Machine Learning Foundation

Time Series Deep Learning

Deep Learning for Time Series Forecasting

Project Introduction

This notebook explores the application of deep learning—specifically **Recurrent Neural Networks (RNNs)** and **Long Short-Term Memory networks (LSTMs)**—to time series forecasting.

Time series data is inherently sequential and often contains seasonality, autocorrelation, and non-stationarity. Traditional forecasting methods like ARIMA require manual feature engineering and assumptions about the data. Deep learning methods, on the other hand, can learn complex patterns directly from the data, enabling more flexible and automated forecasting pipelines.

Objective

The main objectives of this notebook are:

1. To introduce and implement RNNs and LSTMs for univariate time series forecasting.
2. To demonstrate how these models can learn temporal dependencies without explicit feature engineering.
3. To evaluate forecasting performance using RMSE and visualize prediction quality.

Dataset

We use hourly **PM2.5 air quality data from Beijing** in 2015, provided by the UCI Machine Learning Repository. The dataset is ideal for this task due to its consistent sampling frequency, long time range, and rich temporal dynamics.

Learning Outcomes

By the end of this notebook, you should be able to:

- Prepare time series data for deep learning models.
- Build and train an RNN or LSTM using Keras.

- Understand the core parameters that influence model performance.
- Evaluate model results using statistical metrics and visual inspection.

Tools and Libraries

This project makes use of:

- `TensorFlow / Keras` : for building and training deep learning models.
- `NumPy, Pandas` : for data handling and transformation.
- `Matplotlib, Seaborn` : for visualization.
- `Scikit-learn` : for preprocessing and performance evaluation.

In [ ]:

```
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
```

```
Using TensorFlow backend.
```

Section 1: Simple RNN

In this section, we will build a recurrent neural network and train it to forecast a single time series. We'll use a dataset provided by the UCI Machine Learning Repository that measures hourly air quality in Chinese cities/city districts[1].

1. Liang, X., S. Li, S. Zhang, H. Huang, and S. X. Chen (2016), PM2.5 data reliability, consistency, and air quality assessment in five Chinese cities, J. Geophys. Res. Atmos., 121, 10220â€“10236

Setting Up The Data

We'll start by working with Beijing data, and filter the dataset down to records from 2015.

In [ ]:

```
df_Beijing = pd.read_csv('./FiveCitiesPM/Beijing.csv')
df_Beijing = df_Beijing[df_Beijing.year >= 2015]
```

```
df_Beijing.head(10)
```

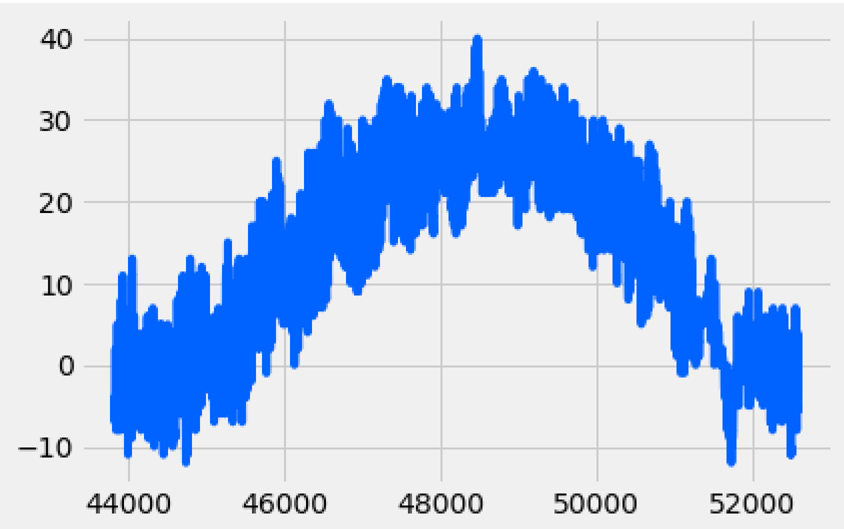
Out[ ]:

	No	year	month	day	hour	season	PM_Dongsi	PM_Dongsihuan	PM_Non
43824	43825	2015	1	1	0	4	5.0	32.0	
43825	43826	2015	1	1	1	4	4.0	12.0	
43826	43827	2015	1	1	2	4	3.0	19.0	
43827	43828	2015	1	1	3	4	4.0	9.0	
43828	43829	2015	1	1	4	4	3.0	11.0	
43829	43830	2015	1	1	5	4	3.0	18.0	
43830	43831	2015	1	1	6	4	3.0	20.0	
43831	43832	2015	1	1	7	4	3.0	22.0	
43832	43833	2015	1	1	8	4	NaN	NaN	
43833	43834	2015	1	1	9	4	5.0	37.0	

We are interested in attempting to forecast the 'PM' series, which are measurements of air pollution for several different districts. Note that there are occasional missing values in these series, which we can fill with simple linear interpolation. To start, we'll focus on the "PM_Dongsi" series and interpolate the missing values.

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

Out[ ]: [<matplotlib.lines.Line2D at 0x7fcf19fae110>]



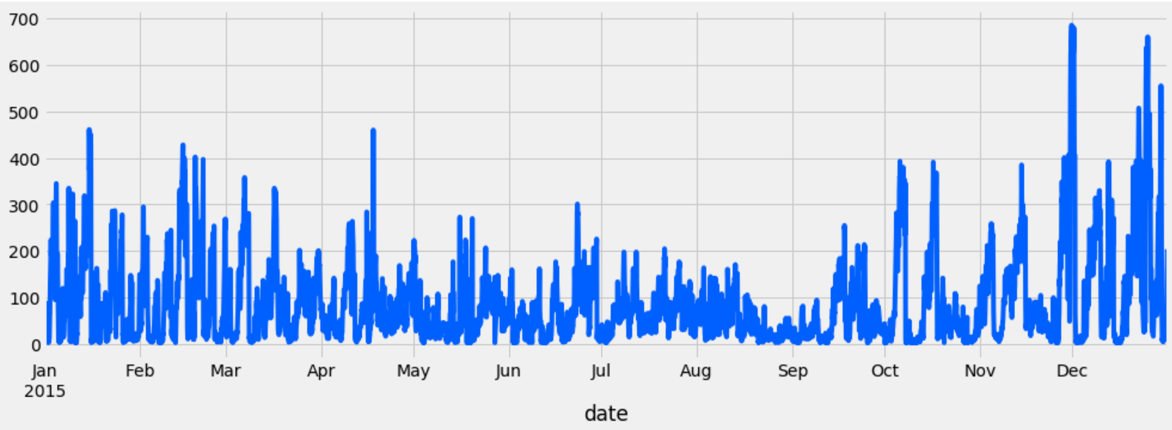
In [ ]:
```
df_Beijing['PM_Dongsi'] = df_Beijing['PM_Dongsi'].interpolate()
df_Beijing['TEMP'] = df_Beijing['TEMP'].interpolate()
df_Beijing['PM_Dongsi'].head(10)
```

```
Out[ ]: 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 [ ]: def make_date(row):
            return datetime(year = row['year'], month = row['month'], day = row['day'],
        df_Beijing['date'] = df_Beijing.apply(make_date,axis=1)
        df_Beijing.set_index(df_Beijing.date,inplace=True)
```

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

```
Out[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7fcf03ee9f10>
```




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

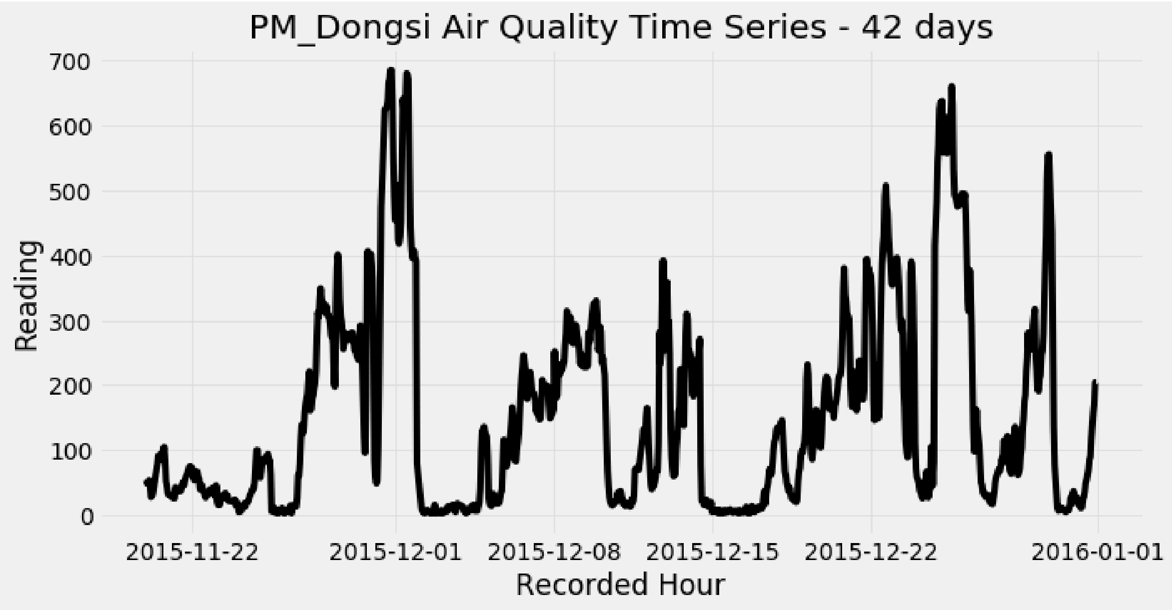
```
Out[ ]: date
        2015-01-01 00:00:00      5.0
        2015-01-01 01:00:00      4.0
        2015-01-01 02:00:00      3.0
        2015-01-01 03:00:00      4.0
        2015-01-01 04:00:00      3.0
                                ...
        2015-12-31 19:00:00    140.0
        2015-12-31 20:00:00    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
```

As usual, it's a good idea for us to generate a run-sequence plot before modeling the data. This way we can get a feel for what we're working with. We'll go ahead and define two utility functions that let us extract and plot the last n days of data (remember that this is an hourly time series, so each day has 24 time steps).

In [ ]:
```
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)
```

What do the last 6 weeks of data look like?

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



Review Question: what components that you've learned in previous lessons appear to be present in this time series?

Answer: There appears to be a periodic component as well as autocorrelation structure.

Example 1: Train a simple RNN to forecast the PM_Dongsi time series

Before we can train a neural network with keras, we need to process the data into a format that the library accepts. In particular, for keras RNNs and LSTMs, training samples should be stored in a 3D numpy array of shape **(n_samples, time_steps, n_features)**. Since we'll be using only the series' history to predict its future, we'll only have 1 feature. Also, for the next-step prediction that we'll do in this notebook, target values can be stored in a simple list.

To this end, we define utility functions that allow us to extract the formatted data. The **get_train_test_data** function gives us the flexibility to define the length of the extracted training and test sequences and the number of time steps to use for prediction -- we'll run simple tests of our models by holding out the end of the extracted sequence and generating predictions to compare against the ground truth.

Since our model will perform better with multiple training samples, we draw many slices from the entire training sequence, starting at different points in time. The gap between starting points of these slices is controlled by the **sample_gap** parameter.

In [ ]:
```
df_Beijing.shape
```

Out[ ]: (8760, 19)

In [ ]:
```
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 # rea

    train = forecast_series[:-test_hours] # training data is remaining days unti
    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 leng
        train_y.append(train[i+input_hours]) # each y is just the next step afte
```

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

    # The set that we had held out for testing (must be same length as original
    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
```

With the **get_train_test_data** utility function in hand, we're all set to extract keras-friendly arrays and start training simple RNN models. We run this function in the cell below. We use the last 56 days of the PM_Dongsi series, and will train a model that takes in 12 time steps in order to predict the next time step. We use the last day of data for visually testing the model.

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

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

Below we see that by taking multiple time slices, we get 436 training samples of 12 time steps each.

In [ ]:
```
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,)
```

And now we're ready to train! Since we'd like to repeatedly adjust our model's hyperparameters to see what works best, we'll write a reusable function for training a simple RNN model using keras. Take some time to understand what the keras syntax accomplishes at each step and how it relates to what we've learned about RNNs so far.

In [ ]:
```
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
```

Great, now let's use this function to fit a very simple baseline model.

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

```
WARNING:tensorflow:From /Applications/anaconda3/lib/python3.7/site-packages/kera
s/backend/tensorflow_backend.py:422: The name tf.global_variables is deprecated.
Please use tf.compat.v1.global_variables instead.
```

Not bad so far. But we need to work a bit harder to actually extract multi-step predictions from this model, as it was trained to predict only one future time step. For multi-step forecasting, we'll iteratively generate one prediction, append it to the end of the input sequence (and shift that sequence forward by one step), then feed the new sequence back to the model. We stop once we've generated all the time step predictions we need.

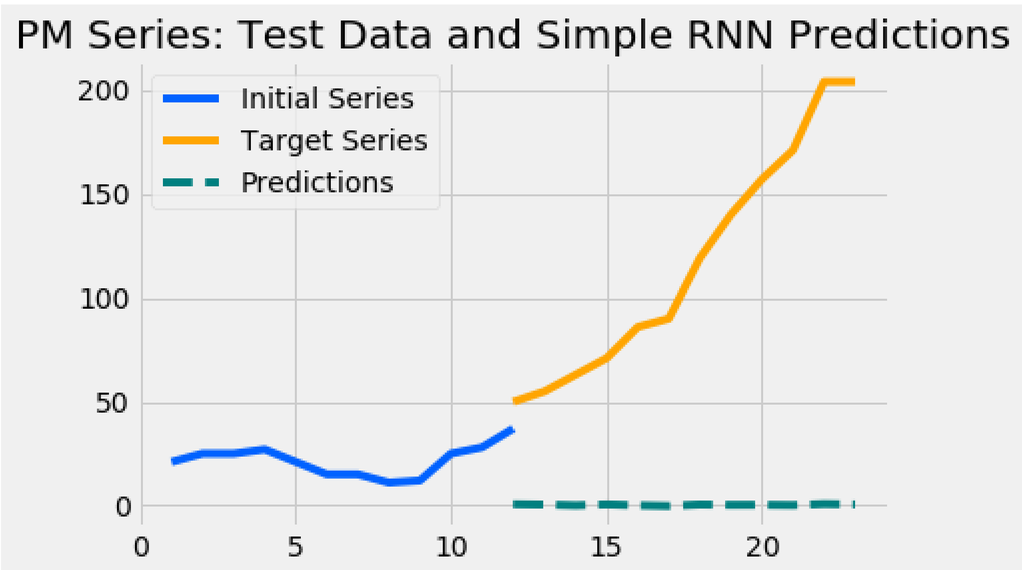
This prediction method and a utility function for plotting its output against the ground truth are defined below. Take some time time to familiarize yourself with the prediction method.

In [ ]:
```
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 thro
        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 throug
    # Below ranges are to set x-axes
    start_range = range(1, test_X_init.shape[0]+1) #starting at one through to l
    predict_range = range(test_X_init.shape[0], test_hours)  #predict range is g

    #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'])
```

Ok, we've finally arrived at the time to see how our baseline model does. We can simply run the **predict_and_plot** function on the extracted test data as below, and inspect the resulting plot.

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

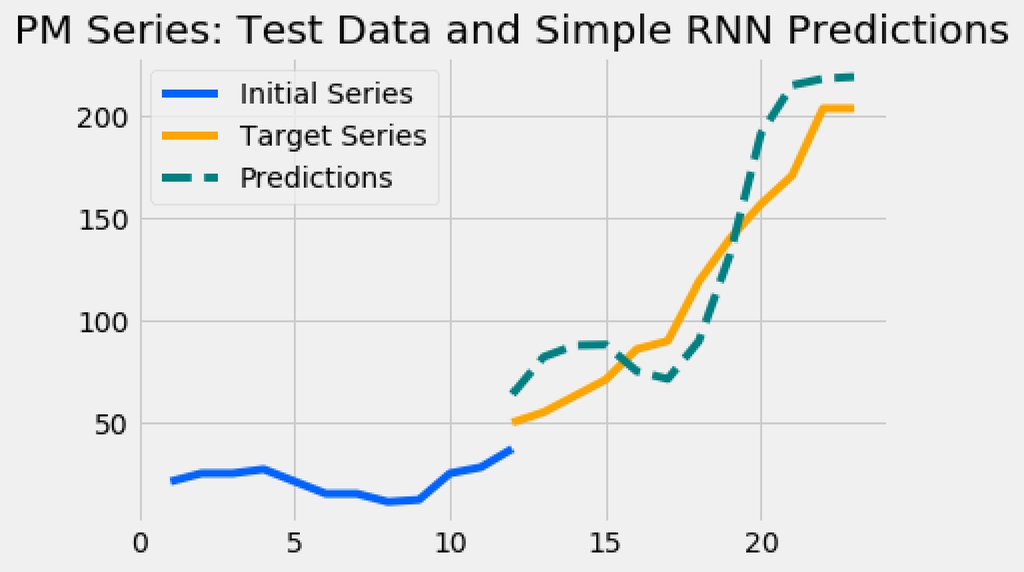


It looks like our model is badly underfit and essentially just making constant predictions. That's ok, it was a very simple baseline and trained very quickly.

We can improve by making the model more expressive, **increasing cell_units**. We can also pass over the training data many more times, **increasing epochs**, giving the model more opportunity to learn the patterns in the data. We'll try that below, it takes a longer time now since our training is more extensive.

Note that there is a significant amount of randomness in neural network training - we may need to retrain the model a few times in order to get results that we're happy with.

In [ ]:
```
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')
```



We can definitely get better results than before. Note that the model has the capacity to forecast an upward trend based on the trough pattern that occured recently (the input sequence).

Once we've created a model object, we can also get information about its structure and number of parameters by using the **summary** function. This is a useful way to measure the complexity of the model and get a feel for how long it may take to train.

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

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn_2 (SimpleRNN)     (None, 30)                960
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 31
=================================================================
Total params: 991
Trainable params: 991
Non-trainable params: 0
_________________________________________________________________
```

Note that even for this relatively simple model, we already have almost a thousand parameters to train. A larger number of cell units would increase the number of parameters - this is why the training process can become so time consuming.

Exercise 1: Train a simple RNN to forecast the PM_Nongzhanguan time series

We can reuse all of the functions we've defined so far in order to train models on different time series. In this exercise, you'll train your own model to forecast the "PM_Nongzhanguan" series from the Beijing dataframe.

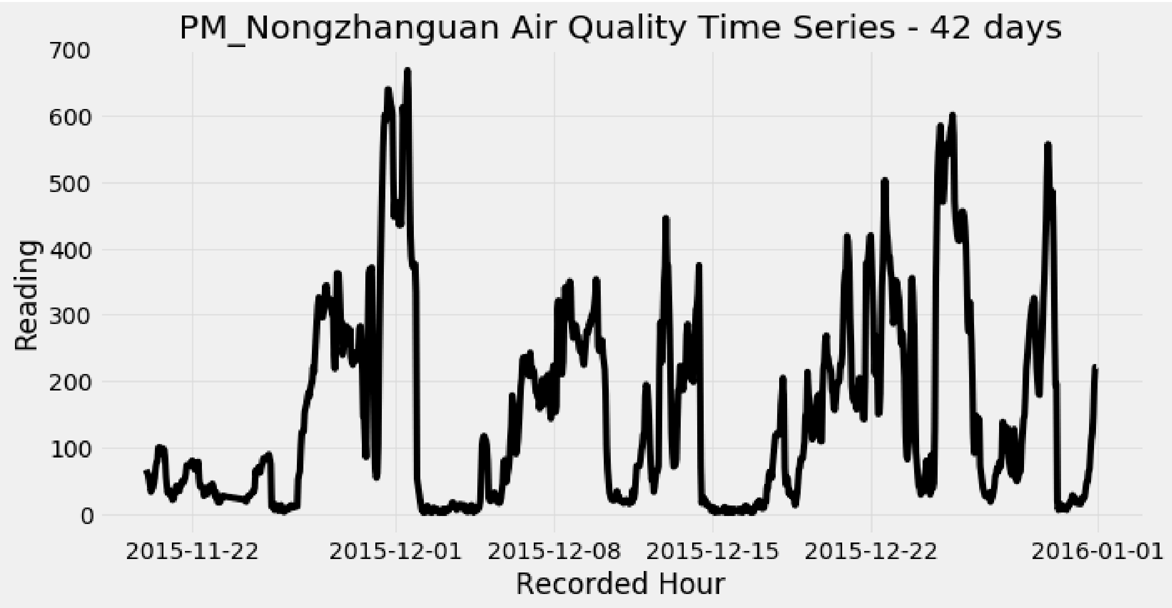
Step 1: Interpolate the missing values in the "PM_Nongzhanguan" series and plot the last 42 days of the series to get a feel for the data.

- We'll do this step together to get started

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

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



Step 2: Extract the train and test data for the "PM_Nongzhanguan" series using the function **get_train_test_data**. Use the following set of parameters:

- **series_days** : 56
- **input_hours** : 6
- **test_hours** : 12

For reference, below is how we called the function earlier on. You can also pull up the function's documentation to review the various arguments.

```
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 [ ]:
```
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))
```

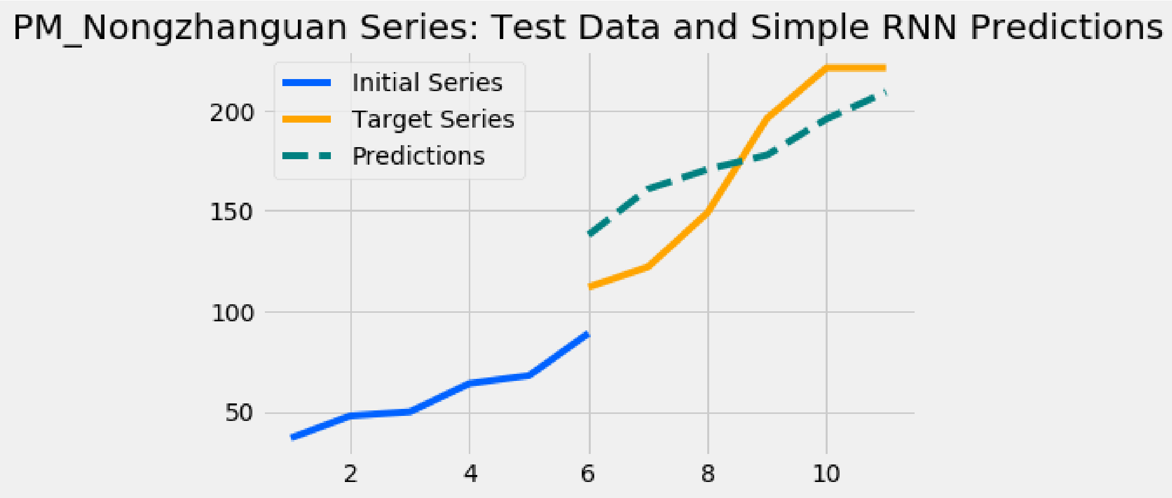
Step 3: Using the extracted train data to fit a simple RNN, and use the test data to generate and plot predictions.

- Start with a simple baseline -- few cell units and epochs. From here, try to make the model more expressive by increasing units and epochs until you're satisfied with the model's predictions.
- Be careful not to set units and/or epochs *too* high. The model may become very slow to train and also start to badly overfit the training data with the extra complexity you've added.

For reference, here's example code that you can adapt:

```
model = fit_SimpleRNN(train_X, train_y, cell_units=10,
epochs=10)

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

In [ ]:
```
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')
```



Again, we're able to do a decent job forecasting the continuation of an uptrend. We'll likely face more difficulty if we try to predict further into the future, especially with a simple RNN.

Section 2: LSTM

In this section, we'll build on our previous work by introducing LSTM models as an enhancement to the RNNs we've trained so far. Our first step will be to write a new function for fitting an LSTM with keras - notice that it's almost the same as our simple RNN function, with **LSTM** substitued for **SimpleRNN** (this is a nice display of how flexible keras is).

Take some time to review the logic of the function while we go ahead and run the example cell below (it will take a while).

In [ ]:
```
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_seque
    #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
```

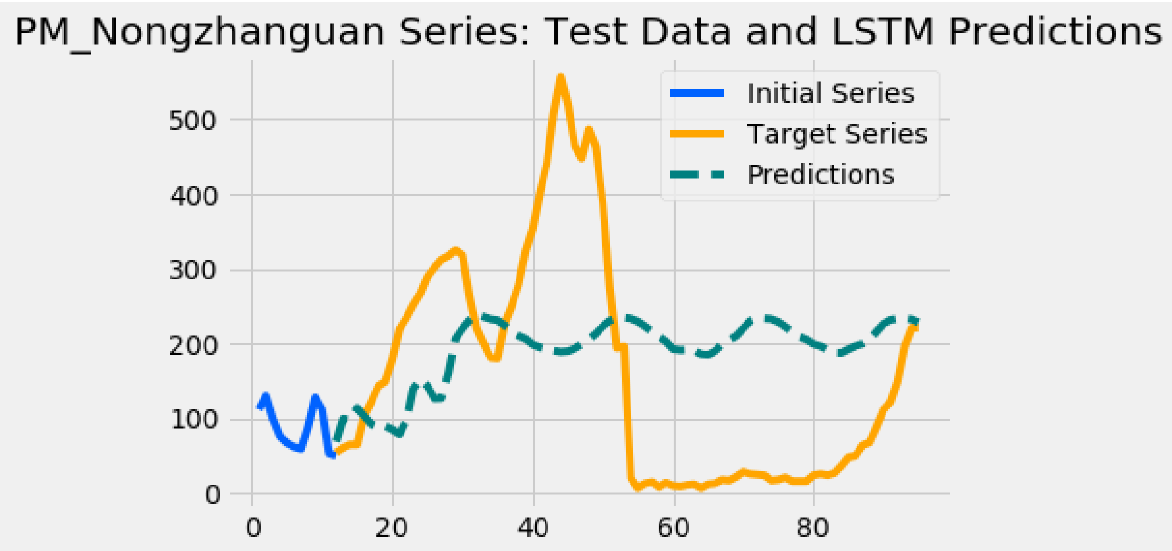
Example 2: Train a LSTM to forecast the PM_Nongzhanguan time series

With our new LSTM training function and all of our previously defined utility functions, adapting our code for LSTM forecasting will be fairly simple. We can extract the data as we did before, call the **fit_LSTM** function to build a model, and run the same *predict_and_plot* code.

Remember that one of the key benefits of LSTMs over simple RNNs is that they are better equipped to handle long input sequences and long-term dependencies. To see this evidence of this, we'll set *input_hours* to 12 and *test_hours* to 96 and see how our model predictions turn out with LSTM.

In [ ]:

```
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')
```



In our prediction plot we can start to see how LSTMs can be more expressive than simple RNNs - instead of just extrapolating a simple trend like our previous RNN models did, this LSTM model can effectively anticipate inflection points.

You should also notice that our model starts to struggle toward the end of the predicted sequence, becoming more conservative in its predictions. To improve the quality of forecasts over many time steps, we'd likely need to use more data and more sophisticated LSTM model structures that are beyond the scope of this lesson.

Take a look at the model summary and compare it with the summary for our simple RNN from example 1. You can see that there are many more trainable parameters for the LSTM, which explains why it took a much longer time for us to train this model.

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

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
lstm_2 (LSTM)                (None, 70)                20160     
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 71        
=================================================================
Total params: 20,231
Trainable params: 20,231
Non-trainable params: 0
_________________________________________________________________
```

Further Exploration

The simple models we've worked with are only the tip of the iceberg for deep learning. We've been time-limited for this exercise, and typical deep learning models involve much longer training times than what we're able to do in this notebook.

Here are several suggestions for how you could explore these ideas further, leveraging the code we've implemented today:

- Try using longer chunks of the series we've looked at in this notebook for modeling (set series_days larger than 56), or modeling other series in the dataset.
- When training with more data, try increasing cell_units and running more training epochs.
- Try using longer input sequences with LSTM, and predicting a wider range of test hours.

Conclusion

This notebook has provided a hands-on introduction to using deep learning for time series forecasting, with a focus on Recurrent Neural Networks (RNNs) and Long Short-Term Memory networks (LSTMs). Here's a summary of what was accomplished and key takeaways from the project:

Key Learnings

1. **Recurrent Neural Networks for Sequences**:
   We explored how RNNs maintain hidden states over time to model temporal

dependencies, making them naturally suited for sequence prediction tasks.

2. **LSTMs as an Improvement over RNNs**:
   LSTMs, with their gated architecture, were shown to overcome the vanishing gradient problem of traditional RNNs. They were more capable of modeling long-range dependencies and anticipating trend changes in the time series.

3. **Time Series Forecasting Process**:

   - Data preprocessing using normalization and reshaping into time windows
   - Sequence creation for supervised learning
   - Model definition and training using Keras
   - Prediction and inverse scaling for interpretation
   - Performance evaluation with RMSE and visualizations

4. **Model Behavior Insights**:

   - RNNs tend to extrapolate existing trends without understanding nuanced patterns.
   - LSTMs, on the other hand, show promise in identifying inflection points and maintaining pattern memory—though they may become conservative in long-term predictions.
   - Model performance is sensitive to input length, number of units, epochs, and the volume of data.

Recommendations for Further Exploration

- **Data Scaling**: Experiment with different scaling methods such as `StandardScaler` or log transformations to stabilize variance.
- **Longer Input Sequences**: Use longer historical sequences as input to provide the model more context.
- **Hyperparameter Tuning**: Use grid search or tools like Keras Tuner to optimize LSTM units, learning rate, dropout rate, etc.
- **Multivariate Forecasting**: Expand the model to use other variables (e.g., temperature, humidity) as additional features to improve accuracy.
- **Bidirectional or Stacked LSTMs**: Explore deeper or bidirectional architectures for complex series.
- **Stateful LSTM Training**: Consider stateful LSTM approaches to better preserve temporal continuity across batches.
- **Advanced Architectures**: Investigate attention mechanisms, Temporal Convolutional Networks (TCNs), or Transformer models for state-of-the-art performance.

Final Thoughts

While this project introduced relatively simple deep learning architectures, it has demonstrated how effective RNNs and LSTMs can be in modeling time series data. The key advantage lies in their ability to learn representations of sequential data without explicit feature engineering.

Forecasting accuracy and generalization improve significantly with careful tuning, more training data, and thoughtful architectural choices. As you scale these models, remember that model complexity must be balanced with interpretability and training efficiency.

> Deep learning offers a powerful alternative to traditional time series methods—but success depends on understanding both the data and the model behavior. This notebook lays a strong foundation for more advanced explorations in sequence modeling.