Perform time series prediction

Problem 1: Perform time series prediction

In this project you will perform time series prediction using a Recurrent Neural Network regressor. In particular you will re-create the figure shown in the notes - where the stock price of Apple was forecasted (or predicted) 7 days in advance. In completing this exercise you will learn how to construct RNNs using Keras, which will also aid in completing the second project in this notebook.

The particular network architecture we will employ for our RNN is known as Long Short Term Memory (LSTM), which helps significantly avoid technical problems with optimization of RNNs.

Long short-term memory (**LSTM**) is a recurrent neural network (RNN) architecture that remembers values[1] over arbitrary intervals. Stored values are not modified as learning proceeds. RNNs allow forward and backward connections between neurons.

An LSTM is well-suited to classify, process and predict time series given time lags of unknown size and duration between important events. Relative insensitivity to gap length gives an advantage to LSTM over alternative RNNs, hidden Markov models and other sequence learning methods in numerous applications

1.1 Getting started

First we must load in our time series - a history of around 140 days of Apple's stock price. Then we need to perform a number of pre-processing steps to prepare it for use with an RNN model. First off, it is good practice to normalize time series - by normalizing its range. This helps us avoid serious numerical issues associated how common activation functions (like tanh) transform very large (positive or negative) numbers, as well as helping us to avoid related issues when computing derivatives.

Here we normalize the series to lie in the range [0,1] using this scikit function,

class sklearn.preprocessing.**MinMaxScaler**(*feature_range=(0, 1)*, *copy=True*)[source]

> Transforms features by scaling each feature to a given range.
>
> This estimator scales and translates each feature individually such that it is in the given range on the training set, i.e. between zero and one.
>
> The transformation is given by:
>
> ```
> X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
> X_scaled = X_std * (max - min) + min
> ```
>
> where min, max = feature_range.
>
> This transformation is often used as an alternative to zero mean, unit variance scaling.

but it is also commonplace to normalize by a series standard deviation.

```
### Load in necessary libraries for data input and normalization
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

%load_ext autoreload
```

Perform time series prediction

```
%autoreload 2

from my_answers import *

%load_ext autoreload
%autoreload 2

from my_answers import *

### load in and normalize the dataset
dataset = np.loadtxt('datasets/normalized_apple_prices.csv')
```

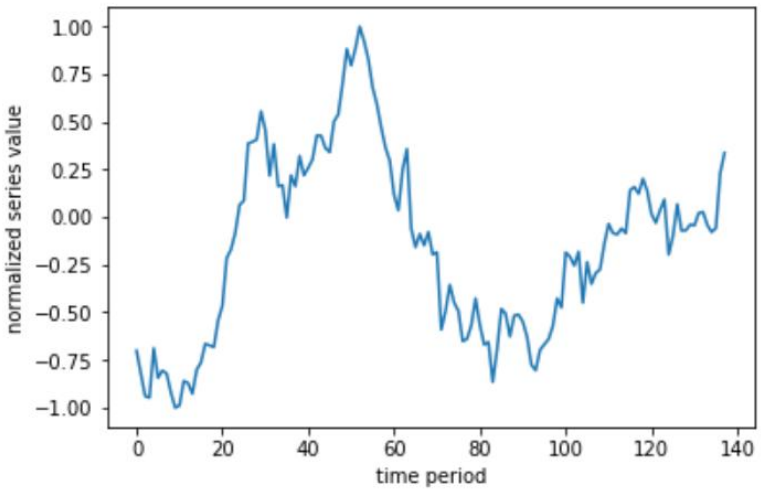
```
Using TensorFlow backend.
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

Lets take a quick look at the (normalized) time series we'll be performing predictions on

```
# lets take a look at our time series
plt.plot(dataset)
plt.xlabel('time period')
plt.ylabel('normalized series value')
```

Out[2]:

```
<matplotlib.text.Text at 0x7f3f81ac0518>
```



1.2 Cutting our time series into sequences

Remember, our time series is a sequence of numbers that we can represent in general mathematically as

$$s0,s1,s2,...,sPs0,s1,s2,...,sP$$

where $spsp$ is the numerical value of the time series at time period pp and where PP is the total length of the series. In order to apply our RNN we treat the time series prediction problem as a regression problem, and so need to use a sliding window to construct a set of associated input/output pairs to regress on. This process is animated in the gif below.

Input	Output
$\langle s_1, s_2, s_3, s_4, s_5 \rangle$	s_6
$\langle s_2, s_3, s_4, s_5, s_6 \rangle$	s_7
$\vdots$	$\vdots$
$\langle s_{P-5}, s_{P-4}, s_{P-3}, s_{P-2}, s_{P-1} \rangle$	s_P

Perform time series prediction

Notice here that each input is a sequence (or vector) of length 4 (and in general has length equal to the window size T) while each corresponding output is a scalar value. Notice also how given a time series of length P and window size T = 5 as shown above, we created P - 5 input/output pairs. More generally, for a window size T we create P - T such pairs.
Now its time for you to window the input time series as described above!

TODO: Implement the function called **window_transform_series** in my_answers.py so that it runs a sliding window along the input series and creates associated input/output pairs. Note that this function should input a) the series and b) the window length, and return the input/output subsequences. Make sure to format returned input/output as generally shown in table above (where window_size = 5), and make sure your returned input is a numpy array.

You can test your function on the list of odd numbers below

In [3]:

```
odd_nums = np.array([1,3,5,7,9,11,13])
```

Here is a hard-coded solution for odd_nums. You can compare its results with what you get from your **window_transform_series** implementation.

In [4]:

```
# run a window of size 2 over the odd number sequence and display the results
window_size = 2

X = []
X.append(odd_nums[0:2])
X.append(odd_nums[1:3])
X.append(odd_nums[2:4])
X.append(odd_nums[3:5])
X.append(odd_nums[4:6])
X.append(odd_nums[5:7])

y = odd_nums[2:]

X = np.asarray(X)
y = np.asarray(y)
y = np.reshape(y, (len(y),1)) #optional

assert(type(X).__name__ == 'ndarray')
assert(type(y).__name__ == 'ndarray')
assert(X.shape == (6,2))
assert(y.shape in [(5,1), (5,)])

# print out input/output pairs --> here input = X, corresponding output = y
print ('--- the input X will look like ----')
print (X)

print ('--- the associated output y will look like ----')
print (y)
```

```
--- the input X will look like ----
[[ 1  3]
 [ 3  5]
 [ 5  7]
```

```
 [ 7  9]
 [ 9 11]
 [11 13]]
--- the associated output y will look like ----
[[ 5]
 [ 7]
 [ 9]
 [11]
 [13]]
```

Again - you can check that your completed **window_transform_series** function works correctly by trying it on the odd_nums sequence - you should get the above output.

In [5]:

```
### TODO: implement the function window_transform_series in the file
my_answers.py
from my_answers import window_transform_series
```

With this function in place apply it to the series in the Python cell below. We use a window_size = 7 for these experiments.

In [6]:

```
# window the data using your windowing function
window_size = 7
X, y = window_transform_series(series=dataset, window_size=window_size)
```

1.3 Splitting into training and testing sets

In order to perform proper testing on our dataset we will lop off the last 1/3 of it for validation (or testing). This is that once we train our model we have something to test it on (like any regression problem!). This splitting into training/testing sets is done in the cell below.

Note how here we are **not** splitting the dataset *randomly* as one typically would do when validating a regression model. This is because our input/output pairs *are related temporally*. We don't want to validate our model by training on a random subset of the series and then testing on another random subset, as this simulates the scenario that we receive new points *within the timeframe of our training set*.

We want to train on one solid chunk of the series (in our case, the first full 2/3 of it), and validate on a later chunk (the last 1/3) as this simulates how we would predict *future* values of a time series.

In [7]:

```
# split our dataset into training / testing sets
train_test_split = int(np.ceil(2*len(y)/float(3)))   # set the split point

# partition the training set
X_train = X[:train_test_split,:]
y_train = y[:train_test_split]

# keep the last chunk for testing
X_test = X[train_test_split:,:]
y_test = y[train_test_split:]

# NOTE: to use keras's RNN LSTM module our input must be reshaped to [samples,
window size, stepsize]
```

Perform time series prediction

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

1.4 Build and run an RNN regression model

Having created input/output pairs out of our time series and cut this into training/testing sets, we can now begin setting up our RNN. We use Keras to quickly build a two hidden layer RNN of the following specifications

- layer 1 uses an LSTM module with 5 hidden units (note here the input_shape = (window_size,1))
- layer 2 uses a fully connected module with one unit
- the 'mean_squared_error' loss should be used (remember: we are performing regression here)

This can be constructed using just a few lines - see e.g., the general Keras documentation and the LTSM documentation in particular for examples of how to quickly use Keras to build neural network models:

RNN

```
keras.layers.RNN(cell, return_sequences=False, return_state=False,
go_backwards=False, stateful=False, unroll=False)
```

Base class for recurrent layers.

Arguments

- **cell**: A RNN cell instance. A RNN cell is a class that has:
  - a `call(input_at_t, states_at_t)` method, returning`(output_at_t, states_at_t_plus_1)`.
  - a `state_size` attribute. This can be a single integer (single state) in which case it is the size of the recurrent state (which should be the same as the size of the cell output). This can also be a list/tuple of integers (one size per state). In this case, the first entry (`state_size[0]`) should be the same as the size of the cell output. It is also possible for `cell` to be a list of RNN cell instances, in which cases the cells get stacked on after the other in the RNN, implementing an efficient stacked RNN.

  return_sequences: Boolean. Whether to return the last output. in the output sequence, or the full sequence.

  return_state: Boolean. Whether to return the last state in addition to the output.

  go_backwards: Boolean (default False). If True, process the input sequence backwards and return the reversed sequence.

  stateful: Boolean (default False). If True, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch.

  unroll: Boolean (default False). If True, the network will be unrolled, else a symbolic loop will be used. Unrolling can speed-up a RNN, although it tends to be more memory-intensive. Unrolling is only suitable for short sequences.

  input_dim: dimensionality of the input (integer). This argument (or alternatively, the keyword argument `input_shape`) is required when using this layer as the first layer in a model.

  input_length: Length of input sequences, to be specified when it is constant. This argument is required if you are going to connect `Flatten` then `Dense` layers upstream (without it, the shape of the dense outputs cannot be computed). Note that if the recurrent layer is not the first layer in your model, you would need to specify the input length at the level of the first layer (e.g. via the `input_shape`argument)

Output shape

- if `return_state`: a list of tensors. The first tensor is the output. The remaining tensors are the last states, each with shape `(batch_size, units)`.
- if `return_sequences`: 3D tensor with shape `(batch_size, timesteps, units)`.
- else, 2D tensor with shape `(batch_size, units)`.

Make sure you are initializing your optimizer given the [keras-recommended approach for RNNs]

Usage of optimizers

An optimizer is one of the two arguments required for compiling a Keras model:

```
from keras import optimizers

model = Sequential()
model.add(Dense(64, kernel_initializer='uniform', input_shape=(10,)))
model.add(Activation('tanh'))
model.add(Activation('softmax'))

sgd = optimizers.SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)

model.compile(loss='mean_squared_error', optimizer=sgd)
```

(given in the cell below). (remember to copy your completed function into the script *my_answers.py* function titled *build_part1_RNN* before submitting your project)

In [8]:

```
import timeit
# graph the history of model.fit
def show_history_graph(history):
    # summarize history for loss
    plt.plot(history.history['loss'])
    plt.title('model loss')
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['train'], loc='upper left')
    plt.show()

class EpochTimer(keras.callbacks.Callback):
    train_start = 0
    train_end = 0
    epoch_start = 0
    epoch_end = 0

    def get_time(self):
        return timeit.default_timer()

    def on_train_begin(self, logs={}):
        self.train_start = self.get_time()

    def on_train_end(self, logs={}):
        self.train_end = self.get_time()
        print('Training took {} seconds'.format(self.train_end -
self.train_start))
    def on_epoch_begin(self, epoch, logs={}):
        self.epoch_start = self.get_time()
```

Perform time series prediction

```
    def on_epoch_end(self, epoch, logs={}):
        self.epoch_end = self.get_time()
        print('Epoch {} took {} seconds'.format(epoch, self.epoch_end -
self.epoch_start))
```

In [9]:

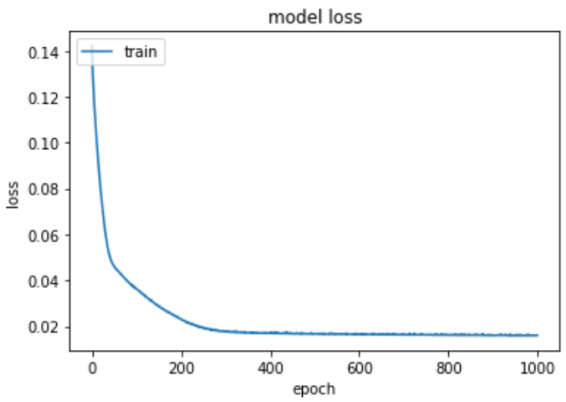
```
### TODO: create required RNN model
# import keras network libraries
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
import keras

# given - fix random seed - so we can all reproduce the same results on our
default time series
np.random.seed(0)

# TODO: implement build_part1_RNN in my_answers.py
from my_answers import build_part1_RNN
model = build_part1_RNN(window_size)

# build model using keras documentation recommended optimizer initialization
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=1e-08,
decay=0.0)

# compile the model
model.compile(loss='mean_squared_error', optimizer=optimizer)
```

With your model built you can now fit the model by activating the cell below! Note: the number of epochs (np_epochs) and batch_size are preset (so we can all produce the same results). You can choose to toggle the verbose parameter - which gives you regular updates on the progress of the algorithm - on and off by setting it to 1 or 0 respectively.

In [10]:

```
# run your model!
epochtimer = EpochTimer()
epochs = 1000
hist = model.fit(X_train, y_train, epochs=epochs, batch_size=50, verbose=0,
callbacks=[epochtimer])
show_history_graph(hist)
```

```
Epoch 0 took 3.100143152001692 seconds
Epoch 1 took 0.022702102000039304
seconds
.....
Epoch 998 took 0.023996402000193484
seconds
Epoch 999 took 0.02392203899944434
seconds
Training took 27.35786962199927 seconds
```



1.5 Checking model performance

Perform time series prediction

With your model fit we can now make predictions on both our training and testing sets.

In [11]:

```
# generate predictions for training
train_predict = model.predict(X_train)
test_predict = model.predict(X_test)
```

In the next cell we compute training and testing errors using our trained model - you should be able to achieve at least

training_error < 0.02 and *testing_error* < 0.02

with your fully trained model.

If either or both of your accuracies are larger than 0.02 re-train your model - increasing the number of epochs you take (a maximum of around 1,000 should do the job) and/or adjusting your batch_size.

In [12]:

```
# print out training and testing errors
training_error = model.evaluate(X_train, y_train, verbose=0)
print('training error = ' + str(training_error))

testing_error = model.evaluate(X_test, y_test, verbose=0)
print('testing error = ' + str(testing_error))
```

```
training error = 0.0160044282675
testing error = 0.0139837323276
```

Activating the next cell plots the original data, as well as both predictions on the training and testing sets.

In [13]:

```
### Plot everything - the original series as well as predictions on training
and testing sets
import matplotlib.pyplot as plt
%matplotlib inline

# plot original series
plt.plot(dataset,color = 'k')

# plot training set prediction
split_pt = train_test_split + window_size
plt.plot(np.arange(window_size,split_pt,1),train_predict,color = 'b')

# plot testing set prediction
plt.plot(np.arange(split_pt,split_pt + len(test_predict),1),test_predict,color
= 'r')

# pretty up graph
plt.xlabel('day')
plt.ylabel('(normalized) price of Apple stock')
plt.legend(['original series','training fit','testing fit'],loc='center left',
bbox_to_anchor=(1, 0.5))
plt.show()
```

Perform time series prediction