Air Quality dataset

I decided to use the same dataset that was used in the Deep Learning demo lab in this course. The dataset contains hourly data of air quality that was gathered in 5 Chinese cities. The dataset is available in UC Irvine's ML repository and can be accessed at https://archive.ics.uci.edu/ml/datasets/PM2.5+Data+of+Five+Chinese+Cities (https://archive.ics.uci.edu/ml/datasets/PM2.5+Data+of+Five+Chinese+Cities)

Main Objective

I repeated all the steps covered in the exercise to gain more understanding of how Deep Learning can be used for time series forecasting. The first section uses Recurrent Neural Networks and the next section uses LSTM (long-short-term-memory). This exercise is much simpler in complexity as compared to other Deep Learning networks. I used RNN and LSTM to forecast air pollution measurements by training the model using the measurement data avaiable in the datasets.

In [2]:
```
# Imports
import sys, os
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.simplefilter(action='ignore')
import seaborn as sns
os.chdir('data')
from colorsetup import colors, palette
plt.style.use('fivethirtyeight')
sns.set_palette(palette)
import pandas as pd
from datetime import datetime
import tensorflow as tf
import keras
from keras.models import Sequential
from keras.layers import Dense, SimpleRNN, LSTM, Activation, Dropout
import math
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
```

Dataset Summary

The first dataset that is used is for training is from Beijing and then it is filtered to only select 2015 data. There are 52584 records in Beijing's dataset and 18 columns.

In [7]:
```
df_Beijing = pd.read_csv('./FiveCitiesPM/Beijing.csv')
df_Beijing.shape
```

Out[7]: (52584, 18)

In [10]:
```
# Filerting for 2015 data
df_Beijing = df_Beijing[df_Beijing.year >= 2015]
df_Beijing.shape
```

Out[10]: (8760, 18)

After filtering, there are 8760 records for 2015.

Note that this an hourly time series data and each day has 24 hours of data i.e., 24 rows for each day.

In [11]:
```
df_Beijing.head(5)
```

Out[11]:

	No	year	month	day	hour	season	PM_Dongsi	PM_Dongsihuan	PM_Nongzhanguan	PM_US Post	D
43824	43825	2015	1	1	0	4	5.0	32.0	8.0	22.0	
43825	43826	2015	1	1	1	4	4.0	12.0	7.0	9.0	
43826	43827	2015	1	1	2	4	3.0	19.0	7.0	9.0	
43827	43828	2015	1	1	3	4	4.0	9.0	11.0	13.0	
43828	43829	2015	1	1	4	4	3.0	11.0	5.0	10.0	

Show list of columns and their data types:

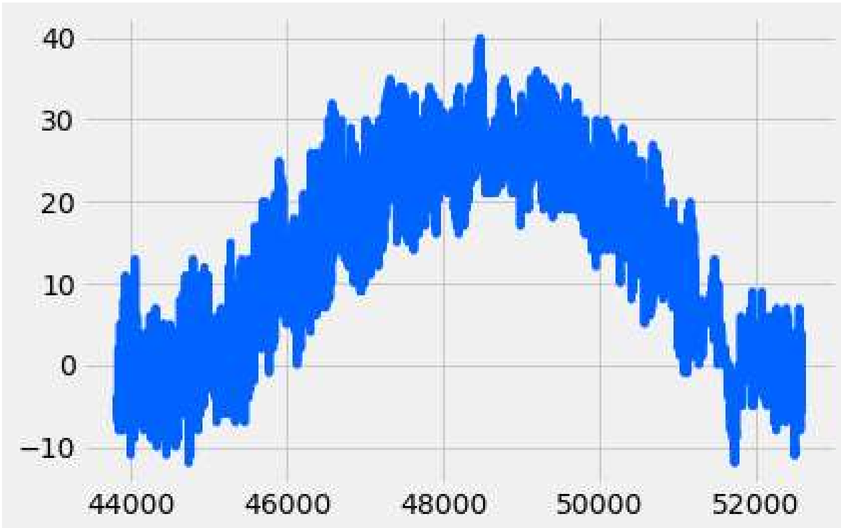
In [14]:
```
df_Beijing.dtypes
```

Out[14]:
```
No                     int64
year                   int64
month                  int64
day                    int64
hour                   int64
season                 int64
PM_Dongsi            float64
PM_Dongsihuan        float64
PM_Nongzhanguan      float64
PM_US Post           float64
DEWP                 float64
HUMI                 float64
PRES                 float64
TEMP                 float64
cbwd                  object
Iws                  float64
precipitation        float64
Iprec                float64
dtype: object
```

There are some missing values in the PM for pollution measurement column. The next step is to interpolate the missing values.

In [20]:
```python
plt.plot(df_Beijing['TEMP'])
```

Out[20]: [<matplotlib.lines.Line2D at 0x256e24790b8>]



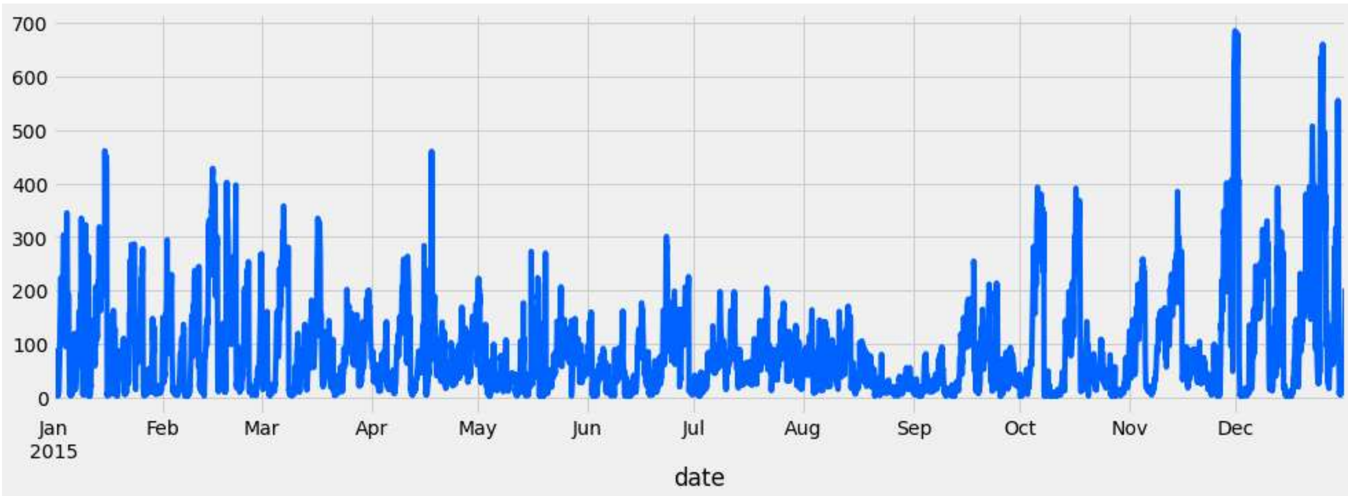
In [21]:
```python
df_Beijing['PM_Dongsi'] = df_Beijing['PM_Dongsi'].interpolate()
df_Beijing['TEMP'] = df_Beijing['TEMP'].interpolate()
df_Beijing['PM_Dongsi'].head(10)
```

Out[21]:
```
43824    5.0
43825    4.0
43826    3.0
43827    4.0
43828    3.0
43829    3.0
43830    3.0
43831    3.0
43832    4.0
43833    5.0
Name: PM_Dongsi, dtype: float64
```

In [22]:
```python
def make_date(row):
    return datetime(year = row['year'], month = row['month'], day = row['day'], hour = r
ow['hour'])
df_Beijing['date'] = df_Beijing.apply(make_date,axis=1)
df_Beijing.set_index(df_Beijing.date,inplace=True)
```

In [23]:
```python
#quick plot of full time series
plt.figure(figsize = (15,5))
df_Beijing['PM_Dongsi'].plot()
```

Out[23]: <matplotlib.axes._subplots.AxesSubplot at 0x256e28cdd68>



In [24]:
```python
df_Beijing['PM_Dongsi']
```

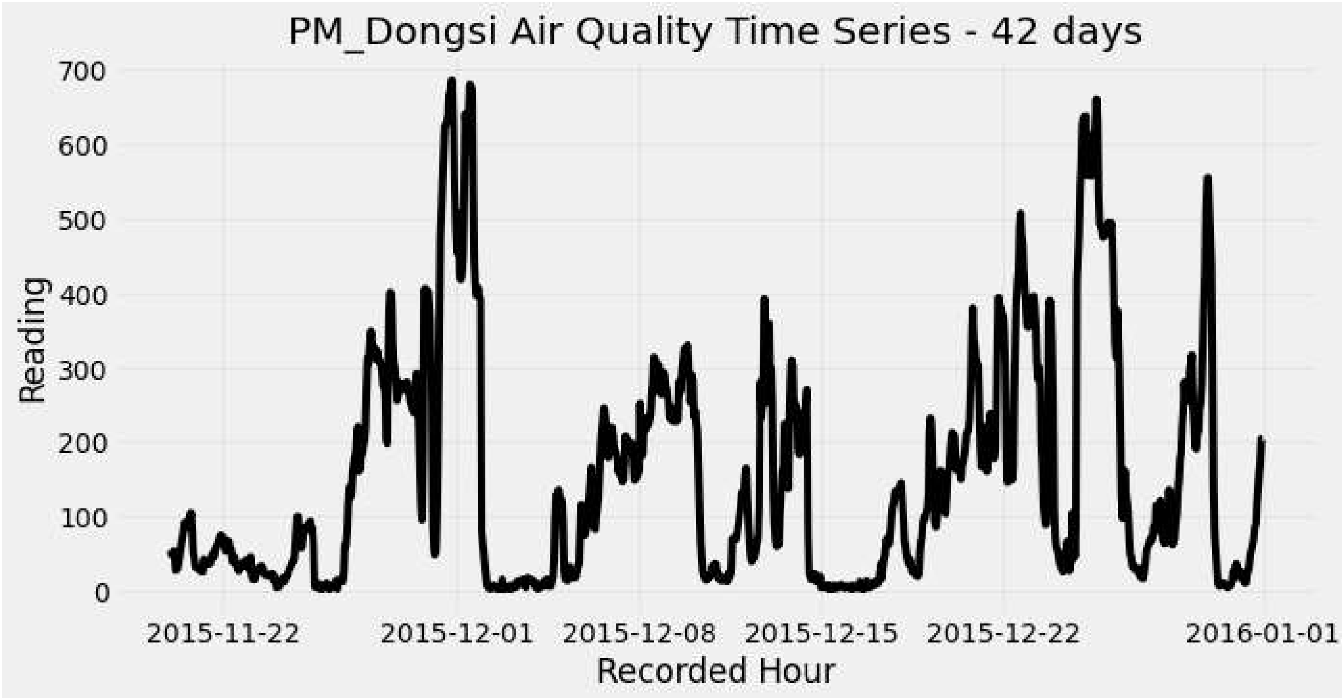
Out[24]:
```
date
2015-01-01 00:00:00      5.0
2015-01-01 01:00:00      4.0
2015-01-01 02:00:00      3.0
2015-01-01 03:00:00      4.0
2015-01-01 04:00:00      3.0
                        ...
2015-12-31 19:00:00    140.0
2015-12-31 20:00:00    157.0
2015-12-31 21:00:00    171.0
2015-12-31 22:00:00    204.0
2015-12-31 23:00:00    204.0
Name: PM_Dongsi, Length: 8760, dtype: float64
```

Generate a run-sequence plot before modeling the data.

In [26]:
```python
def get_n_last_days(df, series_name, n_days):
    """
    Extract last n_days of an hourly time series
    """

    return df[series_name][-(24*n_days):]

def plot_n_last_days(df, series_name, n_days):
    """
    Plot last n_days of an hourly time series
    """
    plt.figure(figsize = (10,5))
    plt.plot(get_n_last_days(df, series_name, n_days), 'k-')
    plt.title('{0} Air Quality Time Series - {1} days'
              .format(series_name, n_days))
    plt.xlabel('Recorded Hour')
    plt.ylabel('Reading')
    plt.grid(alpha=0.3)
```

Plot last 6 weeks of data (42 days) and notice the periodic component as well as autocorrelation structure.

```
In [27]: plot_n_last_days(df_Beijing, 'PM_Dongsi', 42)
```



Training Summary

Training a simple RNN to forecast the PM_Dongsi time series

First format the data in a 3D numpy array that can be processed by keras library.

A utility function is defined to split the data into test and training datasets. With this utility function, we will have the flexibility to define the length of the extracted training and test sequences and the number of time steps to use for prediction.

```
In [28]: df_Beijing.shape
Out[28]: (8760, 19)
```

In [29]:

```python
def get_keras_format_series(series):
    """
    Convert a series to a numpy array of shape
    [n_samples, time_steps, features]
    """

    series = np.array(series)
    return series.reshape(series.shape[0], series.shape[1], 1)

def get_train_test_data(df, series_name, series_days, input_hours,
                        test_hours, sample_gap=3):
    """
    Utility processing function that splits an hourly time series into
    train and test with keras-friendly format, according to user-specified
    choice of shape.

    arguments
    ---------
    df (dataframe): dataframe with time series columns
    series_name (string): column name in df
    series_days (int): total days to extract
    input_hours (int): length of sequence input to network
    test_hours (int): length of held-out terminal sequence
    sample_gap (int): step size between start of train sequences; default 5

    returns
    ---------
    tuple: train_X, test_X_init, train_y, test_y
    """

    forecast_series = get_n_last_days(df, series_name, series_days).values # reducing our forecast series to last n days

    train = forecast_series[:-test_hours] # training data is remaining days until amount of test_hours
    test = forecast_series[-test_hours:] # test data is the remaining test_hours

    train_X, train_y = [], []

    # range 0 through # of train samples - input_hours by sample_gap.
    # This is to create many samples with corresponding
    for i in range(0, train.shape[0]-input_hours, sample_gap):
        train_X.append(train[i:i+input_hours]) # each training sample is of length input hours
        train_y.append(train[i+input_hours]) # each y is just the next step after training sample

    train_X = get_keras_format_series(train_X) # format our new training set to keras format
    train_y = np.array(train_y) # make sure y is an array to work properly with keras

    # The set that we had held out for testing (must be same length as original train input)
    test_X_init = test[:input_hours]
    test_y = test[input_hours:] # test_y is remaining values from test set

    return train_X, test_X_init, train_y, test_y
```

Running the function on the last 56 days of data, and training the model that takes in 12 time steps in order to predict the next step.

In [30]:
```
series_days = 56
input_hours = 12
test_hours = 24

train_X, test_X_init, train_y, test_y = \
    (get_train_test_data(df_Beijing, 'PM_Dongsi', series_days,
                         input_hours, test_hours))
```

In [31]:
```
train_y.shape
```

Out[31]:
```
(436,)
```

There are 436 training samples of 12 time steps each.

In [34]:
```
print('Training input shape: {}'.format(train_X.shape))
print('Training output shape: {}'.format(train_y.shape))
print('Test input shape: {}'.format(test_X_init.shape))
print('Test output shape: {}'.format(test_y.shape))
```

```
Training input shape: (436, 12, 1)
Training output shape: (436,)
Test input shape: (12,)
Test output shape: (12,)
```

Fitting simple RNN.

In [35]:
```
def fit_SimpleRNN(train_X, train_y, cell_units, epochs):
    """
    Fit Simple RNN to data train_X, train_y

    arguments
    ---------
    train_X (array): input sequence samples for training
    train_y (list): next step in sequence targets
    cell_units (int): number of hidden units for RNN cells
    epochs (int): number of training epochs
    """

    # initialize model
    model = Sequential()

    # construct an RNN layer with specified number of hidden units
    # per cell and desired sequence input format
    model.add(SimpleRNN(cell_units, input_shape=(train_X.shape[1],1)))

    # add an output layer to make final predictions
    model.add(Dense(1))

    # define the loss function / optimization strategy, and fit
    # the model with the desired number of passes over the data (epochs)
    model.compile(loss='mean_squared_error', optimizer='adam')
    model.fit(train_X, train_y, epochs=epochs, batch_size=64, verbose=0)

    return model
```

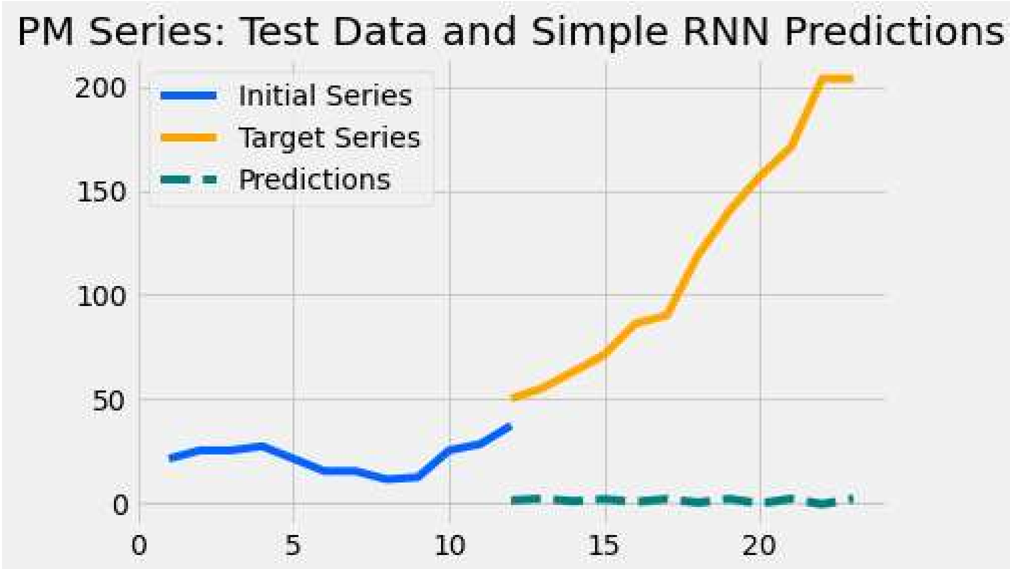
In [36]:
```
model = fit_SimpleRNN(train_X, train_y, cell_units=10, epochs=10)
```

The last step trained the model for only one future step. The next is to train for multiple steps by appending the output of one prediction to the input sequence and feed the new sequence to the model.

In [37]:
```
def predict(X_init, n_steps, model):
    """
    Given an input series matching the model's expected format,
    generates model's predictions for next n_steps in the series
    """

    X_init = X_init.copy().reshape(1,-1,1)
    preds = []

    # iteratively take current input sequence, generate next step pred,
    # and shift input sequence forward by a step (to end with latest pred).
    # collect preds as we go.
    for _ in range(n_steps):
        pred = model.predict(X_init)
        preds.append(pred)
        X_init[:,:-1,:] = X_init[:,1:,:] # replace first 11 values with 2nd through 12th
        X_init[:,-1,:] = pred # replace 12th value with prediction

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

    return preds

def predict_and_plot(X_init, y, model, title):
    """
    Given an input series matching the model's expected format,
    generates model's predictions for next n_steps in the series,
    and plots these predictions against the ground truth for those steps

    arguments
    ---------
    X_init (array): initial sequence, must match model's input shape
    y (array): true sequence values to predict, follow X_init
    model (keras.models.Sequential): trained neural network
    title (string): plot title
    """

    y_preds = predict(test_X_init, n_steps=len(y), model=model) # predict through length
of y
    # Below ranges are to set x-axes
    start_range = range(1, test_X_init.shape[0]+1) #starting at one through to length of
test_X_init to plot X_init
    predict_range = range(test_X_init.shape[0], test_hours)  #predict range is going to
 be from end of X_init to length of test_hours

    #using our ranges we plot X_init
    plt.plot(start_range, test_X_init)
    #and test and actual preds
    plt.plot(predict_range, test_y, color='orange')
    plt.plot(predict_range, y_preds, color='teal', linestyle='--')

    plt.title(title)
    plt.legend(['Initial Series','Target Series','Predictions'])
```

Predicting and plotting the baseline model.

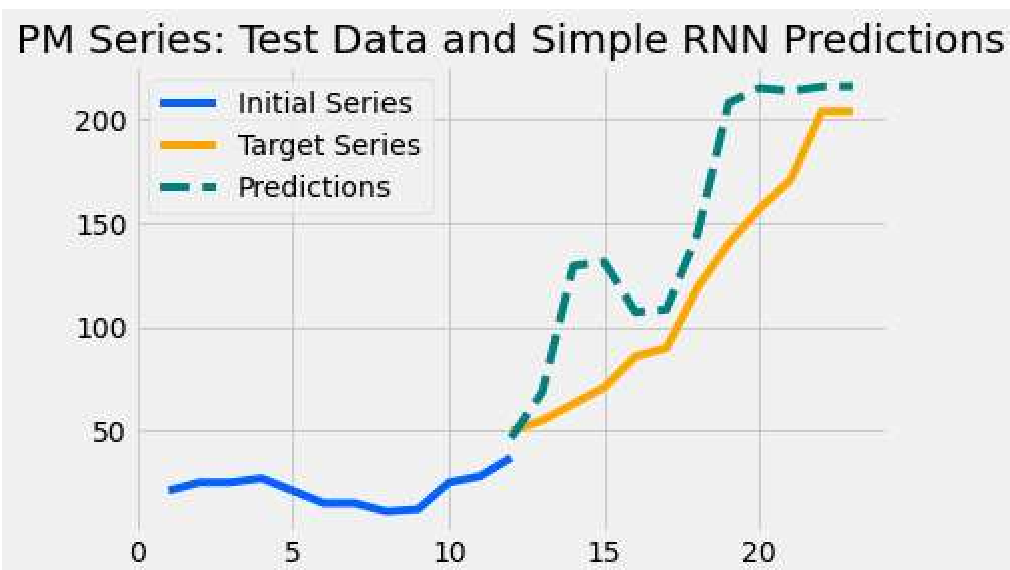
In [38]:
```
predict_and_plot(test_X_init, test_y, model,
                 'PM Series: Test Data and Simple RNN Predictions')
```



Based on the predictions shown in the graph, looks like the model is underfitting.

Training again by running for 1200 epochs. This will give the model more opportunity to learn.

In [39]:
```
model = fit_SimpleRNN(train_X, train_y, cell_units=30, epochs=1200)
predict_and_plot(test_X_init, test_y, model,
                 'PM Series: Test Data and Simple RNN Predictions')
```



The predictions are much better as compared to the last time we had after training the model.

Getting the structure of the model.

In [40]:
```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn_1 (SimpleRNN)     (None, 30)                960
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 31
=================================================================
Total params: 991
Trainable params: 991
Non-trainable params: 0
_________________________________________________________________
```

Even for this simple RNN model, we have about thousand of parameters to train.

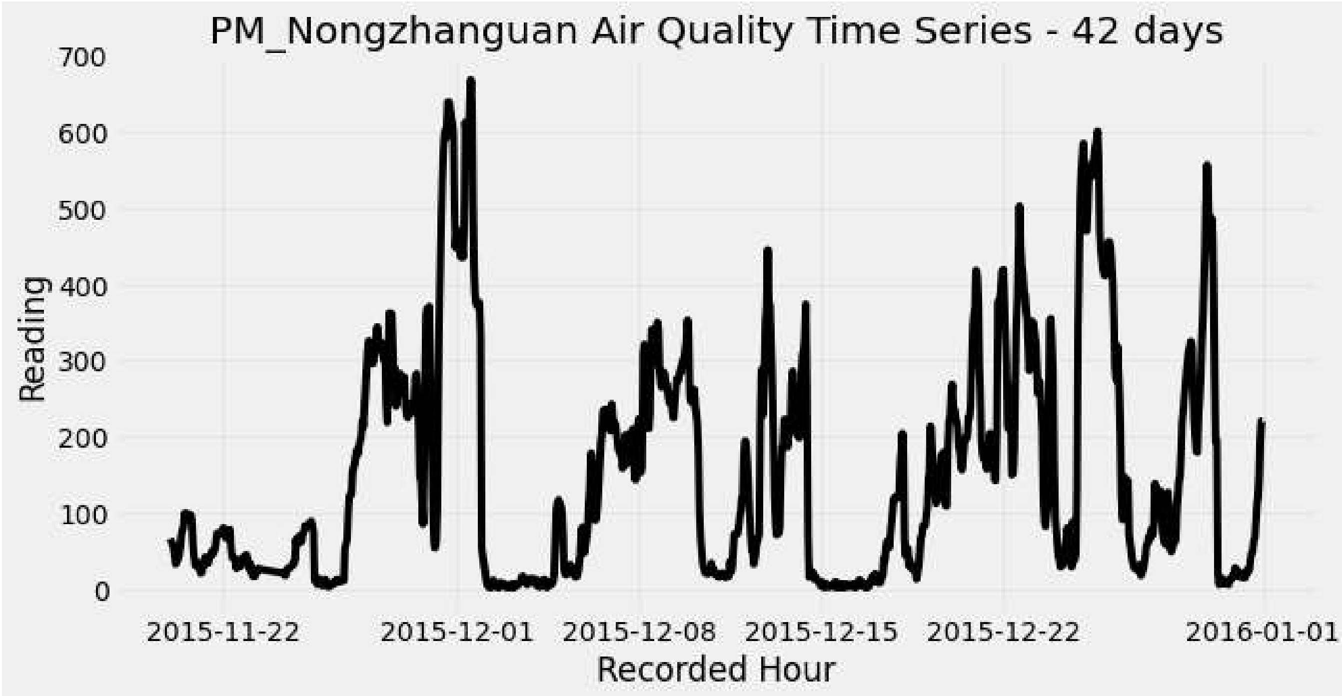
Training a simple RNN to forecast the PM_Nongzhanguan time series

Repeat all the steps for PM_Nongzhanguan as we did for PM_Dongsi.

In [41]:
```
df_Beijing['PM_Nongzhanguan'] = df_Beijing['PM_Nongzhanguan'].interpolate()
df_Beijing['PM_Nongzhanguan'].head(10)
```

Out[41]:
```
date
2015-01-01 00:00:00     8.0
2015-01-01 01:00:00     7.0
2015-01-01 02:00:00     7.0
2015-01-01 03:00:00    11.0
2015-01-01 04:00:00     5.0
2015-01-01 05:00:00     3.0
2015-01-01 06:00:00     6.0
2015-01-01 07:00:00     7.0
2015-01-01 08:00:00     9.0
2015-01-01 09:00:00    11.0
Name: PM_Nongzhanguan, dtype: float64
```

In [42]:
```
plot_n_last_days(df_Beijing, 'PM_Nongzhanguan', 42)
```

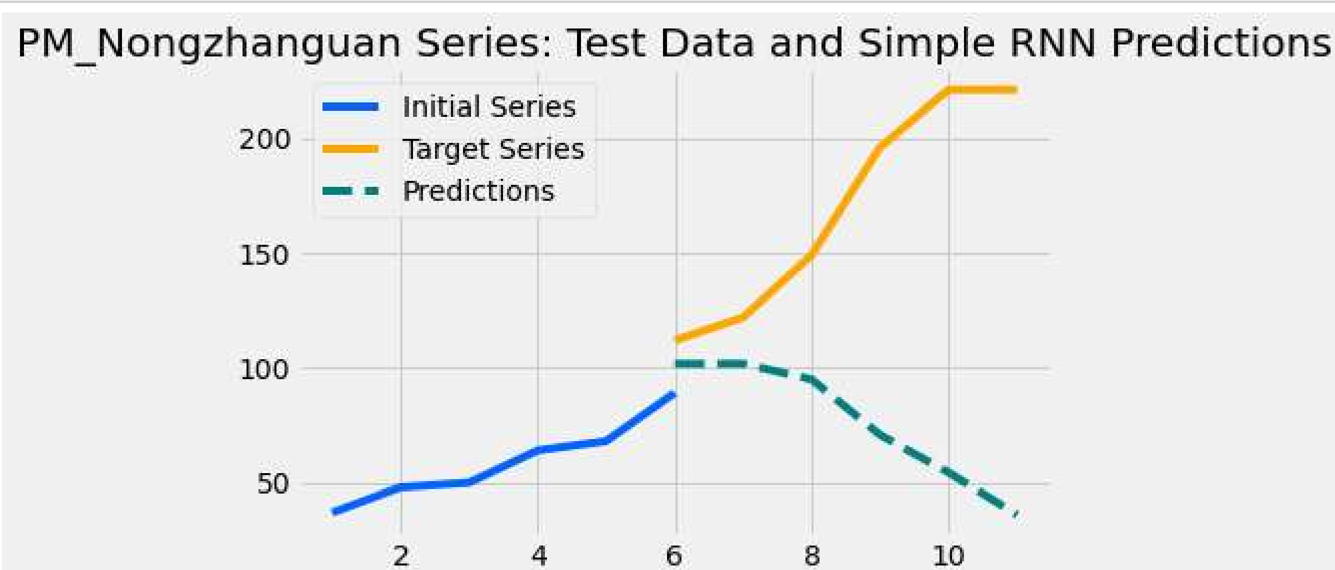


Training "PM_Nongzhanguan" series.

In [43]:
```
series_days = 56
input_hours = 6
test_hours = 12

train_X, test_X_init, train_y, test_y = \
    (get_train_test_data(df_Beijing, 'PM_Nongzhanguan', series_days,
                         input_hours, test_hours))
```

Predicting and plotting the model.

In [44]:
```
model = fit_SimpleRNN(train_X, train_y, cell_units=30, epochs=1200)

predict_and_plot(test_X_init, test_y, model,
                 'PM_Nongzhanguan Series: Test Data and Simple RNN Predictions')
```



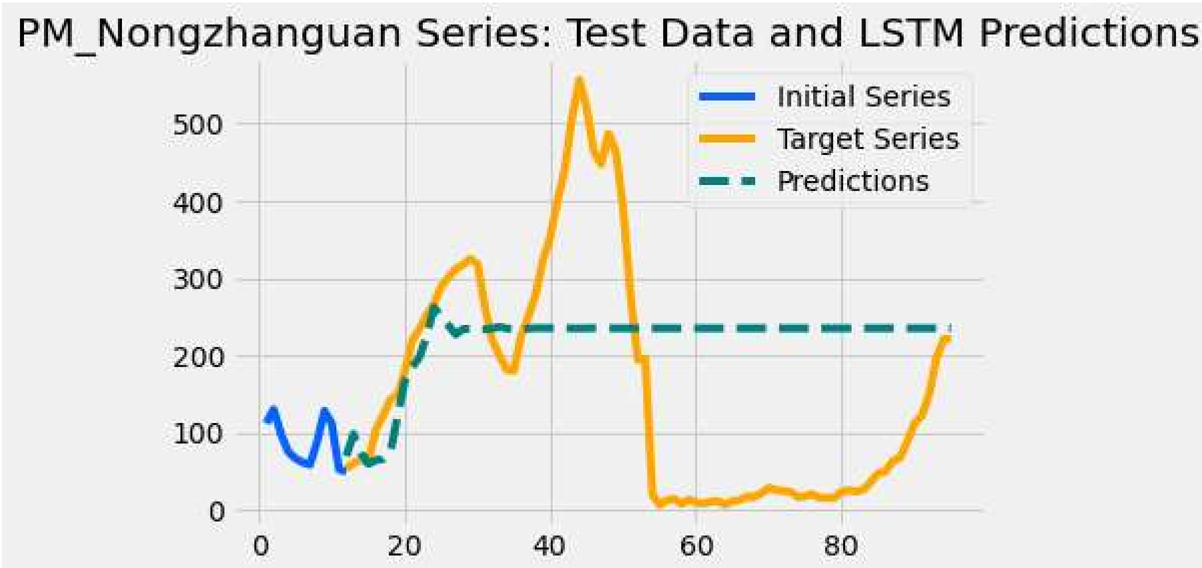
Training LSTM

We will use the model we have trained using RNN and apply LSTM on top of it.

In [45]:
```
def fit_LSTM(train_X, train_y, cell_units, epochs):
    """
    Fit LSTM to data train_X, train_y

    arguments
    ---------
    train_X (array): input sequence samples for training
    train_y (list): next step in sequence targets
    cell_units (int): number of hidden units for LSTM cells
    epochs (int): number of training epochs
    """

    # initialize model
    model = Sequential()

    # construct a LSTM layer with specified number of hidden units
    # per cell and desired sequence input format
    model.add(LSTM(cell_units, input_shape=(train_X.shape[1],1))) #,return_sequences= True))
    #model.add(LSTM(cell_units_l2, input_shape=(train_X.shape[1],1)))

    # add an output layer to make final predictions
    model.add(Dense(1))

    # define the loss function / optimization strategy, and fit
    # the model with the desired number of passes over the data (epochs)
    model.compile(loss='mean_squared_error', optimizer='adam')
    model.fit(train_X, train_y, epochs=epochs, batch_size=64, verbose=0)

    return model
```

Predicting and plotting LSTM model.

In [46]:
```
series_days = 50
input_hours = 12
test_hours = 96

train_X, test_X_init, train_y, test_y = \
    (get_train_test_data(df_Beijing, 'PM_Nongzhanguan', series_days,
                         input_hours, test_hours))

model = fit_LSTM(train_X, train_y, cell_units=70, epochs=3000)

predict_and_plot(test_X_init, test_y, model,
                 'PM_Nongzhanguan Series: Test Data and LSTM Predictions')
```



Showing the model summary. There are far more parameters to train as compared to RNN model.

In [48]:
```
model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm (LSTM)                  (None, 70)                20160
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 71
=================================================================
Total params: 20,231
Trainable params: 20,231
Non-trainable params: 0
_________________________________________________________________
```

Key Findings and Insights

The comparison of RNN and LSTM model summaries reflects that there are many more trainable parameters for LSTM and that's the reason why it took longer to train.

In case of LSTM, the model started to struggle at predicting at the end and became more conservative in its predictions.

Next Steps

Due to time constraint, the training times in this exercise were shorter. Deep Learning models usually take longer time to train themselves. The next steps would be to train the model for other series in the datasets, train by increasing cell_unit and running for more training epochs.

Some other steps that could be taken would be to cover all the topics discussed in the course and create a one python notebook to provide end to end solution for predicting diabetes using other Neural Networks. The dataset used for this exercise was also small. It would be much better to find a larger dataset.