Images haven't loaded yet. Please exit printing, wait for images to load, and try to print again.

ike 👍,

but humans create 👒, discover 🚀 and love ❤️

Jun 19, 2016 · 7 min read

Neural networks for algorithmic trading. Simple time series forecasting

Ciao, people!

IMPORTANT UPDATE:

> ***I've made a mistake in this post while preprocessing data for regression problem—check this issue https://github.com/Rachnog/Deep-Trading/issues/1 to fix it. It causes to worse results, which can be partly improved by better hyperparameter search, using whole OHLC data and training for >50 epochs.***

This is first part of my experiments on application of deep learning to finance, in particular to algorithmic trading.

I want to implement trading system from scratch based only on deep learning approaches, so for any problem we have here (price prediction, trading strategy, risk management) we gonna use different

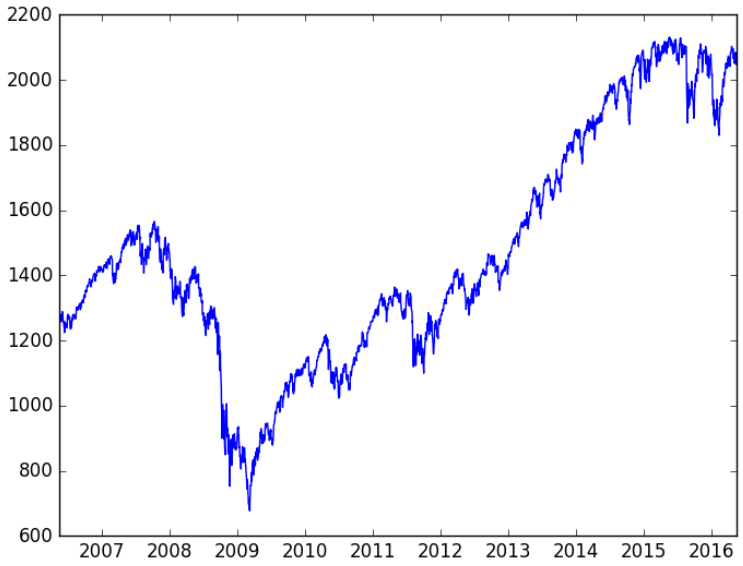
variations of artificial neural networks (ANNs) and check how well they can handle this.

Now I plan to work on next sections:

1. Simple time series forecasting (and mistakes done)
2. Correct 1D time series forecasting + backtesting
3. Multivariate time series forecasting
4. Volatility forecasting and custom losses
5. Multitask and multimodal learning
6. Hyperparameters optimization
7. Enhancing classical strategies with neural nets
8. Probabilistic programming and Pyro forecasts

I highly recommend you to check out code and IPython Notebook in this repository.

In this, first part, I want to show how MLPs, CNNs and RNNs can be used for financial time series prediction. In this part we are not going to use any feature engineering. Let's just consider historical dataset of **S&P 500 index** price movements. We have information from 1950 to 2016 about open, close, high, low prices for every day in the year and volume of trades. First, we will try just to predict close price in the end of the next day, second, we will try to predict return (close price—open price). Download the dataset from Yahoo Finance or from this repository.



S&P 500 daily close prices from 2006 to 2016

Problem definiton

We will consider our problem as 1) regression problem (trying to forecast exactly close price or return next day) 2) binary classification problem (price will go up [1; 0] or down [0; 1]).

For training NNs we gonna use framework Keras.

First let's prepare our data for training. We want to predict $t+1$ value based on N previous days information. For example, having close prices from past 30 days on the market we want to predict, what price will be tomorrow, on the 31st day.

We use first 90% of time series as training set (consider it as historical data) and last 10% as testing set for model evaluation.

Here is example of loading, splitting into training samples and preprocessing of raw input data:

```
def load_snp_close():
    f = open('table.csv', 'rb').readlines()[1:]
    raw_data = []
    raw_dates = []
    for line in f:
        try:
            close_price = float(line.split(',')[4])
            raw_data.append(close_price)
            raw_dates.append(line.split(',')[0])
        except:
            continue
    return raw_data, raw_dates

def split_into_chunks(data, train, predict, step, bin
    X, Y = [], []
    for i in range(0, len(data), step):
        try:
            x_i = data[i:i+train]
            y_i = data[i+train+predict]
            if binary:
                if y_i > 0.:
                    y_i = [1., 0.]
                else:
```

Regression problem. MLP

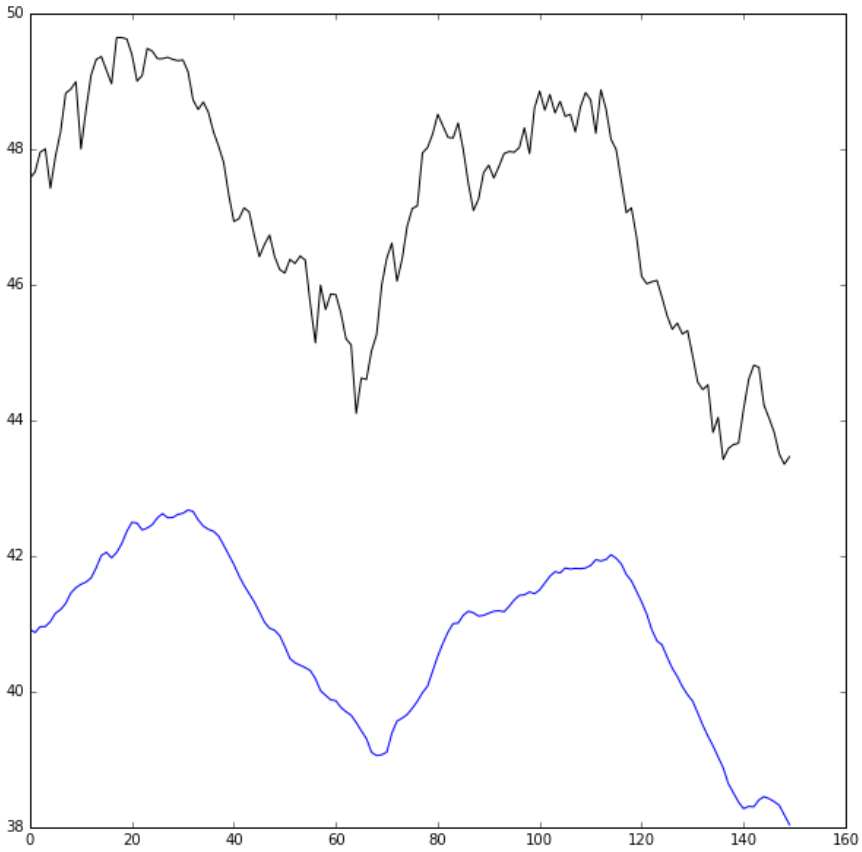
It will be just 2-hidden layer perceptron. Number of hidden neurons is chosen empirically, we will work on hyperparameters optimization in next sections. Between two hidden layers we add one Dropout layer to prevent overfitting.

Important thing is *Dense(1)*, *Activation('linear')* and *'mse'* in compile section. We want one output that can be in any range (we predict real value) and our loss function is defined as mean squared error.

```
model = Sequential()
model.add(Dense(500, input_shape = (TRAIN_SIZE, )))
model.add(Activation('relu'))
model.add(Dropout(0.25))
model.add(Dense(250))
model.add(Activation('relu'))
model.add(Dense(1))
```

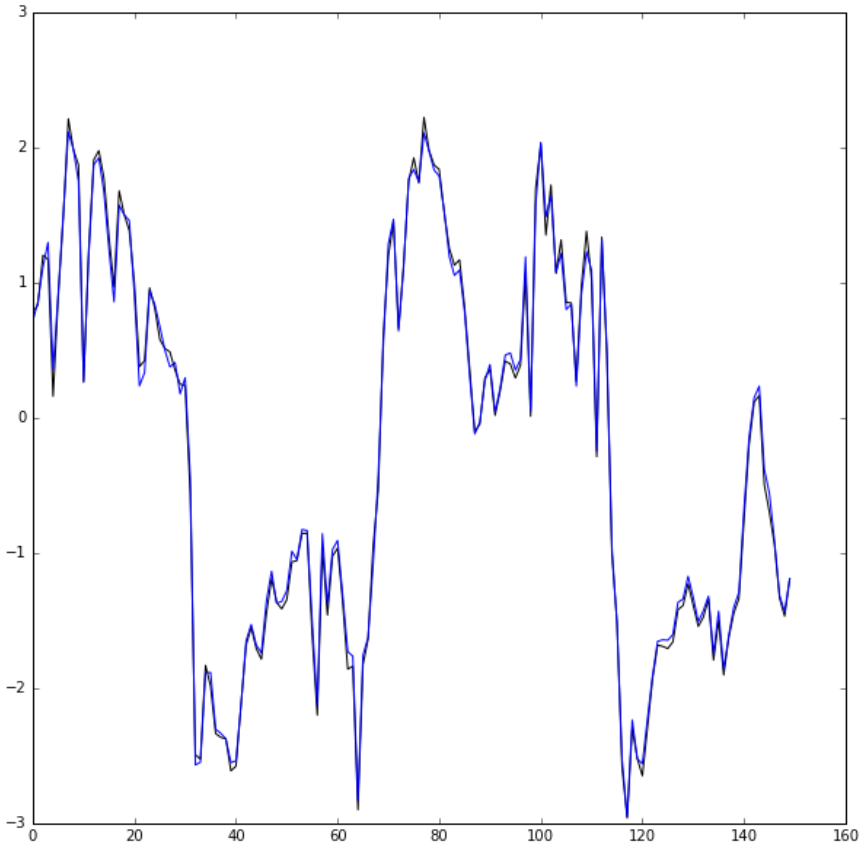
Let's see what happens if we just pass chunks of 20-days close prices and predict price on 21st day. Final MSE= 46.3635263557, but it's not very representative information. Below is plot of predictions for first 150 points of test dataset. Black line is actual data, blue one—predicted. We can clearly see that our algorithm is not even close by value, but can learn the trend.

```
predicted = model.predict(X_test)
try:
    fig = plt.figure(figsize=(width, height))
    plt.plot(Y_test[:150], color='black')
    plt.plot(predicted[:150], color='blue')
    plt.show()
```



Forecasting results of MLP trained on raw data

Let's scale our data using sklearn's method *preprocessing.scale()* to have our time series zero mean and unit variance and train the same MLP. Now we have MSE = 0.0040424330518 (but it is on scaled data). On the plot below you can see actual scaled time series (black)and our forecast (blue) for it:

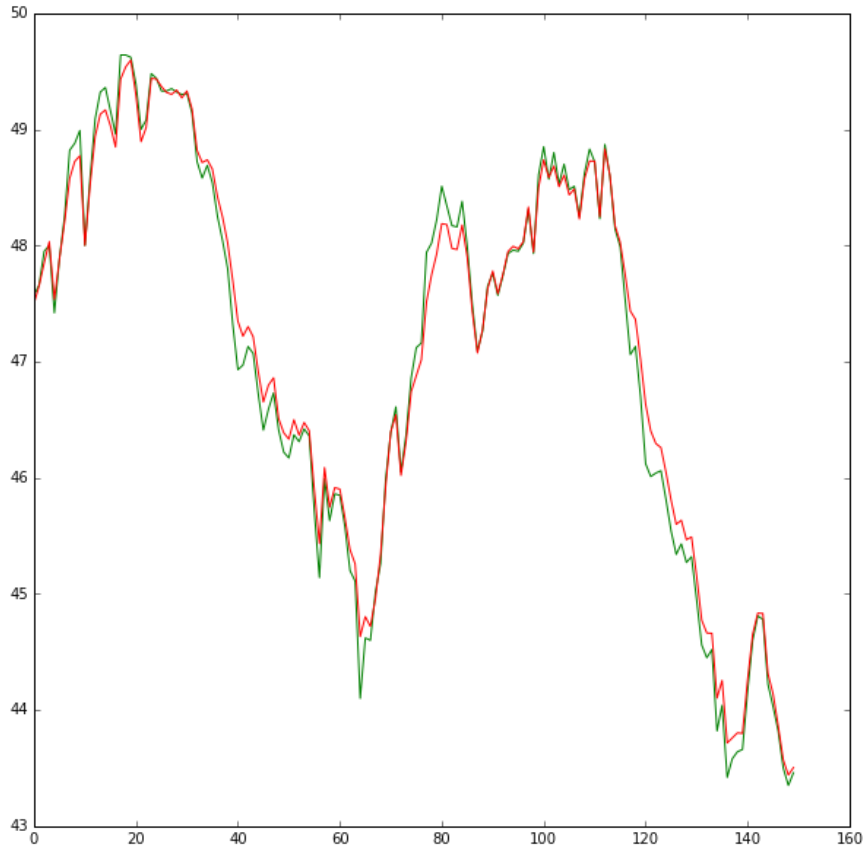


Forecasting results of MLP trained on scaled data, scaled predictions

For using this model in real world we should return back to unscaled time series. We can do it, by multiplying or prediction by standard deviation of time series we used to make prediction (20 unscaled time steps) and add it's mean value:

```
params = []
for xt in X_testp: # taking training data from unscal
    xt = np.array(xt)
    mean_ = xt.mean()
    scale_ = xt.std()
    params.append([mean_, scale_])

predicted = model.predict(X_test)
new_predicted = []

# restoring data
```

MSE in this case equals 937.963649937. Here is the plot of restored predictions (red) and real data (green):



Forecasting results of MLP trained on scaled data, restored predictions

Not bad, isn't it? But let's try more sophisticated algorithms for this problem!

Regression problem. CNN

I am not going to dive into theory of convolutional neural networks, you can check out this amazing resourses:

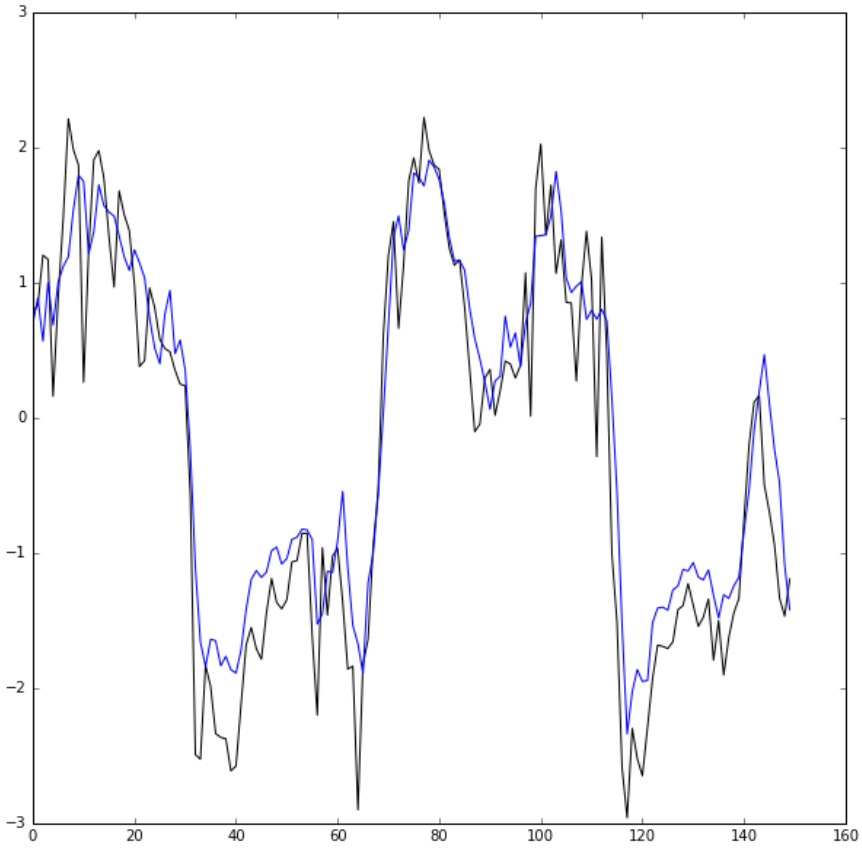
- cs231n.github.io—Stanford CNNs for Computer Vision course
- http://www.wildml.com/2015/12/implementing-a-cnn-for-text-classification-in-tensorflow/—CNNs for text recognition, can be useful for understanding how it works for 1D data

Let's define 2-layer convolutional neural network (combination of convolution and max-pooling layers) with one fully-connected layer

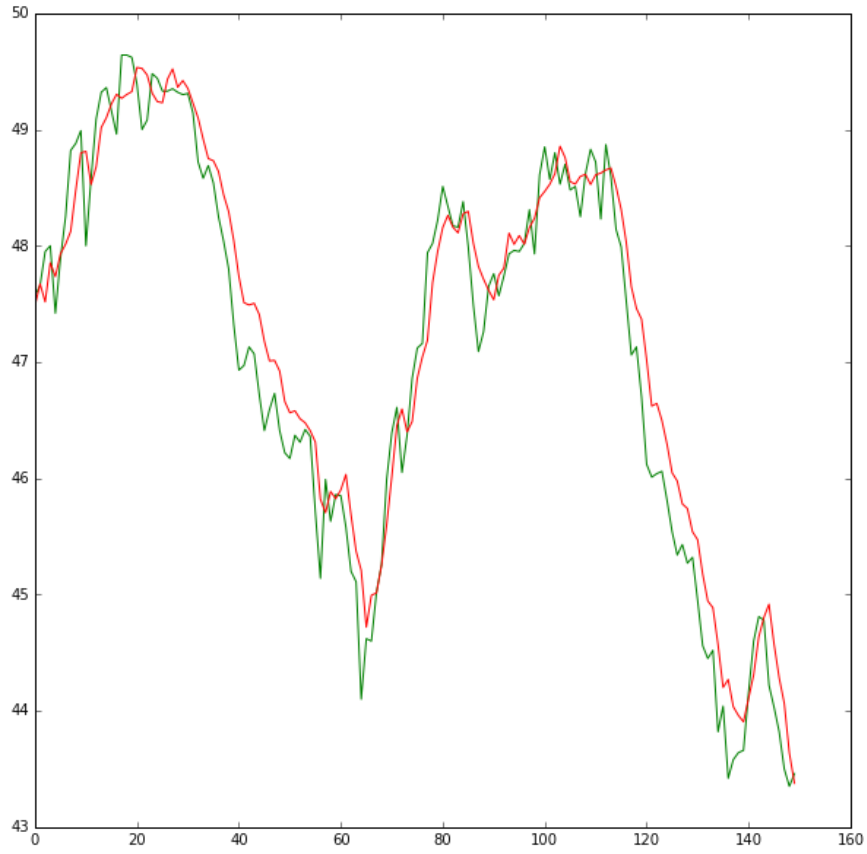
and the same output as earlier:

```
model = Sequential()
model.add(Convolution1D(input_shape = (TRAIN_SIZE, EM
                        nb_filter=64,
                        filter_length=2,
                        border_mode='valid',
                        activation='relu',
                        subsample_length=1))
model.add(MaxPooling1D(pool_length=2))

model.add(Convolution1D(input_shape = (TRAIN_SIZE, EM
                        nb_filter=64,
                        filter_length=2,
                        border_mode='valid',
                        activation='relu',
                        subsample_length=1))
model.add(MaxPooling1D(pool_length=2))

```

Let's check out results. MSEs for scaled and restored data are: 0.227074542433; 935.520550172. Plots are below:



Forecasting results of CNN trained on scaled data, scaled predictions



Forecasting results of CNN trained on scaled data, restored predictions

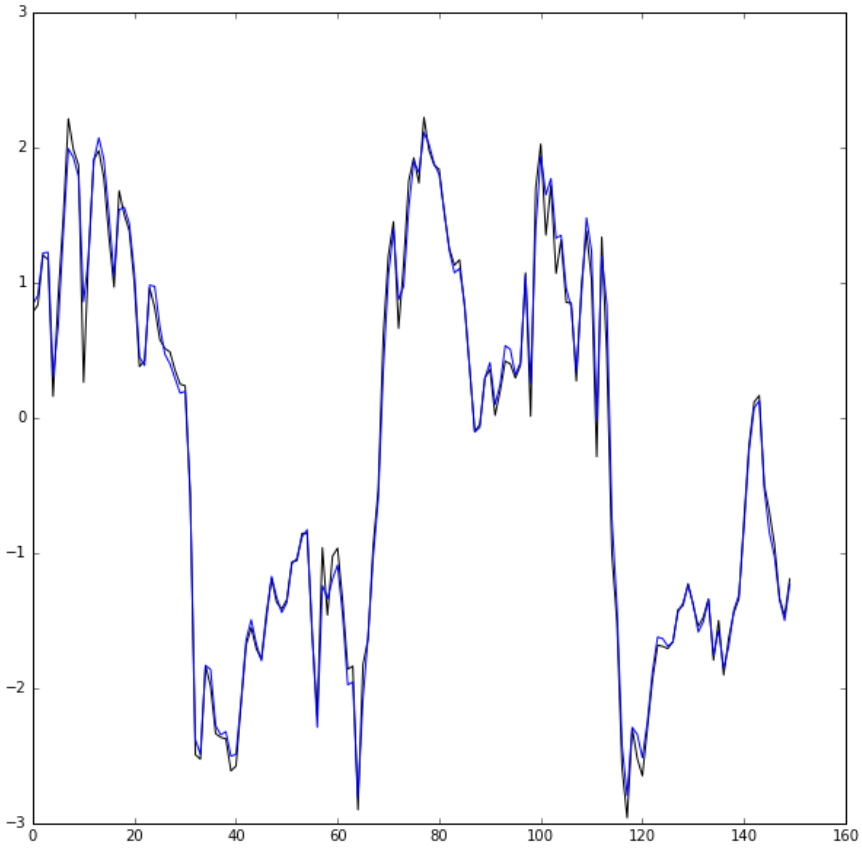
Even looking on MSE on scaled data, this network learned much worse. Most probably, deeper architecture needs more data for training, or it just overfitted due to too high number of filters or layers. We will consider this issue later.

Regression problem. RNN

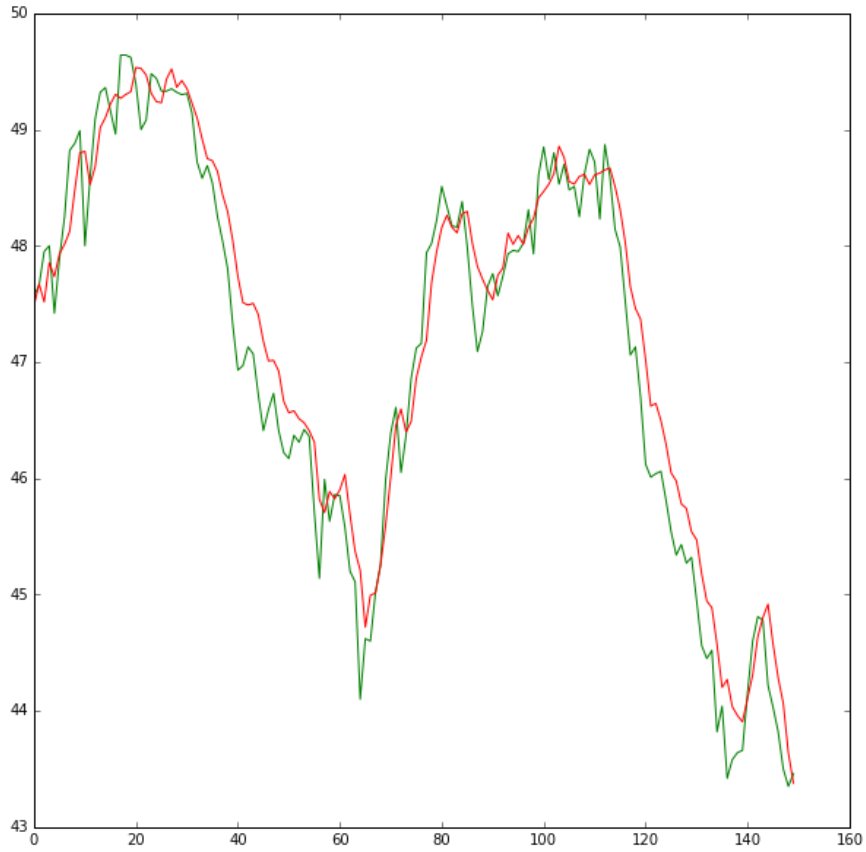
As recurrent architecture I want to use two stacked LSTM layers (read more about LSTMs here).

```
model = Sequential()
model.add(LSTM(input_shape = (EMB_SIZE,), input_dim=EM
model.add(LSTM(input_shape = (EMB_SIZE,), input_dim=EM
model.add(Dense(1))
```

Plots of forecasts are below, MSEs = 0.0246238639582; 939.948636707.



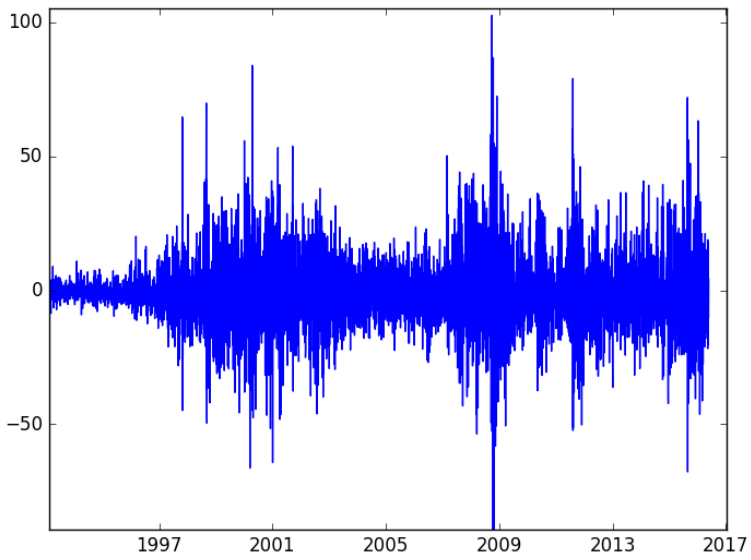
Forecasting results of RNN trained on scaled data, scaled predictions



Forecasting results of RNN trained on scaled data, restored predictions

RNN forecasting looks more like moving average model, it can't learn and predict all fluctuations.

So, it's a bit unexpectable result, but we can see, that MLPs work better for this time series forecasting. Let's check out what will happen if we swith from regression to classification problem. Now we will use not close prices, but daily return (close price-open price) and we want to predict if close price is higher or lower than open price based on last 20 days returns.



Daily returns of S&P 500 index

Classification problem. MLP

Code is changed just a bit—we change our last Dense layer to have output [0; 1] or [1; 0] and add softmax output to expect probabilistic output.

To load binary outputs, change in the code following line:

```
split_into_chunks(timeseries, TRAIN_SIZE, TARGET_TIME,
LAG_SIZE, binary=False, scale=True)
```

```
split_into_chunks(timeseries, TRAIN_SIZE, TARGET_TIME,
LAG_SIZE, binary=True, scale=True)
```

Also we change loss function to binary cross-entopy and add accuracy metrics.

```
model = Sequential()
model.add(Dense(500, input_shape = (TRAIN_SIZE, )))
model.add(Activation('relu'))
model.add(Dropout(0.25))
model.add(Dense(250))
model.add(Activation('relu'))
model.add(Dense(2))
model.add(Activation('softmax'))
```

```
Train on 13513 samples, validate on 1502 samples
Epoch 1/5
13513/13513 [==============================] - 2s - loss:
0.1960 - acc: 0.6461 - val_loss: 0.2042 - val_acc: 0.5992
Epoch 2/5
13513/13513 [==============================] - 2s - loss:
0.1944 - acc: 0.6547 - val_loss: 0.2049 - val_acc: 0.5965
Epoch 3/5
13513/13513 [==============================] - 1s - loss:
0.1924 - acc: 0.6656 - val_loss: 0.2064 - val_acc: 0.6019
Epoch 4/5
13513/13513 [==============================] - 1s - loss:
0.1897 - acc: 0.6738 - val_loss: 0.2051 - val_acc: 0.6039
Epoch 5/5
13513/13513 [==============================] - 1s - loss:
0.1881 - acc: 0.6808 - val_loss: 0.2072 - val_acc: 0.6052
1669/1669 [==============================] - 0s

Test loss and accuracy: [0.25924376667510113,
0.50209706411917387]
```

Oh, it's not better than random guessing (50% accuracy), let's try something better. Check out the results below.

Classification problem. CNN

```
model = Sequential()
model.add(Convolution1D(input_shape = (TRAIN_SIZE, EM
                        nb_filter=64,
                        filter_length=2,
                        border_mode='valid',
                        activation='relu',
                        subsample_length=1))
model.add(MaxPooling1D(pool_length=2))

model.add(Convolution1D(input_shape = (TRAIN_SIZE, EM
                        nb_filter=64,
                        filter_length=2,
                        border_mode='valid',
                        activation='relu',
                        subsample_length=1))
model.add(MaxPooling1D(pool_length=2))

model.add(Dropout(0.25))
model.add(Flatten())

model.add(Dense(250))
```

```
Train on 13513 samples, validate on 1502 samples
Epoch 1/5
13513/13513 [==============================] - 3s - loss:
0.2102 - acc: 0.6042 - val_loss: 0.2002 - val_acc: 0.5979
Epoch 2/5
13513/13513 [==============================] - 3s - loss:
0.2006 - acc: 0.6089 - val_loss: 0.2022 - val_acc: 0.5965
Epoch 3/5
13513/13513 [==============================] - 4s - loss:
0.1999 - acc: 0.6186 - val_loss: 0.2006 - val_acc: 0.5979
Epoch 4/5
13513/13513 [==============================] - 3s - loss:
0.1999 - acc: 0.6176 - val_loss: 0.1999 - val_acc: 0.5932
Epoch 5/5
13513/13513 [==============================] - 4s - loss:
0.1998 - acc: 0.6173 - val_loss: 0.2015 - val_acc: 0.5999
1669/1669 [==============================] - 0s
Test loss and accuracy: [0.24841217570779137,
0.54463750750737105]
```

Classification problem. RNN

```
model = Sequential()
model.add(LSTM(input_shape = (EMB_SIZE,), input_dim=EM
model.add(LSTM(input_shape = (EMB_SIZE,), input_dim=EM
model.add(Dense(2))
model.add(Activation('softmax'))
model.compile(optimizer='adam',
```

```
Train on 13513 samples, validate on 1502 samples
Epoch 1/5
13513/13513 [==============================] - 18s - loss:
0.2130 - acc: 0.5988 - val_loss: 0.2021 - val_acc: 0.5992
Epoch 2/5
13513/13513 [==============================] - 18s - loss:
0.2004 - acc: 0.6142 - val_loss: 0.2010 - val_acc: 0.5959
Epoch 3/5
13513/13513 [==============================] - 21s - loss:
0.1998 - acc: 0.6183 - val_loss: 0.2013 - val_acc: 0.5959
Epoch 4/5
13513/13513 [==============================] - 17s - loss:
0.1995 - acc: 0.6221 - val_loss: 0.2012 - val_acc: 0.5965
Epoch 5/5
13513/13513 [==============================] - 18s - loss:
0.1996 - acc: 0.6160 - val_loss: 0.2017 - val_acc: 0.5965
1669/1669 [==============================] - 0s
Test loss and accuracy: [0.24823409688551315,
0.54523666868172693]
```

Conclusions

We can see, that treating financial time series prediction as regression problem is better approach, it can learn the trend and prices close to the actual.

What was surprising for me, that MLPs are treating sequence data better as CNNs or RNNs which are supposed to work better with time series. I explain it with pretty small dataset (~16k time stamps) and dummy hyperparameters choice.

You can reproduce results and get better using code from repository.

I think we can get better results both in regression and classification using different features (not only scaled time series) like some technical indicators, volume of sales. Also we can try more frequent data, let's say

minute-by-minute ticks to have more training data. All these things I'm going to do later, so stay tuned :)