

# Convolutional Networks for Stock Trading

Ashwin Siripurapu  
Stanford University Department of Computer Science  
353 Serra Mall, Stanford, CA 94305  
ashwin@cs.stanford.edu

## Abstract

*Convolutional neural networks have revolutionized the field of computer vision. In these paper, we explore a particular application of CNNs: namely, using convolutional networks to predict movements in stock prices from a picture of a time series of past price fluctuations, with the ultimate goal of using them to buy and sell shares of stock in order to make a profit.*

## 1. Introduction

At a high level, we will train a convolutional neural network to take in an image of a graph of time series data for past prices of a given asset (in our cases, SPY contracts traded on the NYSE). Then, we will predict the movement of the price in the next few minutes. If the CNN correctly predicts price movements, we can make money by buying when the CNN says the price will go up in the future, and then selling it at the higher price in a few minutes' time.

We evaluate the trained network both using traditional statistical performance measures (viz.,  $R^2$ ) and also with a “paper trade” simulator that enables us to see what would have happened if we had bought and sold contracts according to the CNNs predictions; in particular, we can see how profitable the strategy of following the trained CNN would be. Naturally, this methodology is subject to the vulnerability that it is impossible to tell how other participants in the market would have reacted to the presence of the CNN's buying and selling, but it does give us at least some measure of confidence as to the CNNs abilities as a trader.

## 2. Problem Statement and Technical Approach

### 2.1. Gathering Data

The first step in the process of training a CNN to pick stocks is to gather some historical data. [1] provides minute-by-minute ticker data on the S&P 500 ETF Trust (symbol: SPY), traded on the NYSE. Specifically, for each

minute of each trading day, we have the data listed in Table 1.

### 2.2. Initial Choice of Features

Since the project requires us to use pixel data, I had to convert this price data into images. This presents an interesting challenge in its own right, but a very obvious starting point is to take as our inputs (features) a graph of the price of the contract for some period of time into the past (say, 30 minutes back) and then use that to predict the price at some time in the future (say, 5 minutes ahead). Then, if we predict that the price will go up (down), we will buy (sell) in the present and sell (buy) in 5 minutes to acquire a profit.

Firstly, what do we mean by “the” price of the contract? Recall from above that Google Finance provides us with four separate prices for each minute of the trading day. For the time being, I have elected to use only the high and low prices within a given minute, since these implicitly bound the other two prices (open and close). Moreover, the high and low intuitively contain more information than the open and close prices, because the open and close prices are in a sense statistical artifacts: they are the prices that the market happened to be at at the time that the price series was sampled by Google (or whomever was collecting the data).

Secondly, how far in the past should our time series graph go? This is in principle another hyperparameter that should be tweaked once the convolutional network has been set up, but for now, I have gone with a 30-minute window into the past.

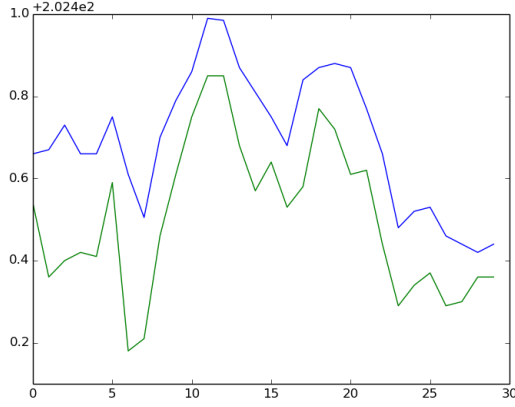
In conclusion, the inputs to the model are images of the graph of high and low prices for 30 minute windows of time. These images are drawn using the `numpy` and `matplotlib` libraries and are saved as RGB images. An example input is shown below in Figure 2.2.

Later on, I experimented with using slightly different

Column Name	Meaning
DATE	Time (which minute of the day)
CLOSE	Closing price (price at the end of the minute)
HIGH	High price (maximum price during the minute)
LOW	Low price (minimum price during the minute)
OPEN	Opening price (price at the beginning of the minute)
VOLUME	How many contracts were offered to be bought/sold in the minute

Table 1. Minute-by-minute data provided by [1]

Figure 1. An example picture input to convolutional network. High prices are in blue, low prices are in green.



features (see Section 6).

### 2.3. Choice of Response

I mentioned earlier that we would use a 30-minute window of past price data (in image form) to predict the price of the asset at some point in the future. We will now explore the intricacies involved in choosing the response variable to predict.

In the first place, as with the features, we have a choice of how far into the future to predict the price fluctuation. In practice, we want this length of time (hereafter called the *prediction horizon*, or just *horizon* for short) to be short, because as time goes on, any predictive signal that we have gets increasingly drowned out by random noise in the market. At the same time, we cannot make the horizon too short, or else we will not be able to execute effectively (i.e., buy or sell) on the basis of our predictive signal; carried to the extreme, it is trivial to predict the price one picosecond into the future with 100% accuracy (since prices cannot change faster than the speed of light), but this prediction is utterly useless for the purposes of making money!

For the time being, I have decided to use a horizon of 5 minutes past the end of the window of past prices that constitutes an input to the model (explained above).

In addition to the choice of horizon, we also have to decide exactly what quantity we want to predict. There is a wide body of financial literature dedicated to this topic. An obvious choice is to try and predict the price itself at the end of the horizon period (i.e., given a window of price data for 30 minutes into the past, from time  $t - 30$  to time  $t$ , try to predict the price at time  $t + 5$  if the horizon is 5 minutes). **However, it is more useful to predict the change in price between time  $t$  and time  $t + 5$ , as this is ultimately what we'll determine whether it is profitable to buy (sell) now and sell (buy) in 5 minutes in order to lock in a profit.**

Moreover, there are multiple ways to measure the difference in price at two times  $t_1 < t_2$ . In the first place, we have to decide which price (high, low, open, or close) we want to predict. Having decided on some measure of price, and obtained price measurements  $p_1$  and  $p_2$  at times  $t_1$  and  $t_2$  respectively, there are two widely-accepted ways of calculating the change in price (also known as *return*) in finance: the arithmetic return, defined as

$$r_1 = \frac{p_2 - p_1}{p_1},$$

and the logarithmic return, defined

$$r_1 = \log\left(\frac{p_2}{p_1}\right).$$

I have chosen to define the price of the asset (for the purposes of defining a response variable) to be the mean of the high and low prices during the minute that we are considering. In addition, I have chosen to use log returns rather than arithmetic returns.

**Concretely, the regression problem is: given as input an image of high and low prices from time  $t - 30$  to time  $t$ , predict as output the logarithmic return from time  $t$  to time  $t + 5$ , i.e., predict**

$$\log\left(\frac{p_{t+5}}{p_t}\right),$$

where  $p_i$  denotes the mean of the high price and low price in minute  $i$  of the trading day.

## 2.4. Choice of Loss Function

I am going to use an  $\ell^2$  loss function when training the convolutional network. In future, we can consider different choices of loss function, but the  $\ell^2$  loss is very standard in regression problems in finance. Moreover, Caffe readily supports  $\ell_2$  regression with its `EUCLIDEAN_LOSS` layers.

It is important to note that, unlike the length of the input window, the choice of loss function is *not* a hyperparameter to be tuned. This is because different loss functions are different problems entirely, not merely different solutions to the same problem. Different loss functions correspond to different notions of the “displeasure” or “dissatisfaction” with our predictions that we are trying to minimize. It makes no sense to argue that one setting of parameters is “better” than another when the comparison is across different loss functions.

That said, in trading, the ultimate test of how good a strategy or model is is how much money it makes. In that sense, and in that sense alone, it may make sense to experiment with different loss functions to derive different optimization problems, and then see which optimization problem yields the most profitable strategy.

## 3. Establishing a Baseline

The most basic financial model is ordinary least-squares regression (OLS). For purposes of establishing a baseline for performance, I used this model on a very simple set of features.

Concretely, I took the  $600 \times 800$  time series graph images and scaled each one down to a  $32 \times 54$  thumbnail image. In addition, I converted the images from four channels (RGBA) to one (grayscale). The thumbnails then corresponded to points in the space  $\mathbb{R}^{1728}$ .

Treating each grayscale thumbnail and its corresponding log return as a training pair  $(x_i, y_i)$ , I then fit a linear model to a training data set of 4000 points and tested it on a data set of 996 points.

The within-sample  $R^2$  of the linear model was 0.428, which is fairly impressive for such noisy data. However, the ultimate test of any statistical model is how it performs out of sample. The out-of-sample  $R^2$  for this linear model on the test set was an embarrassing  $-12.2$ . Clearly no one should use this model to trade on the market, unless he wants to lose a lot of money!

It should be possible for the final convolutional network to beat these results easily. In the first place, the baseline

model used (OLS) was extremely simple. Secondly, the features (pixel data) bore little linear structure that could have been exploited to predict log returns well. A convolutional network with many nonlinearities can rectify this (no pun intended). Finally, the feature space used in this OLS baseline was heavily reduced: we shrunk the images to thumbnails and removed all color information. Given the full input data, a CNN should be able to do significantly better.

Ideally, we should be able to get  $R^2 > 0$  on an out-of-sample test set. This means that we are doing better than the naive strategy of always guessing that the log return in the next 5 minutes will be the mean log return in the test set (usually around 0). If we can do this regularly, then provided we have good execution (ability to buy and sell reasonably quickly), we have the makings of a profitable trading strategy.

## 4. Workflow

In the following sections, I describe how I systematically made changes to the network architecture, to the hyperparameters, and to the features (images) that were put into the model. Concretely, my workflow was as follows:

1. Generate features from the data using `matplotlib`.
2. Convert image features and log return response into HDF5 using `hdf5_convert.py`.
3. Generate network architecture file using [4], a script provided by a fellow student on Piazza.
4. Tune hyperparameters by modifying `solver.txt`.
5. Train network using Caffe.
6. Visualize weights in trained network using `visualize_weights.py`.
7. Evaluate network by computing out-of-sample  $R^2$  with `caffe_compute_r2.py`.

## 5. Hyperparameter Tuning

The first thing that I did to achieve lower loss (hence higher  $R^2$ ) was to tweak the optimization hyperparameters, as specified in the `solver.prototxt` file. This includes the starting learning rate, the learning rate update scheme and parameters, and the type of solver (SGD, Adagrad, or NAG [Nesterov accelerated gradient]). I started out with 10,000 training iterations, with momentum SGD.  $\alpha$  started out at 0.01 and was cut down by a factor of  $\gamma = 0.1$  every 5,000 iterations (i.e., `step_size` was set to 5,000). In addition, the momentum term was set to  $\mu = 0.9$ .

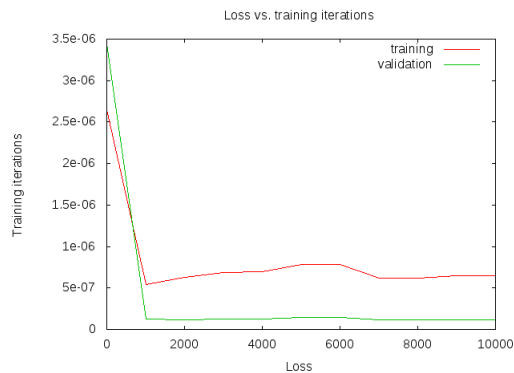


Figure 2. Training and validation loss with SGD,  $\alpha_{\text{init}} = 0.2$ ,  $\mu = 0.9$ ,  $\gamma = 0.5$ ,  $\text{step\_size} = 2000$

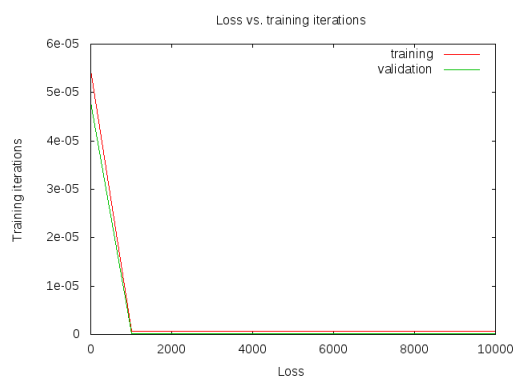


Figure 3. Training and validation loss with NAG

This was far too low a learning rate, and too low a rate of annealing. As a result, training loss hardly moved from its initial value and validation loss remained fairly flat, too.

I decided to increase the mobility of the optimization hyperparameters by increasing the initial learning rate, increasing the value of  $\gamma$ , and decreasing the step size (so  $\alpha$  would be updated more frequently). Concretely, I set the initial learning rate to 0.2,  $\gamma$  to 0.5, and  $\text{step\_size}$  to 2000.  $\mu$  remained at the original value of 0.9. This resulted in the training and validation loss plot shown in Figure 2.

Following this, I decided to experiment with Nesterov's accelerated gradient. To do this, I simply added the line `solver_type: NESTEROV` to the solver file. This resulted in the training and validation loss depicted in Figure 3. This did not significantly improve over momentum SGD loss.

When I switched to using different network architectures and different features (see below), I had to update the hyperparameters in `solver.prototxt` appropriately. Nonetheless, the same basic approach (come up with some hyperparameters, run the network, plot the training and validation loss curves) proved useful and, in fact, the

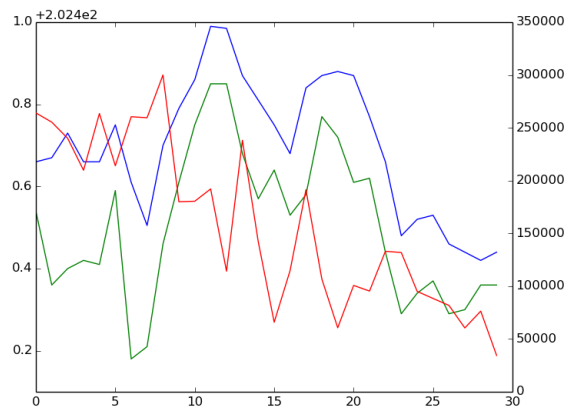


Figure 4. An example image input. As before, high prices in blue, low prices in green. Volume (right axis) in red.

same hyperparameter settings were generally near-optimal with slight modifications.

## 6. Feature Engineering

Recall from Figure 2.2 what a typical input price window image looks like. After the poster session, some commentators suggested a better choice of inputs. In particular, my image inputs did not use the red channel to encode any data at all. The red channel could have been put to better use, for example, by using it to store data about the average of the low and high prices, or the volume at each minute of the trading day<sup>1</sup>. Others suggested that I use a different visualization in the image data: rather than plotting the absolute price at each time for a short window, I could instead plot a spectrogram and visualize the price data in the frequency domain.

Ultimately, I experimented with two more kinds of inputs. The first one was similar to the original image data in that it used a time-domain representation of the price series, except that I also used volume data, which was plotted in red on a separate set of axes. An example of this kind of input is shown in Figure 4.

The other kind of representation that I tried was the so-called correlation features. Recall that the S&P 500 is a weighted basket of 500 different individual stocks (equities). That is, owning a single unit (share) of SPY is equivalent to owning some number of shares of each of the 500 constituent corporations. The ten companies which comprise the biggest share of the S&P 500 basket are shown in

<sup>1</sup>Recall that volume is the total quantity of contracts available to be bought or sold in a given minute. In actual trading scenarios, this is usually expressed as two numbers (number of contracts available for sale, and number available for purchase), but Google Finance's data added the two together and expressed them as a single sum.

Company	Symbol	% Assets
Apple Inc.	AAPL	4.03
Exxon Mobil Corporation Common	XOM	2.01
Microsoft Corporation	MSFT	1.93
Johnson & Johnson Common Stock	JNJ	1.54
Berkshire Hathaway Inc Class B	BRK.B	1.44
General Electric Company Common	GE	1.40
Wells Fargo & Company Common St	WFC	1.38
Procter & Gamble Company (The)	PG	1.23
JP Morgan Chase & Co. Common St	JPM	1.23
Pfizer, Inc. Common Stock	PFE	1.16

Table 2. Top 10 components of the S&P 500. Data from [2]

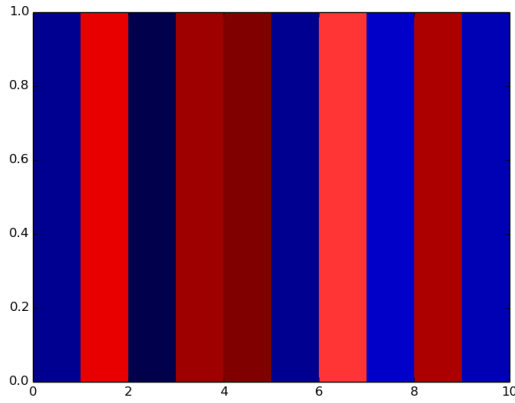


Figure 5. An example image input. There are ten color swatches; each represents the correlation of SPY with a different stock from among the top 10.

Table 2.

Collectively, these top ten comprise 17.53% of the S&P 500 basket. I used these 10 constituent components to compute inputs to the neural network in the following way: within each 30 minute window, I gathered the average price (mean of low and high) time series for the S&P 500 and for each of the top ten constituents. Then, I computed the correlation coefficient between the price series for the SPY and the price series of each of the constituents. This resulted in ten different values between -1 and 1. I then plotted these ten values in a heatmap, with colors ranging from red (high correlation) to blue (low correlation). An example feature image is shown below. The hope is that, when the S&P 500 is trending upwards, its correlations with its constituent stocks will exhibit a different pattern than when it is trending downward.

## 7. Network Engineering

The first network structure that I tried repeated convolution-ReLU-pool layers (5 blocks) followed by a

fully-connected layer. Call this the *original architecture*. Unfortunately, this led to rather poor performance: out of sample  $R^2$  was  $-0.06$ , slightly worse than the strategy of “always guess that the log return will be the mean log return (approximately 0)” which would achieve an  $R^2$  of 0.

Following Yuke’s advice, I visualized the weights in the last convolution layer in the initial network structure to see what the problem might be. The problem was that there has been too much pooling, so that the input to the final layers was very blurry, and the final layers of the network (in particular the affine layer) could not distinguish between distinct images when they are so blurred.

To remedy this, I tried a new network structure which didn’t employ pooling until several cycles of convolution-ReLU. Concretely, I used three blocks of conv-ReLU, followed by two blocks of conv-ReLU-pool, followed by a single affine layer. Let us call this the *reduced architecture* since it has fewer layers than the original architecture. This resulted in somewhat improved performance; results are shown in Table 3. The weights of this new network structure, when trained on the windows of price and volume data features from above (as in Figure 4) are shown below in Figure 6 and Figure 7. Clearly some of the structure of the price and volume charts is being captured by the weights in the last layer, i.e., the image is not getting blurred too badly as it passes through the network.

In addition, I visualized the final convolution layer weights for the same (reduced) architecture network trained on the heatmap features. This data appears in Figure 8. The influence of the vertical stripes of colors is clearly visible.

## 8. Results

For each of the two architectures (the original one, which resulted in blurring of the image by the final layers; and the reduced architecture) and each of the two new types of features, we trained the network, tweaked the hyperparameters until convergence was achieved in at most 10,000 iterations,

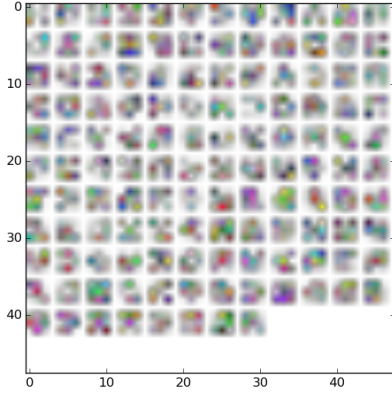


Figure 6. The weights of the first layer of the reduced architecture network after training on price and volume features.

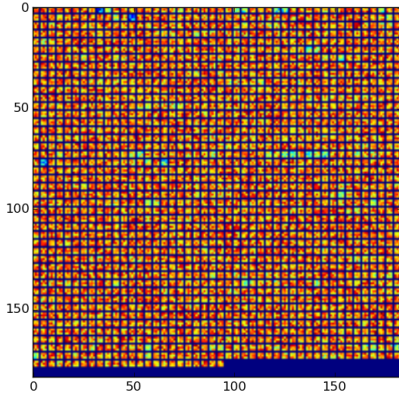


Figure 7. The weights of the last convolution layer of the reduced architecture network after training on price and volume features.

	Original arch.	Reduced arch.
Price windows with volume	−0.05	−0.014
Correlation features	−0.42	−0.013

Table 3. Out-of-sample  $R^2$  for different features, architectures

and computed an out-of-sample  $R^2$  on a test set of data. The results of that process are shown in Table 3.

Overall the results are underwhelming. We were unable to achieve an out-of-sample  $R^2$  greater than 0; that is to say, you would be better off guessing that the next price movement is going to be the mean price movement (typically 0) than following our model. Nonetheless, this was a good learning experience and

## 9. Further Work

It remains to be seen whether  $\ell_2$  regression is the best formulation of this finance problem as a machine learning

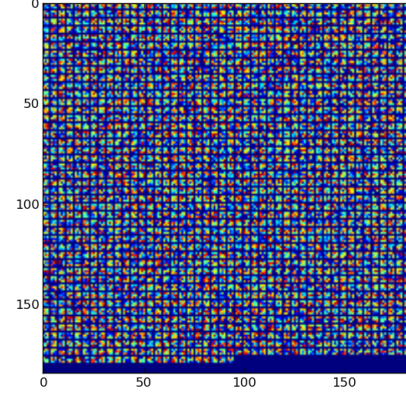


Figure 8. The weights of the last convolution layer of the reduced architecture network after training on heatmap features.

problem. I suspect that using a classification-based approach may be better in practice, especially in the high-frequency setting, because in reality, price movements are discrete (prices fluctuate in ticks no smaller than a penny).

Future work will also focus on the execution side of this problem, once  $R^2$  is positive; that is, once we have a better-than-chance predictive edge, it remains to be seen whether this statistical predictive ability can be translated into actual cash by executing the strategy optimally (buying and selling when the strategy says to).

Lastly, it remains to be seen whether image features are best. Naturally we could feed historical price, volume, correlation, or any other data into a neural network directly, instead of encoding it as an image. This would get the data into the network more directly than via the circuitous route that the current implementation takes.

## 10. References

### References

- [1] <http://www.google.com/finance/getprices?i=60&p=20d&f=d,o,h,l,c,v&df=cpct&q=SPY>
- [2] <http://finance.yahoo.com/q/h1?s=SPY+Holdings>
- [3] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.
- [4] Das, Subhasis. `gen_proto.py`, a Python script for generating Caffe network architecture files.