5
Stock Price Prediction with LSTM

In this chapter, you'll be introduced to how to predict a timeseries composed of real values. Specifically, we will predict the stock price of a large company listed on the NYSE stock exchange, given its historical performance.

In this chapter we will look at:

- How to collect the historical stock price information
- How to format the dataset for a timeseries prediction task
- How to use regression to predict the future prices of a stock
- Long short-term memory (LSTM) 101
- How LSTM will boost the predictive performance
- How to visualize the performance on the Tensorboard

Each of these bullet points is a section in this chapter. Moreover, to make the chapter visually and intuitively easier to understand, we will first apply each technique on a simpler signal: a cosine. A cosine is more deterministic than a stock price and will help with the understanding and the potentiality of the algorithm.

Note: we would like to point out that this project is just an experiment that works on the simple data we have available. Please don't use the code or the same model in a real-world scenario, since it may not perform at the same level. Remember: your capital is at risk, and there are no guarantees you'll always gain more.

Input datasets – cosine and stock price

As we claimed before, we will use two mono-dimensional signals as timeseries for our experiment. The first is a cosine wave with some added uniform noise.

This is the function to generate the cosine signal, given (as parameters) the number of points, the frequency of the signal, and the absolute intensity of the uniform generator for the noise. Also, in the body of the function, we're making sure to set the random seed, so we can make our experiments replicable:

```
def fetch_cosine_values(seq_len, frequency=0.01, noise=0.1):
    np.random.seed(101)
    x = np.arange(0.0, seq_len, 1.0)
    return np.cos(2 * np.pi * frequency * x) + np.random.uniform(low=-
noise, high=noise, size=seq_len)
```

To print 10 points, one full oscillation of the cosine (therefore `frequency` is `0.1`) with 0.1 magnitude noise, run:

```
print(fetch_cosine_values(10, frequency=0.1))
```

The output is:

```
[ 1.00327973 0.82315051 0.21471184 -0.37471266 -0.7719616 -0.93322063
-0.84762375 -0.23029438 0.35332577 0.74700479]
```

In our analysis, we will pretend this is a stock price, where each point of the timeseries is a mono-dimensional feature representing the price of the stock itself for that day.

The second signal, instead, comes from the real financial world. Financial data can be expensive and hard to extract, that's why in this experiment we use the Python library `quandl` to obtain such information. The library has been chosen since it's easy to use, cheap (XX free queries per day), and great for this exercise, where we want to predict only the closing price of the stock. If you're into automatic trading, you should look for more information, in the premium version of the library, or in some other libraries or data sources.

Quandl is an API, and the Python library is a wrapper over the APIs. To see what's returned, run the following command in your prompt:

```
$> curl "https://www.quandl.com/api/v3/datasets/WIKI/FB/data.csv"
Date,Open,High,Low,Close,Volume,Ex-Dividend,Split Ratio,Adj. Open,Adj.
High,Adj. Low,Adj. Close,Adj. Volume
2017-08-18,166.84,168.67,166.21,167.41,14933261.0,0.0,1.0,166.84,168.67,166
.21,167.41,14933261.0
```

```
2017-08-17,169.34,169.86,166.85,166.91,16791591.0,0.0,1.0,169.34,169.86,166
.85,166.91,16791591.0
2017-08-16,171.25,171.38,169.24,170.0,15580549.0,0.0,1.0,171.25,171.38,169.
24,170.0,15580549.0
2017-08-15,171.49,171.5,170.01,171.0,8621787.0,0.0,1.0,171.49,171.5,170.01,
171.0,8621787.0
...
```

The format is a CSV, and each line contains the date, the opening price, the highest and the lowest of the day, the closing, the adjusted, and some volumes. The lines are sorted from the most recent to the least. The column we're interested in is the `Adj. Close`, that is, the closing price after adjustments.

The adjusted closing price is a stock closing price after it has been amended to include any dividend, split, or merge.

Keep in mind that many online services show the unadjusted price or the opening price, therefore the numbers may not match.

Now, let's build a Python function to extract the adjusted price using the Python APIs. The full documentation of the APIs is available at `https://docs.quandl.com/v1.0/docs`, but we will just use the `quandl.get` function. Note that the default sorting is ascending, that is, from the oldest price to the newest one.

The function we're looking for should be able to cache calls and specify an initial and final timestamp to get the historical data beyond the symbol. Here's the code to do so:

```
def date_obj_to_str(date_obj):
    return date_obj.strftime('%Y-%m-%d')

def save_pickle(something, path):
    if not os.path.exists(os.path.dirname(path)):
        os.makedirs(os.path.dirname(path))
    with open(path, 'wb') as fh:
        pickle.dump(something, fh, pickle.DEFAULT_PROTOCOL)

def load_pickle(path):
    with open(path, 'rb') as fh:
    return pickle.load(fh)

def fetch_stock_price(symbol,
                      from_date,
                      to_date,
                      cache_path="./tmp/prices/"):
```

```
    assert(from_date <= to_date)
    filename = "{}_{}_{}.pk".format(symbol, str(from_date), str(to_date))
    price_filepath = os.path.join(cache_path, filename)
    try:
        prices = load_pickle(price_filepath)
        print("loaded from", price_filepath)
    except IOError:
        historic = quandl.get("WIKI/" + symbol,
        start_date=date_obj_to_str(from_date),
        end_date=date_obj_to_str(to_date))
        prices = historic["Adj. Close"].tolist()
        save_pickle(prices, price_filepath)
        print("saved into", price_filepath)
    return prices
```

The returned object of the function `fetch_stock_price` is a mono-dimensional array, containing the stock price for the requested symbol, ordered from the `from_date` to the `to_date`. Caching is done within the function, that is, if there's a cache miss, then the `quandl` API is called. The `date_obj_to_str` function is just a helper function, to convert `datetime.date` to the correct string format needed for the API.

Let's print the adjusted price of the Google stock price (whose symbol is GOOG) for January 2017:

```
import datetime
print(fetch_stock_price("GOOG",
      datetime.date(2017, 1, 1),
      datetime.date(2017, 1, 31)))
```

The output is:

```
[786.14, 786.9, 794.02, 806.15, 806.65, 804.79, 807.91, 806.36, 807.88,
804.61, 806.07, 802.175, 805.02, 819.31, 823.87, 835.67, 832.15, 823.31,
802.32, 796.79]
```

To have all the preceding functions available for all the scripts, we suggest you put them in a Python file, for example, in the code distributed within this book, they are in the `tools.py` file.

Format the dataset

Classic machine-learning algorithms are fed with multiple observations, where each of them has a pre-defined size (that is, the feature size). While working with timeseries, we don't have a pre-defined length: we want to create something that works for both 10 days look-back, but also for three years look-back. How is this possible?

It's very simple, instead of varying the number of features, we will change the number of observations, maintaining a constant feature size. Each observation represents a temporal window of the timeseries, and by sliding the window of one position on the right we create another observation. In code:

```
def format_dataset(values, temporal_features):
    feat_splits = [values[i:i + temporal_features] for i in
range(len(values) - temporal_features)]
    feats = np.vstack(feat_splits)
    labels = np.array(values[temporal_features:])
    return feats, labels
```

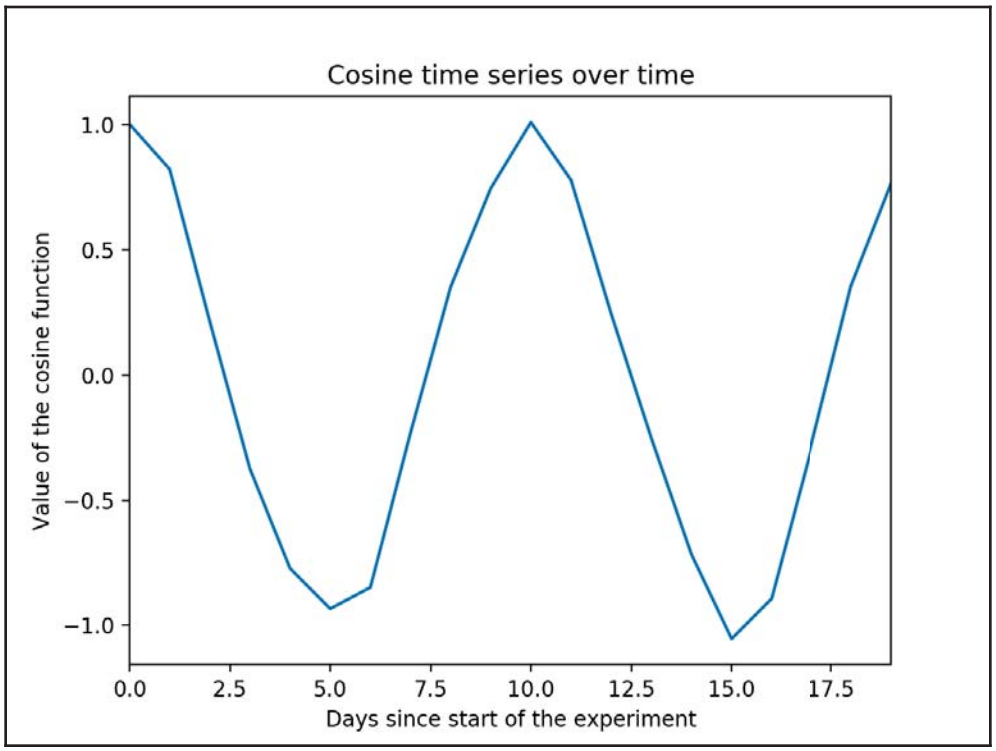
Given the timeseries, and the feature size, the function creates a sliding window which sweeps the timeseries, producing features and labels (that is, the value following the end of the sliding window, at each iteration). Finally, all the observations are piled up vertically, as well as the labels. The outcome is an observation with a defined number of columns, and a label vector.

We suggest putting this function in the `tools.py` file, so it can be accessed later.

Graphically, here's the outcome of the operation. Starting with the cosine signal, let's first plot a couple of oscillations of it, in another Python script (in the example, it's named `1_visualization_data.py`):

```
import datetime
import matplotlib.pyplot as plt
import numpy as np
import seaborn
from tools import fetch_cosine_values, fetch_stock_price, format_dataset
np.set_printoptions(precision=2)

cos_values = fetch_cosine_values(20, frequency=0.1)
seaborn.tsplot(cos_values)
plt.xlabel("Days since start of the experiment")
plt.ylabel("Value of the cosine function")
plt.title("Cosine time series over time")
plt.show()
```

The code is very simple; after a few imports, we plot a 20-point cosine timeseries with period 10 (that is frequency 0.01):



Let's now format the timeseries to be ingested by the machine learning algorithm, creating an observation matrix with five columns:

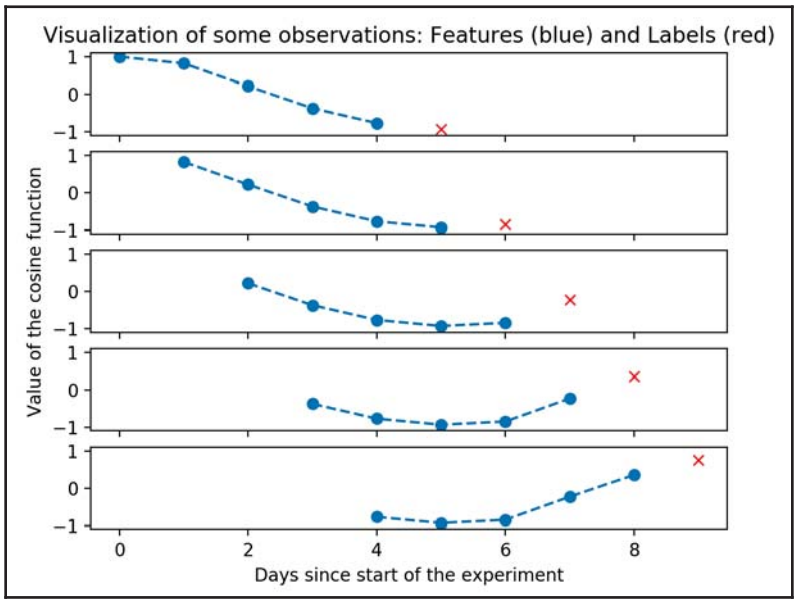
```
features_size = 5
minibatch_cos_X, minibatch_cos_y = format_dataset(cos_values,
features_size)
print("minibatch_cos_X.shape=", minibatch_cos_X.shape)
print("minibatch_cos_y.shape=", minibatch_cos_y.shape)
```

Starting from a timeseries with 20 points, the output will be an observation matrix of size *15x5*, while the label vector will be 15 elements long. Of course, by changing the feature size, the number of rows will also change.

Let's now visualize the operation, to make it simpler to understand. For example, let's plot the first five observations of the observation matrix. Let's also print the label of each feature (in red):

```
samples_to_plot = 5
f, axarr = plt.subplots(samples_to_plot, sharex=True)
for i in range(samples_to_plot):
    feats = minibatch_cos_X[i, :]
    label = minibatch_cos_y[i]
    print("Observation {}: X={} y={}".format(i, feats, label))
    plt.subplot(samples_to_plot, 1, i+1)
    axarr[i].plot(range(i, features_size + i), feats, '--o')
    axarr[i].plot([features_size + i], label, 'rx')
    axarr[i].set_ylim([-1.1, 1.1])
plt.xlabel("Days since start of the experiment")
axarr[2].set_ylabel("Value of the cosine function")
axarr[0].set_title("Visualization of some observations: Features (blue) and
Labels (red)")
plt.show()
```

And here's the plot:

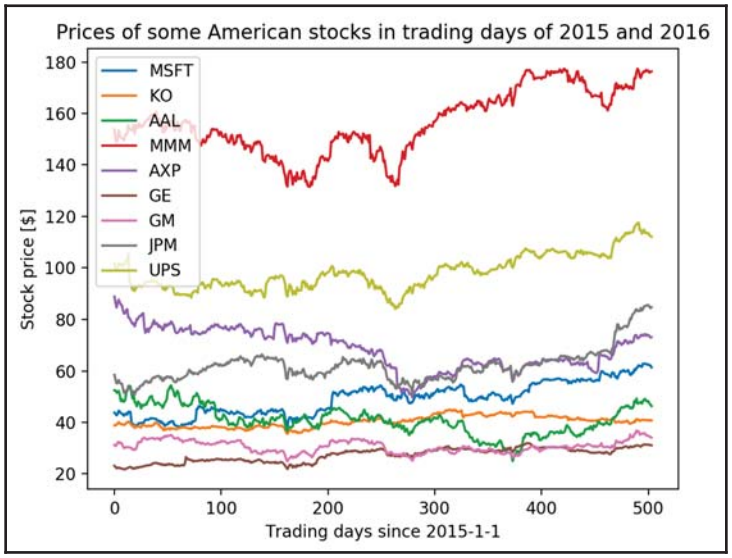


As you can see, the timeseries became an observation vector, each of them with size five.

So far, we haven't shown what the stock prices look like, therefore let's print them here as a timeseries. We selected (cherry-picked) some of the best-known companies in the United States; feel free to add your favorites to see the trend in the last year. In this plot, we'll just limit ourselves to two years: 2015 and 2016. We will also use the very same data in this chapter, therefore the next runs will have the timeseries cached:

```
symbols = ["MSFT", "KO", "AAL", "MMM", "AXP", "GE", "GM", "JPM", "UPS"]
ax = plt.subplot(1,1,1)
for sym in symbols:
    prices = fetch_stock_price(
    sym, datetime.date(2015, 1, 1), datetime.date(2016, 12, 31))
    ax.plot(range(len(prices)), prices, label=sym)

handles, labels = ax.get_legend_handles_labels()
ax.legend(handles, labels)
plt.xlabel("Trading days since 2015-1-1")
plt.ylabel("Stock price [$]")
plt.title("Prices of some American stocks in trading days of 2015 and
2016")
plt.show()
```

And this is the plot of the prices:



Each of the lines is a timeseries, and as we did for the cosine signal, in this chapter it will be transformed into an observation matrix (with the `format_dataset` function).

Are you excited? The data is ready, now let's move on to the interesting data science part of the project.

Using regression to predict the future prices of a stock

Given the observation matrix and a real value label, we are initially tempted to approach the problem as a regression problem. In this case, the regression is very simple: from a numerical vector, we want to predict a numerical value. That's not ideal. Treating the problem as a regression problem, we force the algorithm to think that each feature is independent, while instead, they're correlated, since they're windows of the same timeseries. Let's start anyway with this simple assumption (each feature is independent), and we will show in the next chapter how performance can be increased by exploiting the temporal correlation.

In order to evaluate the model, we now create a function that, given the observation matrix, the true labels, and the predicted ones, will output the metrics (in terms of **mean square error** (**MSE**) and **mean absolute error** (**MAE**) of the predictions. It will also plot the training, testing, and predicted timeseries one onto another, to visually check the performance. In order to compare the results, we also include the metrics in case we don't do use any model, but we simply predict the day-after value as the value of the present day (in the stock market, this means that we will predict the price for tomorrow as the price the stock has today).

Before that, we need a helping function to reshape matrices to mono-dimensional (1D) arrays. Please keep this function in the `tools.py` file, since it will be used by multiple scripts:

```
def matrix_to_array(m):
    return np.asarray(m).reshape(-1)
```

Now, time for the evaluation function. We decided to put this function into the `evaluate_ts.py` file, so many other scripts can access it:

```
import numpy as np
from matplotlib import pylab as plt
from tools import matrix_to_array
```

```
def evaluate_ts(features, y_true, y_pred):
    print("Evaluation of the predictions:")
    print("MSE:", np.mean(np.square(y_true - y_pred)))
    print("mae:", np.mean(np.abs(y_true - y_pred)))

    print("Benchmark: if prediction == last feature")
    print("MSE:", np.mean(np.square(features[:, -1] - y_true)))
    print("mae:", np.mean(np.abs(features[:, -1] - y_true)))

    plt.plot(matrix_to_array(y_true), 'b')
    plt.plot(matrix_to_array(y_pred), 'r--')
    plt.xlabel("Days")
    plt.ylabel("Predicted and true values")
    plt.title("Predicted (Red) VS Real (Blue)")
    plt.show()

    error = np.abs(matrix_to_array(y_pred) - matrix_to_array(y_true))
    plt.plot(error, 'r')
    fit = np.polyfit(range(len(error)), error, deg=1)
    plt.plot(fit[0] * range(len(error)) + fit[1], '--')
    plt.xlabel("Days")
    plt.ylabel("Prediction error L1 norm")
    plt.title("Prediction error (absolute) and trendline")
    plt.show()
```

Now, time to move to the modeling phase.

As previously, we start first with the cosine signal and then we move to the stock price prediction.

We also suggest you put the following code in another file, for example, in `2_regression_cosine.py` (you can find the code in the code bundle under this name).

Let's start with some imports and with the seed for `numpy` and `tensorflow`:

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from evaluate_ts import evaluate_ts
from tensorflow.contrib import rnn
from tools import fetch_cosine_values, format_dataset

tf.reset_default_graph()
tf.set_random_seed(101)
```

Then, it's time to create the cosine signal and to transform it into an observation matrix. In this example, we will use 20 as feature size, since it's roughly the equivalent number of working days in a month. The regression problem has now shaped this way: given the 20 values of the cosine in the past, forecast the next day value.

As training and testing, we will use datasets of `250` observation each, to have the equivalent of one year of data (one year contains just under `250` working days). In this example, we will generate just one cosine signal, and then it will be broken into two pieces: the first half will contain the train data, and the second half the testing. Feel free to change them, and observe how the performance changes when these parameters are changed:

```
feat_dimension = 20
train_size = 250
test_size = 250
```

1. Now, in this part of the script, we will define some parameters for Tensorflow. More specifically: the learning rate, the type of optimizer to use, and the number of `epoch` (that is, how many times the training dataset goes into the learner during the training operation). These values are not the best, feel free to change them to predict some better ones:

```
learning_rate = 0.01
optimizer = tf.train.AdamOptimizer
n_epochs = 10
```

2. Finally, it's time to prepare the observation matrices, for training and testing. Keep in mind that to speed up the Tensorflow analysis, we will use `float32` (4 bytes long) in our analysis:

```
cos_values = fetch_cosine_values(train_size + test_size + feat_dimension)
minibatch_cos_X, minibatch_cos_y = format_dataset(cos_values,
feat_dimension)
train_X = minibatch_cos_X[:train_size, :].astype(np.float32)
train_y = minibatch_cos_y[:train_size].reshape((-1, 1)).astype(np.float32)
test_X = minibatch_cos_X[train_size:, :].astype(np.float32)
test_y = minibatch_cos_y[train_size:].reshape((-1, 1)).astype(np.float32)
```

Given the datasets, let's now define the placeholders for the observation matrix and the labels. Since we're building a generic script, we just set the number of features, and not the number of observations:

```
X_tf = tf.placeholder("float", shape=(None, feat_dimension), name="X")
y_tf = tf.placeholder("float", shape=(None, 1), name="y")
```

Here's the core of our project: the regression algorithm implemented in Tensorflow.

1. We opted for the most classic way of implementing it, that is, the multiplication between the observation matrix with a weights array plus the bias. What's coming out (and the returned value of this function) is the array containing the predictions for all the observations contained in `x`:

```
def regression_ANN(x, weights, biases):
    return tf.add(biases, tf.matmul(x, weights))
```

2. Now, let's define the trainable parameters of the regressor, which are the `tensorflow` variables. The weights are a vector with as many values as the feature size, while the bias is just a scalar.

Note that we initialized the weights using a truncated normal distribution, to have values close to zero, but not too extreme (as a plain normal distribution could output); for the bias we instead set it to zero.

Again, feel free to change the initializations, to see the changes in performance:

```
weights = tf.Variable(tf.truncated_normal([feat_dimension, 1], mean=0.0,
stddev=1.0), name="weights")
biases = tf.Variable(tf.zeros([1, 1]), name="bias")
```

3. The last thing we need to define in the `tensorflow` graphs are how the predictions are calculated (in our case, it's simply the output of the function which defines the model), the cost (in the example we use the MSE), and the training operator (we want to minimize the MSE, using the optimizer with the learning rate set previously):

```
y_pred = regression_ANN(X_tf, weights, biases)
cost = tf.reduce_mean(tf.square(y_tf - y_pred))
train_op = optimizer(learning_rate).minimize(cost)
```

We're now ready to open a `tensorflow` session, and train the model.

4. We will first initialize the variables, then, in a loop, we will feed the `training` dataset into the `tensorflow` graph (using the placeholders). At each iteration, we will print the training MSE:

```
with tf.Session() as sess:
   sess.run(tf.global_variables_initializer())
   # For each epoch, the whole training set is feeded into the tensorflow
graph

   for i in range(n_epochs):
       train_cost, _ = sess.run([cost, train_op], feed_dict={X_tf: train_X,
y_tf: train_y})
       print("Training iteration", i, "MSE", train_cost)

   # After the training, let's check the performance on the test set
   test_cost, y_pr = sess.run([cost, y_pred], feed_dict={X_tf: test_X,
y_tf: test_y})
   print("Test dataset:", test_cost)

   # Evaluate the results
   evaluate_ts(test_X, test_y, y_pr)

   # How does the predicted look like?
   plt.plot(range(len(cos_values)), cos_values, 'b')
   plt.plot(range(len(cos_values)-test_size, len(cos_values)), y_pr, 'r--')
   plt.xlabel("Days")
   plt.ylabel("Predicted and true values")
   plt.title("Predicted (Red) VS Real (Blue)")
   plt.show()
```

After the training, we evaluated the MSE on the testing dataset, and finally, we printed and plotted the performance of the model.

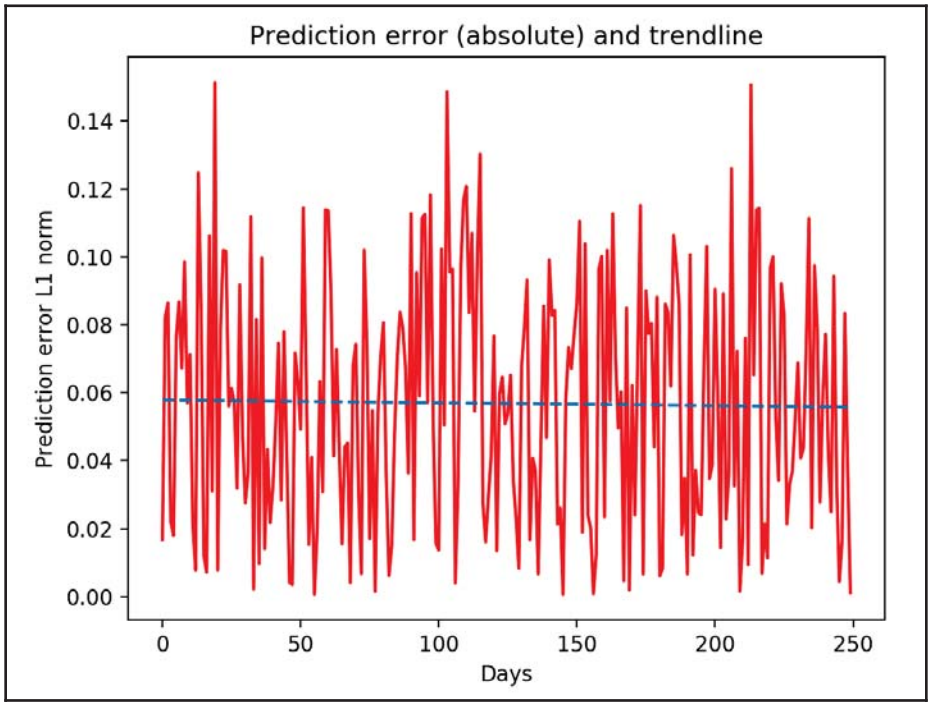
With the default values we provided in the scripts, performances are worse than the non-modeling performance. With some tuning, the results improve. For example, by setting the learning rate equal to 0.1 and the number of training epoch to *1000*, the output of the script will be similar to this:

```
Training iteration 0 MSE 4.39424
Training iteration 1 MSE 1.34261
Training iteration 2 MSE 1.28591
Training iteration 3 MSE 1.84253
Training iteration 4 MSE 1.66169
Training iteration 5 MSE 0.993168
...
...
Training iteration 998 MSE 0.00363447
```

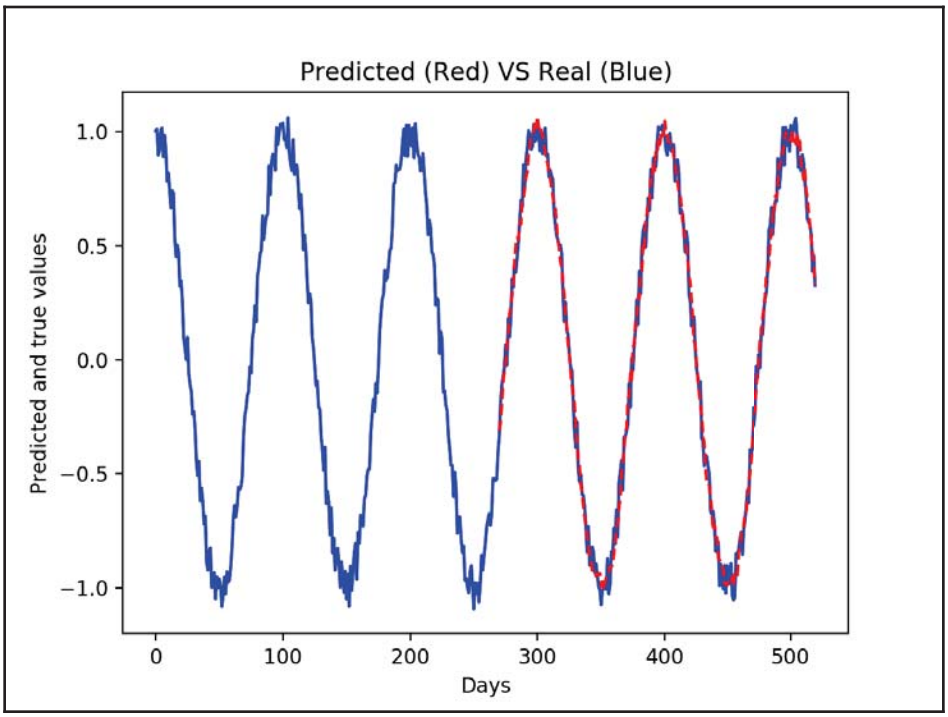
```
Training iteration 999 MSE 0.00363426
Test dataset: 0.00454513
Evaluation of the predictions:
MSE: 0.00454513
mae: 0.0568501
Benchmark: if prediction == last feature
MSE: 0.964302
mae: 0.793475
```

Training performance and testing performance are very similar (therefore we're not overfitting the model), and both the MSE and the MAE are better than a no-modeling prediction.

That's how the error looks for each timepoint. It seems that it's contained between +/-0.15, and doesn't have any trend over time. Remember that the noise we artificially introduced with the cosine had a magnitude of +/- 0.1 with a uniform distribution:



Finally, the last graph shows both the training timeseries overlapped with the predicted one. Not bad for a simple linear regression, right?



Let's now apply the same model on a stock price. We suggest you copy the content of the current file to a new one, named `3_regression_stock_price.py`. Here we will change only the data importing bit, leaving everything else as it is.

Let's use the Microsoft stock price in this example, whose symbol is `"MSFT"`. It's simple to load the prices for this symbol for 2015/16 and format them as an observation matrix. Here's the code, also containing the casting to float32 and the train/test split. In this example, we have one year of training data (2015) which will be used to predict the stock price for the whole of 2016:

```
symbol = "MSFT"
feat_dimension = 20
train_size = 252
test_size = 252 - feat_dimension

# Settings for tensorflow
learning_rate = 0.05
```

```
optimizer = tf.train.AdamOptimizer
n_epochs = 1000

# Fetch the values, and prepare the train/test split
stock_values = fetch_stock_price(symbol, datetime.date(2015, 1, 1),
datetime.date(2016, 12, 31))
minibatch_cos_X, minibatch_cos_y = format_dataset(stock_values,
feat_dimension)
train_X = minibatch_cos_X[:train_size, :].astype(np.float32)
train_y = minibatch_cos_y[:train_size].reshape((-1, 1)).astype(np.float32)
test_X = minibatch_cos_X[train_size:, :].astype(np.float32)
test_y = minibatch_cos_y[train_size:].reshape((-1, 1)).astype(np.float32)
```

In this script, we found that the best performances have been obtained with the following settings:

```
learning_rate = 0.5
n_epochs = 20000
optimizer = tf.train.AdamOptimizer
```

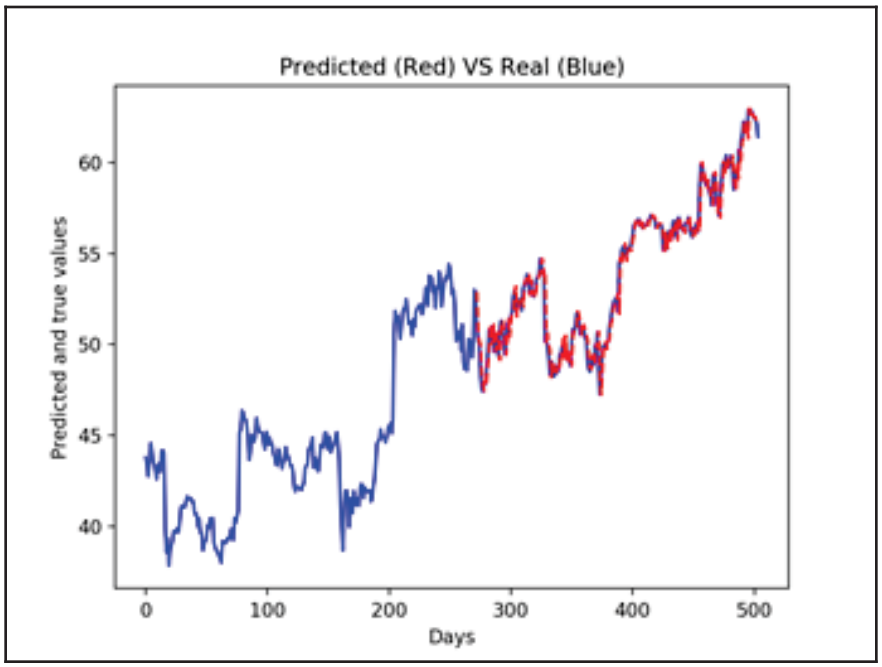
The output of the script should look like this:

```
Training iteration 0 MSE 15136.7
Training iteration 1 MSE 106385.0
Training iteration 2 MSE 14307.3
Training iteration 3 MSE 15565.6
...
...
Training iteration 19998 MSE 0.577189
Training iteration 19999 MSE 0.57704
Test dataset: 0.539493
Evaluation of the predictions:
MSE: 0.539493
mae: 0.518984
Benchmark: if prediction == last feature
MSE: 33.7714
mae: 4.6968
```

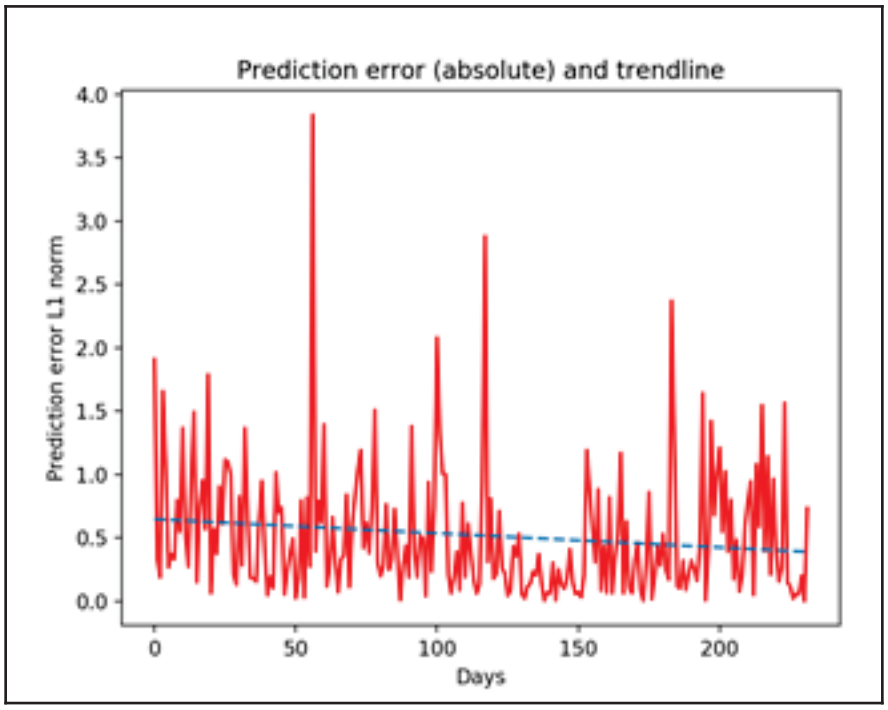
Even in this case, we're not overfitting, and the simple regressor performs better than no model at all (we all would bet that). At the beginning, the cost is really high, but iteration after iteration, it gets very close to zero. Also, the `mae` score is easy to interpret in this case, they are dollars! With a learner, we would have predicted on average half a dollar closer to the real price the day after; without any learner, nine times more.

Let's now visually evaluate the performance of the model, impressive isn't it?

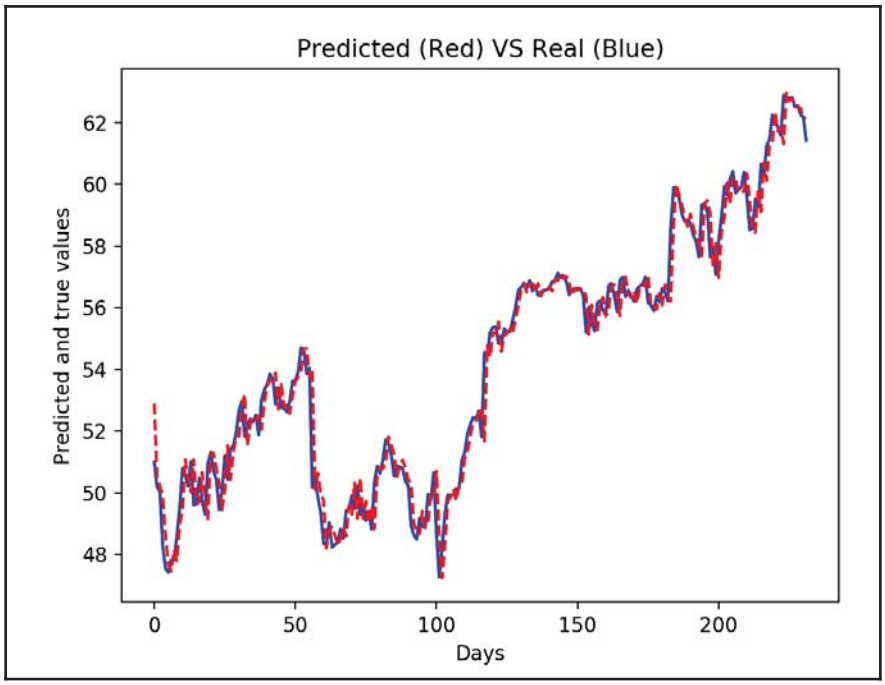
This is the predicted value:



That's the absolute error, with the trend line (dotted):



And finally, the real and predicted value in the train set:



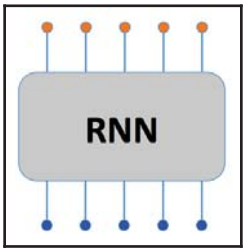
Remember that those are the performances of a simple regression algorithm, without exploiting the temporal correlation between features. How can we exploit it to perform better?

Long short-term memory – LSTM 101

Long Short-Term Memory (**LSTM**), models are a special case of RNNs, Recurrent Neural Networks. A full, rigorous description of them is out of the scope of this book; in this section, we will just provide the essence of them.

You can have a look at the following books published by Packt:
`https://www.packtpub.com/big-data-and-business-intelligence/neural-network-programming-tensorflow`
Also, you can have a look at this: `https://www.packtpub.com/big-data-and-business-intelligence/neural-networks-r`

Simply speaking, RNN works on sequences: they accept multidimensional signals as input, and they produce a multidimensional output signal. In the following figure, there's an example of an RNN able to cope with a timeseries of five-time steps (one input for each time step). The inputs are in the bottom part of the RNN, with the outputs in the top. Remember that each input/output is an N-dimensional feature:



Inside, an RNN has multiple stages; each stage is connected to its input/output and to the output of the previous stage. Thanks to this configuration, each output is not just a function of the input of its own stage, but depends also on the output of the previous stage (which, again, is a function of its input and the previous output). This configuration ensures that each input influences all the following outputs, or, from the other side, an output is a function of all the previous and current stages inputs.

Note that not all the outputs are always used. Think about a sentiment analysis task, in that case, given a sentence (the timeseries input signals), we just want to get one class (positive/negative); therefore only the last output is considered as output, all the others exist, but they're not used. Keep in mind that we just use the last one because it's the only one with the full visibility of the sentence.

LSTM models are an evolution of RNNs: with long RNNs, the training phase may lead to very tiny or huge gradients back-propagated throughout the network, which leads the weights to zero or to infinity: that's a problem usually expressed as a vanishing/exploding gradient. To mitigate this problem, LSTMs have two outputs for each stage: one is the actual output of the model, and the other one, named memory, is the internal state of the stage.

Both outputs are fed into the following stage, lowering the chances of having vanishing or exploding gradients. Of course, this comes with a price: the complexity (numbers of weights to tune) and the memory footprint of the model are larger, that's why we strongly suggest using GPU devices when training RNNs, the speed up in terms of time is impressive!

Unlike regression, RNNs need a three dimensional signal as input. Tensorflow specifies the format as:

- Samples
- Time steps
- Features

In the preceding example, the sentiment analysis, the training tensor will have the sentences on the *x*-axis, the words composing the sentence on the *y*-axis, and the bag of words with the dictionary on the *z*-axis. For example, for classifying a 1 M corpora in English (with about 20,000 different words), whose sentences are long, up to 50 words, the tensor dimension is 1 M x 50 x 20 K.

Stock price prediction with LSTM

Thanks to LSTM, we can exploit the temporal redundancy contained in our signals. From the previous section, we learned that the observation matrix should be reformatted into a 3D tensor, with three axes:

1. The first containing the samples.
2. The second containing the timeseries.
3. The third containing the input features.

Since we're dealing with just a mono-dimensional signal, the input tensor for the LSTM should have the size (None, `time_dimension`, 1), where `time_dimension` is the length of the time window. Let's code now, starting with the cosine signal. We suggest you name the file `4_rnn_cosine.py`.

1. First of all, some imports:

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from evaluate_ts import evaluate_ts
from tensorflow.contrib import rnn
from tools import fetch_cosine_values, format_dataset
tf.reset_default_graph()
tf.set_random_seed(101)
```

2. Then, we set the window size to chunk the signal. This operation is similar to the observation matrix creation.

```
time_dimension = 20
train_size = 250
test_size = 250
```

3. Then, some settings for Tensorflow. At this stage, let's start with default values:

```
learning_rate = 0.01
optimizer = tf.train.AdagradOptimizer
n_epochs = 100
n_embeddings = 64
```

4. Now, it's time to fetch the noisy cosine, and reshape it to have a 3D tensor shape (None, `time_dimension`, 1). This is done here:

```
cos_values = fetch_cosine_values(train_size + test_size + time_dimension)
minibatch_cos_X, minibatch_cos_y = format_dataset(cos_values,
time_dimension)
train_X = minibatch_cos_X[:train_size, :].astype(np.float32)
train_y = minibatch_cos_y[:train_size].reshape((-1, 1)).astype(np.float32)
test_X = minibatch_cos_X[train_size:, :].astype(np.float32)
test_y = minibatch_cos_y[train_size:].reshape((-1, 1)).astype(np.float32)
train_X_ts = train_X[:, :, np.newaxis]
test_X_ts = test_X[:, :, np.newaxis]
```

5. Exactly as in the previous script, let's define the placeholders for Tensorflow:

```
X_tf = tf.placeholder("float", shape=(None, time_dimension, 1), name="X")
y_tf = tf.placeholder("float", shape=(None, 1), name="y")
```

6. Here, let's define the model. We will use an LSTM with a variable number of embeddings. Also, as described in the previous chapter, we will consider just the last output of the cells through a linear regression (fully connected layer) to get the prediction:

```
def RNN(x, weights, biases):
    x_ = tf.unstack(x, time_dimension, 1)
    lstm_cell = rnn.BasicLSTMCell(n_embeddings)
    outputs, _ = rnn.static_rnn(lstm_cell, x_, dtype=tf.float32)
    return tf.add(biases, tf.matmul(outputs[-1], weights))
```

7. Let's set the `trainable` variables (`weights`) as before, the `cost` function and the training operator:

```
weights = tf.Variable(tf.truncated_normal([n_embeddings, 1], mean=0.0,
stddev=1.0), name="weights")
biases = tf.Variable(tf.zeros([1]), name="bias")
y_pred = RNN(X_tf, weights, biases)
cost = tf.reduce_mean(tf.square(y_tf - y_pred))
train_op = optimizer(learning_rate).minimize(cost)

# Exactly as before, this is the main loop.
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    # For each epoch, the whole training set is feeded into the tensorflow
graph
    for i in range(n_epochs):
        train_cost, _ = sess.run([cost, train_op], feed_dict={X_tf:
train_X_ts, y_tf: train_y})
        if i%100 == 0:
            print("Training iteration", i, "MSE", train_cost)

    # After the training, let's check the performance on the test set
    test_cost, y_pr = sess.run([cost, y_pred], feed_dict={X_tf: test_X_ts,
y_tf: test_y})
    print("Test dataset:", test_cost)

    # Evaluate the results
    evaluate_ts(test_X, test_y, y_pr)

    # How does the predicted look like?
    plt.plot(range(len(cos_values)), cos_values, 'b')
    plt.plot(range(len(cos_values)-test_size, len(cos_values)), y_pr, 'r--
')
    plt.xlabel("Days")
    plt.ylabel("Predicted and true values")
    plt.title("Predicted (Red) VS Real (Blue)")
    plt.show()
```

The output, after a hyperparameter optimization, is the following:

```
Training iteration 0 MSE 0.0603129
Training iteration 100 MSE 0.0054377
Training iteration 200 MSE 0.00502512
Training iteration 300 MSE 0.00483701
...
Training iteration 9700 MSE 0.0032881
```

```
Training iteration 9800 MSE 0.00327899
Training iteration 9900 MSE 0.00327195
Test dataset: 0.00416444
Evaluation of the predictions:
MSE: 0.00416444
mae: 0.0545878
```

Performances are pretty similar to the ones we obtained with the simple linear regression. Let's see if we can get better performance using a less predictable signal as the stock price. We'll use the same timeseries we used in the previous chapter, to compare the performance.

Modifying the previous program, let's plug in the stock price timeseries instead of the cosine. Modify some lines to load the stock price data:

```
stock_values = fetch_stock_price(symbol, datetime.date(2015, 1, 1),
datetime.date(2016, 12, 31))
minibatch_cos_X, minibatch_cos_y = format_dataset(stock_values,
time_dimension)
train_X = minibatch_cos_X[:train_size, :].astype(np.float32)
train_y = minibatch_cos_y[:train_size].reshape((-1, 1)).astype(np.float32)
test_X = minibatch_cos_X[train_size:, :].astype(np.float32)
test_y = minibatch_cos_y[train_size:].reshape((-1, 1)).astype(np.float32)
train_X_ts = train_X[:, :, np.newaxis]
test_X_ts = test_X[:, :, np.newaxis]
```

Since the dynamic of this signal is wider, we'll also need to modify the distribution used to extract the initial weights. We suggest you set it to:

```
weights = tf.Variable(tf.truncated_normal([n_embeddings, 1], mean=0.0,
stddev=10.0), name="weights")
```

After a few tests, we found we hit the maximum performance with these parameters:

```
learning_rate = 0.1
n_epochs = 5000
n_embeddings = 256
```

The output, using these parameters is:

```
Training iteration 200 MSE 2.39028
Training iteration 300 MSE 1.39495
Training iteration 400 MSE 1.00994
...
Training iteration 4800 MSE 0.593951
Training iteration 4900 MSE 0.593773
Test dataset: 0.497867
Evaluation of the predictions:
MSE: 0.497867
mae: 0.494975
```

This is 8% better than the previous model (test MSE). Remember, it comes with a price! More parameters to train also means the training time is much longer than the previous example (on a laptop, a few minutes, using the GPU).

Finally, let's check the Tensorboard. In order to write the logs, we should add the following code:

1. At the beginning of the files, after the imports:

```
import os
tf_logdir = "./logs/tf/stock_price_lstm"
os.makedirs(tf_logdir, exist_ok=1)
```

2. Also, the whole body of the `RNN` function should be inside the named-scope LSTM, that is:

```
def RNN(x, weights, biases):
    with tf.name_scope("LSTM"):
        x_ = tf.unstack(x, time_dimension, 1)
        lstm_cell = rnn.BasicLSTMCell(n_embeddings)
        outputs, _ = rnn.static_rnn(lstm_cell, x_, dtype=tf.float32)
        return tf.add(biases, tf.matmul(outputs[-1], weights))
```

3. Similarly, the `cost` function should be wrapped in a Tensorflow scope. Also, we will add the `mae` computation within the `tensorflow` graph:

```
y_pred = RNN(X_tf, weights, biases)
with tf.name_scope("cost"):
    cost = tf.reduce_mean(tf.square(y_tf - y_pred))
    train_op = optimizer(learning_rate).minimize(cost)
    tf.summary.scalar("MSE", cost)
        with tf.name_scope("mae"):
        mae_cost = tf.reduce_mean(tf.abs(y_tf - y_pred))
        tf.summary.scalar("mae", mae_cost)
```

4. Finally, the main function should look like this:

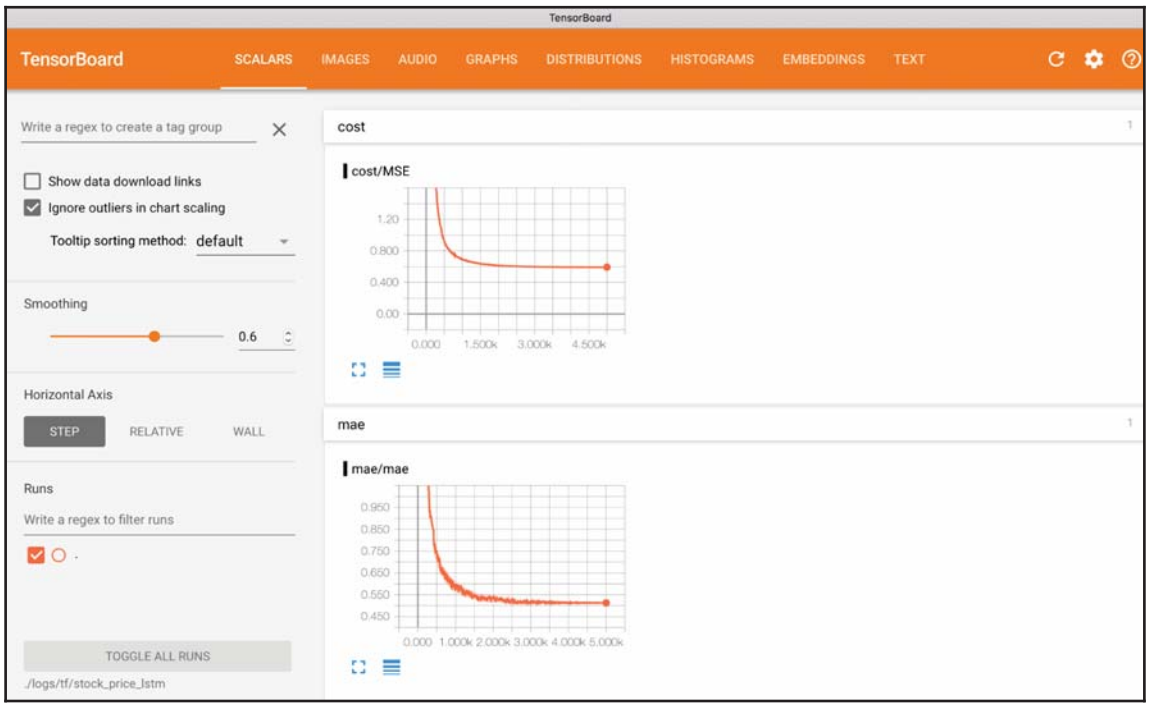
```
with tf.Session() as sess:
    writer = tf.summary.FileWriter(tf_logdir, sess.graph)
    merged = tf.summary.merge_all()
    sess.run(tf.global_variables_initializer())

    # For each epoch, the whole training set is feeded into the tensorflow
graph
    for i in range(n_epochs):
        summary, train_cost, _ = sess.run([merged, cost, train_op],
feed_dict={X_tf:
                                                    train_X_ts, y_tf:
train_y})
        writer.add_summary(summary, i)
        if i%100 == 0:
            print("Training iteration", i, "MSE", train_cost)
    # After the training, let's check the performance on the test set
    test_cost, y_pr = sess.run([cost, y_pred], feed_dict={X_tf: test_X_ts,
y_tf:
           test_y})
    print("Test dataset:", test_cost)
```

This way, we separate the scopes of each block, and write a summary report for the trained variables.

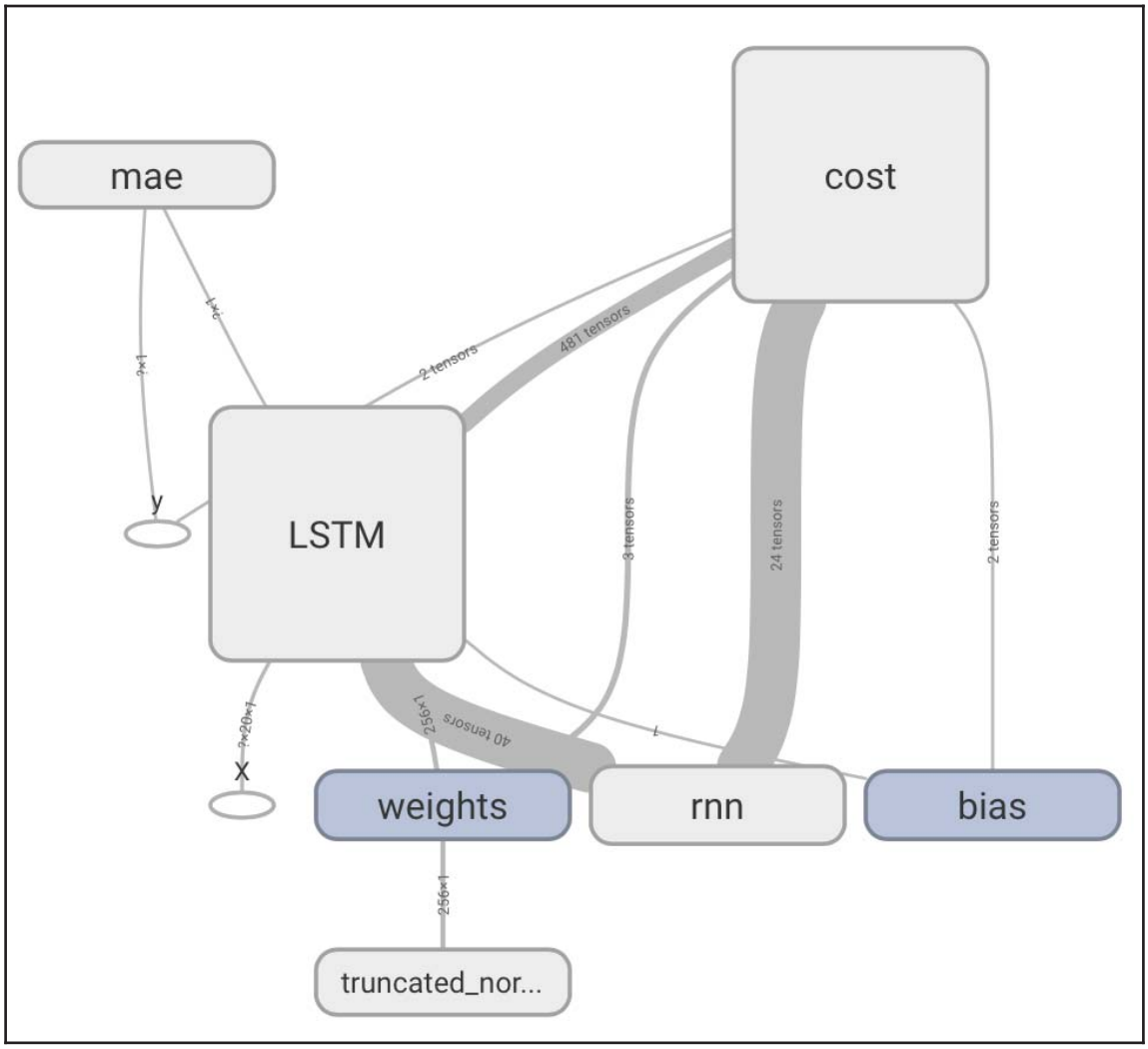
Now, let's launch `tensorboard`:

```
$> tensorboard --logdir=./logs/tf/stock_price_lstm
```

After opening the browser at `localhost:6006`, from the first tab, we can observe the behavior of the MSE and MAE:



The trend looks very nice, it goes down until it reaches a plateau. Also, let's check the `tensorflow` graph (in the tab **GRAPH**). Here we can see how things are connected together, and how operators are influenced by each other. You can still zoom in to see exactly how LSTMs are built in Tensorflow:



And that's the end of the project.

Possible follow - up questions

- Replace the LSTM with an RNN, and then with a GRU. Who's the best performer?
- Instead of predicting the closing price, try predicting also the high/low the day after. To do so, you can use the same features while training the model (or you can just use the closing price as input).
- Optimize the model for other stocks: is it better to have a generic model working for all the stocks or one specific for each stock?
- Tune the retraining. In the example, we predicted a full year with the model. Can you notice any improvement if you train the model once a month/week/day?
- If you have some financial experience, try building a simple trading simulator and feed it with the predictions. Starting the simulation with $100, will you gain or lose money after a year?

Summary

In this chapter we've shown how to perform a prediction of a timeseries: specifically, we observed how well RNN performs with real datasets as the stock prices. In the next chapter, we will see another application of the RNN, for example, how to perform an automatic machine translation for translating a sentence in another language.