The City College of The City University of New York

CSC 1800: Topics in Artificial Intelligence

Project “Stock Forecasting”

Authors: Aleksandr Tsema
Lalchandra Rampersaud

New York

2016

Task

The task here will be to be able to predict values for a timeseries: the history of 6 months Apple's Inc. stock prices. We are going to use a multi-layered LSTM recurrent neural network to predict the last value of a sequence of values. Put another way, given 59 timesteps(minutes) of consumption, what will be the 60th value?

Imports

For that task we are going to use Keras with Theano backend.

We start with importing everything we'll need:

```
import matplotlib.pyplot as plt
import numpy as np
import time
import csv
from keras.models import Sequential
from keras.layers.core import Dense, Activation, Dropout
from keras.layers.recurrent import LSTM
np.random.seed(1234)
```

- `matplotlib`, `numpy`, `time` are pretty straight forward.
- `csv` is a module that will be used to load the data from the `txt` file.
- `models` are the core of Keras's neural networks implementation. It is the object that represents the network: it will have layers, activations and so on. It is the object that will be 'trained' and 'tested'. `Sequetial` means we will use a 'layered' model, not a graphical one.
- `Dense, Activation, Dropout` core layers are used to build the network: feedforward standard layers and Activation and Dropout modules to parametrize the layers.
- `LSTM` is a reccurent layer.

Last thing is that for reproductibility, a seed is used in numpy's random.

Loading the data

```
def data_power_consumption(path_to_dataset='stock.csv', sequence_length=60,
ratio=1.0):

    max_values = ratio * 33852

    with open(path_to_file) as f:
        data = csv.reader(f, delimiter=";")
        power = []
        nb_of_values = 0
        for line in data:
            try:
                power.append(float(line[2]))
                nb_of_values += 1
            except ValueError:
                pass
            # 2049280.0 is the total number of valid values, i.e. ratio = 1.0
            if nb_of_values / 33852.0 >= ratio:
                break
    result = []
    for index in range(len(power) - sequence_length):
        result.append(power[index: index + sequence_length])
    result = np.array(result)  # shape (33852, 50
```

Once all the datapoints are loaded as one large timeseries, we have to **split** it into examples. Again, one example is made of a sequence of 60 values. Using the first 59, we are going to try and predict the 60th. Moreover, we'll do this for every minute given the 59 previous ones so we use a sliding buffer of size 60.

```
    result_mean = result.mean()
    result -= result_mean
    print "Shift : ", result_mean
    print "Data  : ", result.shape
```

Neural networks usually learn way better when data is pre-processed. However, regarding time-series we do not want the network to learn on data too far from the real world. So here we'll keep it simple and simply center the data to have a `0` mean.

```
    row = round(0.9 * result.shape[0])
    train = result[:row, :]
    np.random.shuffle(train)
    X_train = train[:, :-1]
    y_train = train[:, -1]
    X_test = result[row:, :-1]
    y_test = result[row:, -1]
```

Now that the examples are formatted, we need to split them into train and test, input and target. Here we select 10% of the data as test and 90% to train. We also select the last value of each example to be the target, the rest being the sequence of inputs.

We shuffle the training examples so that we train in no particular order and the distribution is uniform (for the batch calculation of the loss) but not the test set so that we can visualize our predictions with real signals.

```
    X_train = np.reshape(X_train, (X_train.shape[0], X_train.shape[1], 1))
    X_test = np.reshape(X_test, (X_test.shape[0], X_test.shape[1], 1))

    return [X_train, y_train, X_test, y_test]
```

Last thing regards input formats. So we reshape the inputs to have dimensions (`#examples`, `#values in sequences`, `dim. of each value`). Here each value is 1-dimensional, they are only one measure (of power consumption at time t). However, if we were to predict speed vectors they could be 3 dimensional for instance.

In fine, we return `X_train, y_train, X_test, y_test` in a list (to be able to feed it as one only object to our `run` function).

Building the model

```
def build_model():

    model = Sequential()
    layers = [1, 60, 100, 1]
```

So here we are going to build our `Sequential` model. This means we're going to stack layers in this object.

Also, `layers` is the list containing the sizes of each layer. We are therefore going to have a network with 1-dimensional input, two hidden layers of sizes 60 and 100 and eventually a 1-dimensional output layer.

```
    model.add(LSTM(
            input_dim=layers[0],
            output_dim=layers[1],
            return_sequences=True))
    model.add(Dropout(0.2))
```

After the model is initialized, we create a first layer, in this case an LSTM layer. Here we use the default parameters so it behaves as a standard recurrent layer. Since our input is of 1 dimension, we declare that it should expect an `input_dim` of`1`. Then we say we want `layers[1]` units in this layer. We also add 20% `Dropout` in this layer.

```
    model.add(LSTM(
            layers[2],
            return_sequences=False))
    model.add(Dropout(0.2))
```

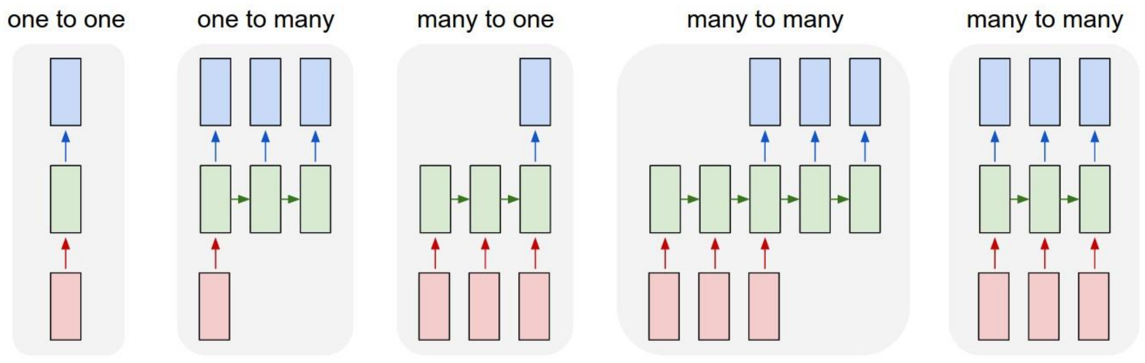
Second layer is even simpler to create, we just say how many units we want (`layers[2]`) and Keras takes care of the rest.

```
    model.add(Dense(
            output_dim=layers[3]))
    model.add(Activation("linear"))
```

The last layer we use is a Dense layer (feedforward). Since we are doing a regression, its activation is linear.

```
    start = time.time()
    model.compile(loss="mse", optimizer="rmsprop")
    print "Compilation Time : ", time.time() - start
    return model
```

Lastly, we compile the model using a Mean Square Error (again, it's standard for regression) and the `RMSprop` optimizer.



The difference between `return_sequence=True` and `return_sequence=False` is that in the first case the network behaves as in the 5th illustration (second *many to many*) and in the latter it behaves as the 3rd, *many to one*.

In our case, the first LSTM layer returns sequences because we want it to transfer its information both to the next layer (upwards in the chart) and to itself for the next timestep (arrow to the right).

However, for the second one, we just expect its last sequence prediction to be compared to the target. This means for inputs 0 to `sequence_length - 2` the prediction is only passed to the layer itself for the next timestep and not as an input to the next (output) layer. However, the `sequence_length - 1`th input is passed forward to the Dense layer for the loss computation against the target.

To conclude, the fact that `return_sequence=True` for the first layer means that its output is always fed to the second layer. As a whole regarding time, all its activations can be seen as the sequence of prediction this first layer has made from the input sequence.
On the other hand, `return_sequence=False` for the second layer because its output is only fed to the next layer at the end of the sequence. As a whole regarding time, it does not output a prediction for the sequence but one only prediction-vector (of size `layer[2]`) for the whole input sequence. The linear `Dense` layer is used to aggregate all the information from this prediction-vector into one single value, the predicted 4th timestep of the sequence.

Had we stacked three recurrent hidden layers, we'd have set `return_sequence=True` to the second hidden layer and`return_sequence=False` to the last. In other words, `return_sequence=False` is used as an interface from recurrent to feedforward layers (dense or convolutionnal).

Running the network

```
def run_network(model=None, data=None):
    epochs = 1
    ratio = 0.5
    path_to_dataset = 'stock.csv'
    if data is None:
        print 'Loading data... '
        X_train, y_train, X_test, y_test = data_power_consumption(
                path_to_dataset, sequence_length, ratio)
    else:
        X_train, y_train, X_test, y_test = data

    print '\nData Loaded. Compiling...\n'

    if model is None:
        model = build_model()
```

Just like before, to be as modular as possible we start with checking whether or not `data` and `model` values were provided. If not, we load the data and build the model. Set `ratio` to the proportion of the entire dataset you want to load (of course `ratio <= 1` ... if not `data_power_consumption` will behave as if `ratio = 1`)

```
    try:
        model.fit(
            X_train, y_train,
            batch_size=512, nb_epoch=epochs, validation_split=0.05)
        predicted = model.predict(X_test)
        predicted = np.reshape(predicted, (predicted.size,))
    except KeyboardInterrupt:
        print 'Training duration (s) : ', time.time() - global_start_time
        return model, y_test, 0
```

Again, we put the training into a try/except statement so that we can interrupt the training without losing everything to a `KeyboardInterrupt`.

To train the model, we call the `model`'s `fit` method. Nothing new here. .

Let's focus a bit on `predicted`.

- by construction `X_test` is an array with 59 columns (timesteps). The list `[ X_test[i][0] ]` is the entire signal (minus the last 59 values) from which it was built since we've used a 1-timestep sliding buffer.
- `X_test[0]` is the first sequence, that is to say the first 59 values of the original signal.
- `predict(X_test[0])` is therefore the prediction for the 50th value and its associated target is `y_test[0]`. Moreover, by construction, `y_test[0] = X_test[1][48] = X_test[2][47] = ...`

- then `predict(X_test[1])` is the prediction of the 51th value, associated with `y_test[1]` as a target.
- therefore `predict(X_test)` is the predicted signal, one step ahead, and `y_test` is its target.
- `predict(X_test)` is a list of lists (in fact a 2-dimensional numpy array) with one value, therefore we reshape it so that it simply is a list of values (1-dimensional numpy array).

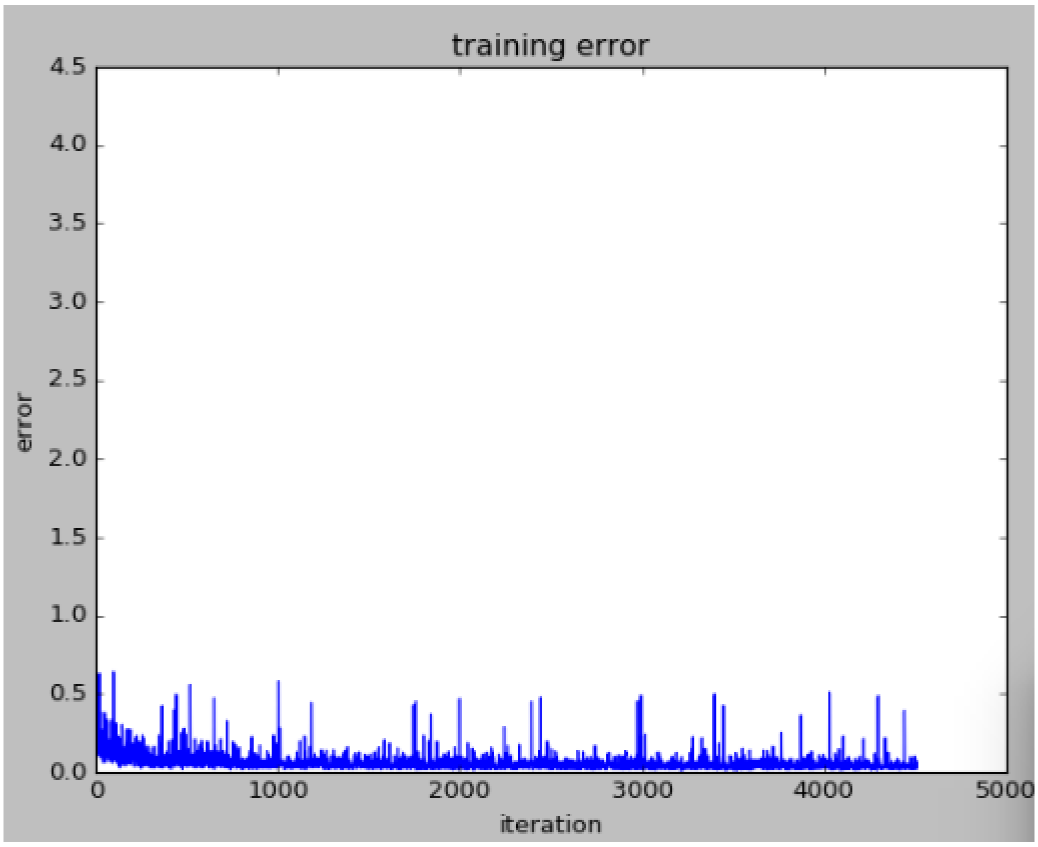
In case of keyboard interruption, we return the `model`, `y_test` and `X_test`.

Plotting

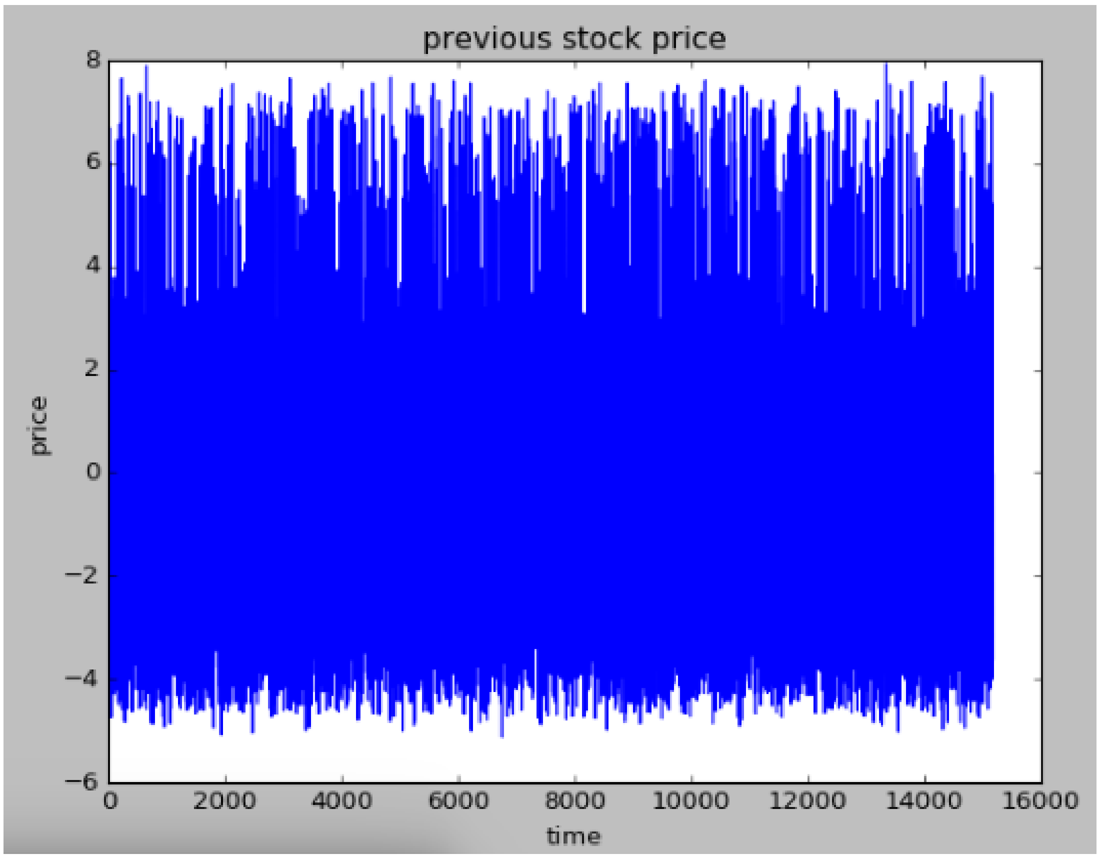
```
 plt.figure(1)
plt.plot(history.losses, 'b')
plt.ylabel('error')
plt.xlabel('iteration')
plt.title('training error')

plt.figure(2)
plt.plot(y_test, 'g', label='real price')
plt.plot(predicted, 'r', label='predicted price')
plt.ylabel('price')
plt.xlabel('time')
plt.title('predicted stock price')

plt.figure(3)
plt.plot(y_train, 'b')
plt.ylabel('price')
plt.xlabel('time')
plt.title('previous stock price')

plt.show()
```

Results

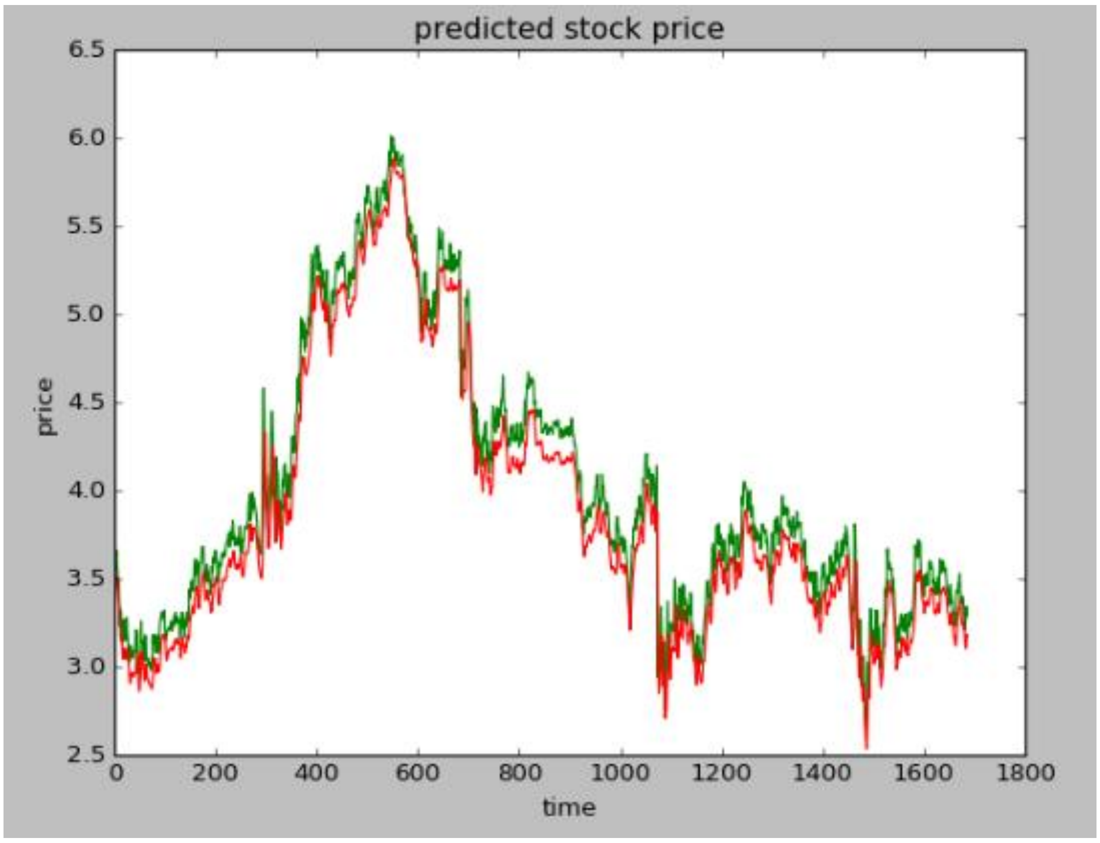
The network predicts prices not enough accurate for real trading. The error is about 20%.



Picture 1. Cost function



Picture 2. Stock price over last 33852 minutes (6 months).



Picture 3. Stock price for next 1700 minutes. Green is real price, red is predicted.