estimated seasonality')
        plt.show()
        plt.plot(estimated_residual)
        plt.title('Estimated residual')
        plt.show()

    # ---------------------------THIS WORKS-------------------------------------
    def identify_clean_time_series(self, x, y):
        if self.df[y].isnull().sum() > 0:
            print('Y variable needs to be cleaned. It contains NaN values')
            print('The number of NaN values is: ', self.df[y].isnull().sum())
            print('-------------------------------------------------------')
    # ---------------------------THIS WORKS-------------------------------------


    # ---------------------------THIS WORKS-------------------------------------
    def clean_time_series_interpolate(self, x, y):
        if self.df[y].isnull().sum() > 0:
            self.df[y] = self.df[y].interpolate()
            print('The data has been interpolated. Number of NaN values is: ', self.df[y].i

            if self.df[y].isnull().sum() > 0:
                print('It is still not clean, this needs special help. The number of NaN va
                      self.df[y].isnull().sum())
            else:
                print('This series is clean')
        else:
            print('This series is clean')



    def check_stationarity(self, df):
        print('The series can be assesed to be stationary with the help of histogram and Di
        df.hist()
        plt.title('Distribution of data')
        plt.show()

        self.adf, self.pvalue, self.usedlag, self.nobs, self.critical_values, self.icbest =
        print('The Dickey-Fuller test for stationarity reveals the following: ')
        print(f'ADF test value: {round(self.adf,2)} | P-value {self.pvalue}')

        print('-----------------------------------------------------------------------')
        print('If stationarity is tested at a p-value of 0.05, ')
        if self.pvalue < 0.05:
            print('the series is stationary')
        elif self.pvalue > 0.05:
            print('the series is non-stationary')
        else:
            print('Something went wrong')
```

```
    # ---------------------------THIS WORKS-----------------------------------

    # ---------------------------THIS WORKS-----------------------------------
    def get_first_dates(self, df, column, n_units):
        return self.df[column][:-n_units]

    def plot_first_dates(self, df, column, n_units):
        print(f'This is the dates without the last {n_units} dates of {column}')
        self.get_first_dates(df, column,n_units).plot()
        plt.title(f'{column}')


    def get_last_dates(self, df, column, n_units):
        return self.df[column][-n_units:]

    def plot_last_dates(self, df, column, n_units):
        print(f'This is the last {n_units} dates of {column}')
        self.get_last_dates(df, column,n_units).plot()
        plt.title(f'{column}')
    # ---------------------------THIS WORKS-----------------------------------

    # TRAIN TEST SPLITTING
    def train_test_split_time_series(self, df, y, test_dates, subset = False, subset_start

        # Split into first and last dates depeding on specified test_dates
        self.train = self.get_first_dates(self.df, y, test_dates)
        self.test = self.get_last_dates(self.df, y, test_dates)

        # Able to decrease size of data
        if subset == True:
            self.train = self.train[subset_start:]
        else:
            pass

        print('The shape of the train set is: ', self.train.shape)
        print('The shape of the test set is: ', self.test.shape)

        #Return train and test
        return self.train, self.test


    def X_y_split_time_series(self, train, test, len_of_sequence, features = 1):
        self.train_X = []
        self.train_y = []

        for i in range(0, train.shape[0]-len_of_sequence):
            self.train_X.append(train[i:i+len_of_sequence])
            self.train_y.append(train[i+len_of_sequence])

        # Train X
        self.train_X = np.array(self.train_X)
        self.train_X = self.train_X.reshape(self.train_X.shape[0],self.train_X.shape[1],fea

        # Train y
        self.train_y = np.array(self.train_y)

        # Tests
        self.test_X = np.array(test[:len_of_sequence])
        self.test_y = np.array(test[len_of_sequence:])
```

```
        # Info
        print('The shape of train_X is: ', self.train_X.shape)
        print('The shape of train_y is: ', self.train_y.shape)
        print('The shape of test_X is: ', self.test_X.shape)
        print('The shape of test_y is: ', self.test_y.shape)


        return self.train_X, self.train_y, self.test_X, self.test_y


    # CREATING THE NETWORK
    def make_network_for_timeseries(self, train_X, train_y, n_cells, epochs, cell1 = Simple
                                    batch_size = 64, verbose = 1,
                                    loss='mean_squared_error', optimizer = 'adam'):

        # Initialize network object
        self.network = Sequential()

        # Make first hidden state
        self.network.add(cell1(n_cells, input_shape = (train_X.shape[1], features)))

        # Add output layer
        self.network.add(Dense(1))

        # Compile and fit
        self.network.compile(loss=loss, optimizer=optimizer)
        self.network.fit(train_X, train_y, epochs = epochs, batch_size = batch_size, verbos

        return self.network

    # PREDICTING
    def predict_time_series(self, network, test_X, test_y):
        self.test_X_copy = test_X.copy().reshape(1,-1,1)
        self.y_pred = []

        # Make predictions
        for _ in range(len(test_y)):
            self.pred = network.predict(self.test_X_copy)
            self.y_pred.append(self.pred)

            # Include new predictions as the last value and remove the first (for future, c
            self.test_X_copy[:, :-1, :] = self.test_X_copy[:, 1:, :]
            self.test_X_copy[:, -1, :] = self.pred

        self.preds = np.array(self.y_pred).reshape(-1,1)

        return self.preds


    # PLOTTING PREDICTIONS
    def plot_time_series_predictions(self, test_X, test_y, y_preds, test_dates = test_dates
        # Range
        start_range = range(1, test_X.shape[0]+1)
        predict_range = range(test_X.shape[0], test_dates)

        # Plotting
        plt.plot(start_range, test_X)
        plt.plot(predict_range, test_y, color = 'orange')
        plt.plot(predict_range, y_preds, color = 'teal', linestyle = '--')
```

```
        # Title and legend
        plt.title('Test data and predictions')
        plt.legend(['Initial Series','Target Series','Predictions'])
```

1. Variables and data

In [6]:

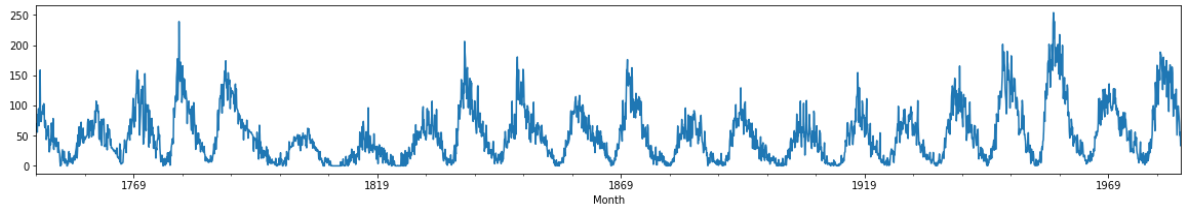
```
# Define variables
# Objective
objective = 'Time Series'

# Dataframe
df = pd.read_csv('monthly-sunspots.csv')

# Specific for Sunspots dataframe
df.set_index(pd.to_datetime(df['Month']), inplace=True)

# Define x
x = 'Month'

# Define y
y = 'Sunspots'

# Interested in testing last 24 days
test_dates = 24

# Length of sequence (for the current dataset, it would be interesting to forecast a year i
len_of_sequence = 12

# Create object of DataAnalysis class
model = DataAnalysis(objective, df, x, y)
```

```
The objective of the analysis is:  Time Series
------------------------------------------------------------------------------
--
This is the DataFrame to be analyzed:
```

	Month	Sunspots
Month		
1749-01-01	1749-01	58.0
1749-02-01	1749-02	62.6
1749-03-01	1749-03	70.0
1749-04-01	1749-04	55.7
1749-05-01	1749-05	85.0

```
------------------------------------------------------------------------------
----
This is the info of the dataframe

<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2820 entries, 1749-01-01 to 1983-12-01
Data columns (total 2 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   Month     2820 non-null   object
 1   Sunspots  2820 non-null   float64
dtypes: float64(1), object(1)
memory usage: 66.1+ KB
None

------------------------------------------------------------------------------
----
```

```
This is the plot of the Y variable with the index

Remember to consider:
Does the series contain trend, seasonality or clear noise?
Does the series seem additivie, multiplicative or psuedo-additive?
Is the series stationary? Consider constant mean, constant variance, const
ant autocorrelation and periodic components
```



In [7]:

```
# Check
print(model.df.isna().sum())
print('')

# Check if needs to be cleaned
model.identify_clean_time_series(x,y)

# Clean
model.clean_time_series_interpolate(x,y)
```

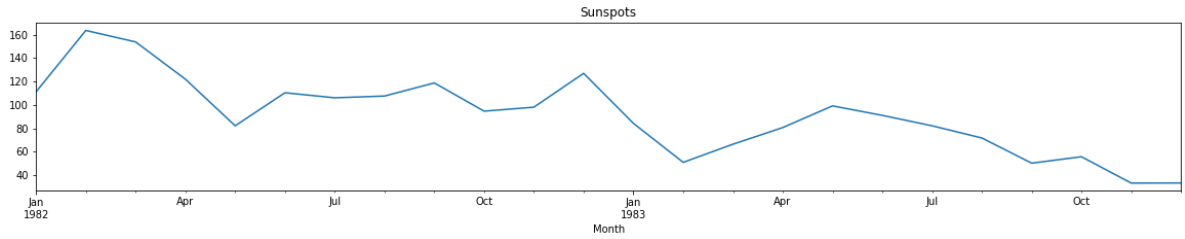
```
Month       0
Sunspots    0
dtype: int64

This series is clean
```

2. EDA

In [8]:

```
# Lets take a look at the last year
model.get_last_dates(df, 'Sunspots', 24)
model.plot_last_dates(df, 'Sunspots', 24)
```

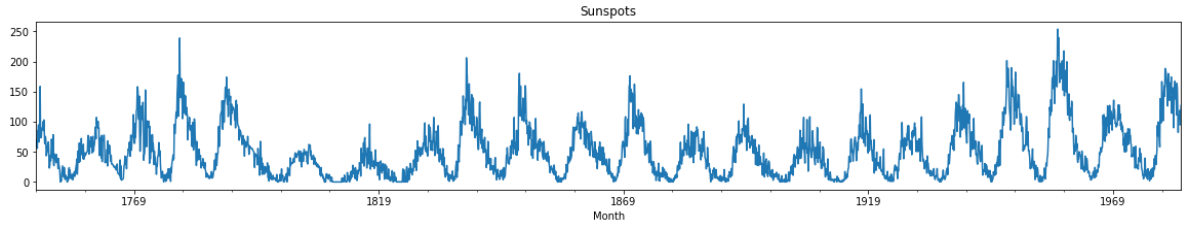
```
This is the last 24 dates of Sunspots
```



In [9]:

```
# And the first years
model.get_first_dates(df, 'Sunspots', 12)
model.plot_first_dates(df, 'Sunspots', 12)
```

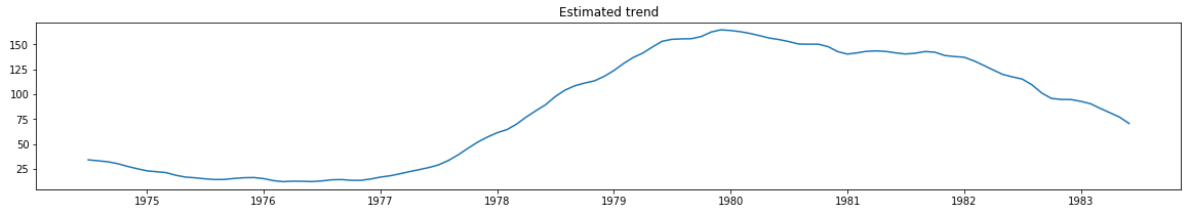
This is the dates without the last 12 dates of Sunspots

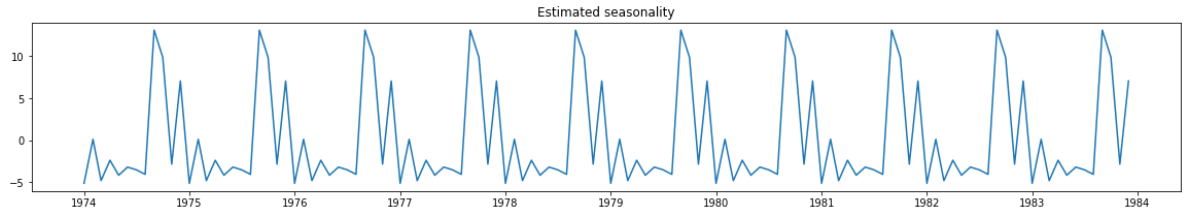


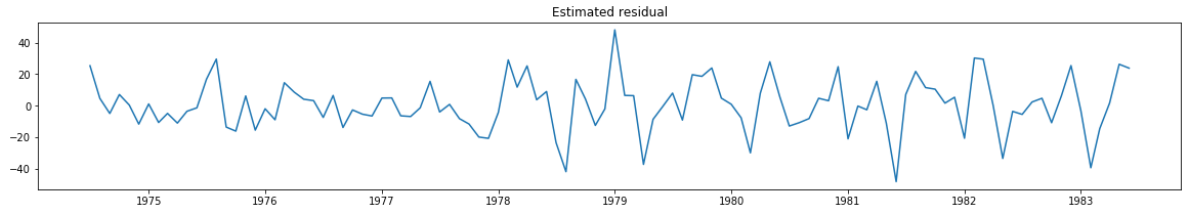
Decomposition

In [10]:

```
last = model.get_last_dates(model.df, y, 120)
model.decompose_time_series(last)
```





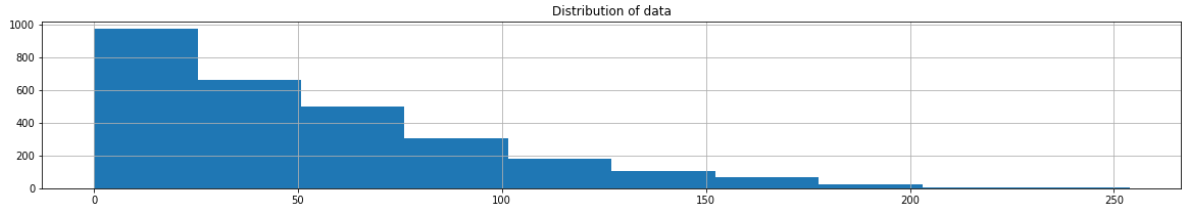


Stationarity

In [11]:

```
model.check_stationarity(model.df[y])
```

```
The series can be assesed to be stationary with the help of histogram and Di
ckey-Fuller test
```

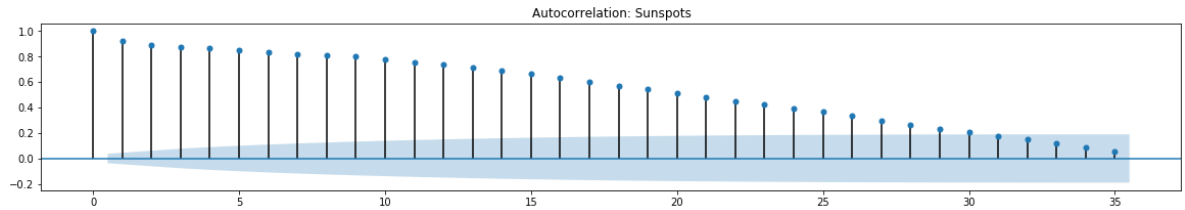


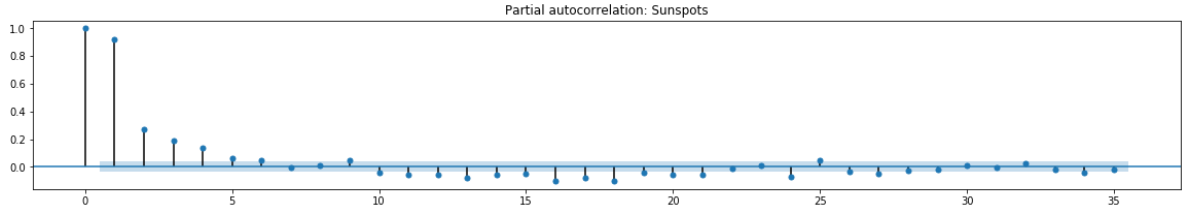

```
The Dickey-Fuller test for stationarity reveals the following:
ADF test value: -9.57 | P-value 2.333452143866187e-16
------------------------------------------------------------------------
If stationarity is tested at a p-value of 0.05,
the series is stationary
```

ACF and PACF

In [12]:

```
# ACF
plot_acf(model.df[y], title = f'Autocorrelation: {y}', lags = 35);

# PACF
plot_pacf(model.df[y], title = f'Partial autocorrelation: {y}', lags = 35);
```





3. Forecasting

In [13]:

```
# Train-test
train, test = model.train_test_split_time_series(df, y, test_dates, subset = True, subset_s
```

```
The shape of the train set is:  (796,)
The shape of the test set is:  (24,)
```

In [14]:

```
# X and y split
train_X, train_y, test_X, test_y = model.X_y_split_time_series(train, test, len_of_sequence
```

```
The shape of train_X is:  (784, 12, 1)
The shape of train_y is:  (784,)
The shape of test_X is:  (12,)
The shape of test_y is:  (12,)
```

3.1 Create model

In [15]:

```
epochs = 10
n_cells = 10

network = model.make_network_for_timeseries(train_X, train_y, n_cells, epochs, features = 1
                                    batch_size = 64, verbose = 1,
                                    loss='mean_squared_error', optimizer = 'adam')
```

```
Epoch 1/10
784/784 [==============================] - 0s 458us/step - loss: 7199.2981
Epoch 2/10
784/784 [==============================] - 0s 48us/step - loss: 7079.7178
Epoch 3/10
784/784 [==============================] - 0s 52us/step - loss: 6993.3911
Epoch 4/10
784/784 [==============================] - 0s 55us/step - loss: 6960.8694
Epoch 5/10
784/784 [==============================] - 0s 52us/step - loss: 6938.3198
Epoch 6/10
784/784 [==============================] - 0s 51us/step - loss: 6917.8188
Epoch 7/10
784/784 [==============================] - 0s 52us/step - loss: 6898.0430
Epoch 8/10
784/784 [==============================] - 0s 54us/step - loss: 6878.4580
Epoch 9/10
784/784 [==============================] - 0s 56us/step - loss: 6858.6022
Epoch 10/10
784/784 [==============================] - 0s 56us/step - loss: 6835.0848
```

3.2 Predict

In [16]:

```
y_preds = model.predict_time_series(network, test_X, test_y)
```

3.3 Plot

In [17]:

```
model.plot_time_series_predictions(test_X, test_y, y_preds, test_dates = test_dates)
```

Lets see some results

In [18]:

```
# Defining function to evaluate model
def calc_MSE(test_y, y_preds):
    assert test_y.shape == y_preds.shape
    error = (test_y - y_preds).sum()
    sq_error = error**2
    mse = sq_error / len(test_y)

    return mse
```

SimpleRNN

In [19]:

```
epoch_list = [50,100]
cell_units = [10,30]

for i in epoch_list:
    for j in cell_units:
        print('Following combination is finished: ')
        print(f'Cell units = {j} | epochs = {i}')
        MSE = calc_MSE(test_y, y_preds.reshape(12))
        print('Mean Squared Error for this combination is : ', round(MSE, 2))
        # Create model
        network = model.make_network_for_timeseries(train_X, train_y, n_cells=j, epochs=i,
                                    batch_size = 64, verbose = 0,
                                    loss='mean_squared_error', optimizer = 'adam')
        # Predict
        y_preds = model.predict_time_series(network, test_X, test_y)

        # Plotting
        model.plot_time_series_predictions(test_X, test_y, y_preds, test_dates = test_dates

        plt.show() # SimpleRNN
```

```
Following combination is finished:
Cell units = 10 | epochs = 50
Mean Squared Error for this combination is :  50615.47
```


```
Following combination is finished:
Cell units = 30 | epochs = 50
Mean Squared Error for this combination is :  41351.17
```


```
Following combination is finished:
Cell units = 10 | epochs = 100
Mean Squared Error for this combination is :  25378.65
```


```
Following combination is finished:
Cell units = 30 | epochs = 100
Mean Squared Error for this combination is :  32137.18
```


Test data and predictions
Initial Series
Target Series
Predictions
160
140
120
100
80
60
40
0
5
10
15
20

LSTM

In [20]:

```
epoch_list = [100, 1000, 5000]
cell_units = [20,50, 100]

for i in epoch_list:
    for j in cell_units:
        print('Following combination is finished: ')
        print(f'Cell units = {j} | epochs = {i}')

        # Create model
        network = model.make_network_for_timeseries(train_X, train_y, cell1 = LSTM, n_cells
                                    batch_size = 64, verbose = 0,
                                    loss='mean_squared_error', optimizer = 'adam')
        # Predict
        y_preds = model.predict_time_series(network, test_X, test_y)

        # Performance
        MSE = calc_MSE(test_y, y_preds.reshape(12))
        print('Mean Squared Error for this combination is : ', round(MSE, 2))

        # Plotting
        model.plot_time_series_predictions(test_X, test_y, y_preds, test_dates = test_dates

        plt.show()
```

```
Following combination is finished:
Cell units = 20 | epochs = 100
Mean Squared Error for this combination is :  18559.92
```


```
Following combination is finished:
Cell units = 50 | epochs = 100
Mean Squared Error for this combination is :  686.6
```


```
Following combination is finished:
Cell units = 100 | epochs = 100
Mean Squared Error for this combination is :  6953.91
```

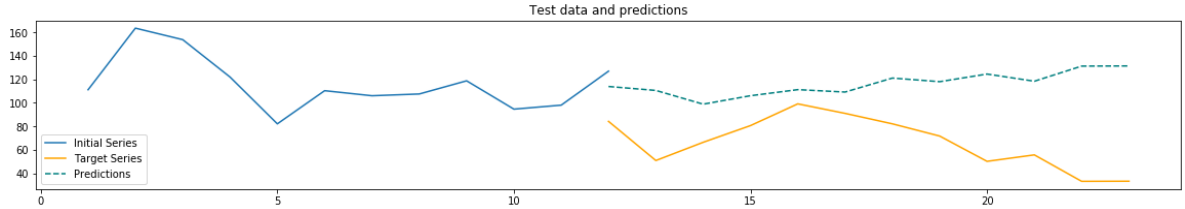

```
Following combination is finished:
Cell units = 20 | epochs = 1000
```

```
Mean Squared Error for this combination is :  42221.22
```


```
Following combination is finished:
Cell units = 50 | epochs = 1000
Mean Squared Error for this combination is :  651.14
```


```
Following combination is finished:
Cell units = 100 | epochs = 1000
Mean Squared Error for this combination is :  736.44
```


```
Following combination is finished:
Cell units = 20 | epochs = 5000
Mean Squared Error for this combination is :  29552.44
```




```
Following combination is finished:
Cell units = 50 | epochs = 5000
Mean Squared Error for this combination is :  6388.82
```


```
Following combination is finished:
Cell units = 100 | epochs = 5000
Mean Squared Error for this combination is :  42137.36
```

Test data and predictions

End of notebook