Neural Networks for Learning Stock Market Features

Dan Schmidt
Department of Mathematics
Harvey Mudd College
Claremont, CA 91711
dschmidt@hmc.edu

Abstract

Predicting stock market prices from observed data is a highly challenging but potentially lucrative task. The usual procedure for finding profitable patterns is to have a hypothesis about how the market works, and then to engineer features that model this hypothesis. In this paper, I investigate methods for exctracting potential features from stock price and volume data. First, both linear (PCA) and non-linear (autoencoder) dimensionality techniques are applied to windowed data. Regularization techniques and Bayesian analysis are applied to reveal different sets of features. Then a layered supervised network is trained against future returns to try to find features that correspond to predictive power. In each case, a GPU is used to train millions of data points effectively. I find that the supervised neural network trained on price and volume data could be used for what is in theory a profitible trading strategy.

1 Introduction

Note: my code lives at `https://github.com/thedanschmidt/unsupervised-stock-features`, and the data is at `://drive.google.com/drive/folders/0B3w8qRTObMYecHA3YmtaY184ZlE?usp=sharing`

The hot machine learning topic of the moment is so called deep-learning. The idea with deep learning is not to improve models by hand-engineering features but by turning loose a large amount of computational power on neural networks with many hidden layers. Layers learn a hierarchy of features that have large predictive power for the goal the network is trained against. My approach here is not to use the most cutting-edge or trendy deep learning techniques but to use a deep learning software system, Keras with a Theano backend, to try basic feature learning on stock market data.

Neural networks are some of the most powerful learning algorithms known due to their non-linear nature and ability to learn any continuous function. For complciated non-linear regression problems, neural networks serve as a highly efficient function basis. The challenges of using neural networks are that they are difficult to interpret, resistant to simple probabilistic interpretation, and difficult to train without overfitting. These downsides can be mitigated by feeding them a lot of data. To do this, one needs both an abundance of data and a computing system capable of processing that much data in reasonable time frames. For my data, I use a couple years worth of minute-resolution stock price and volume data provided by PiTrading.com. For my efficient computation system, I train my networks on a GTX 750ti, a GPU that is a couple of years old but still is orders of magnitude faster to perform the matrix operations than my CPU.

In this paper, I first describe the data used and how it was preprocessed. I then discuss applying a linear model, PCA, to try and reduce the dimensionality of windowed data. Then a non-linear method, an autoencoder, is trained to replicate windowed data in a lower dimensional space. These techniques are unsupervised methods to try and find patterns without any regard to predictive power.

30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

Following these unsupervised techniques, I describe a supervised technique to extract price and volume patterns that most correspond to predicting future returns. This model, while lacking in explanatory power, seems to have some predictive power as a simple training strategy executed with this model is profitable out-of-sample for 24 of 30 Dow Jones Industrial average tickers. I then wrap up by describing the tips and tricks my analysis has revealed for applying neural networks to learn stock market data.

2 The Data

2.1 Format and Preprocessing

The data is minute-resolution price and volume data on Down Jones Industrial Average stock tickers. The data is provided in CSV format from PiTrading. This raw data is then processed out of CSV files into Python pandas dataframes, which are saved in a binary format to disk for faster processing later.

A major note is that there are many missing price dates, and that the data also may contain survivor bias. These issues are mitigated in this analysis by selecting tickers and time frames that have mostly (95%) data and using only relatively short time frames over which established companies are trading in a normal environment.

2.2 Returns: Working in Percent Return Space

One important starting point for analyzing price data is to work in return space. This is because the raw dollar value of a stock doesn't contain much information, and because modeling price is more difficult due to the fact that it is autoregressive. The data used in this paper does not have market cap information, and so the prices cannot be weighted that way. That leaves three basic options:

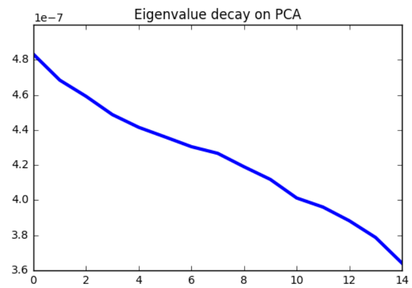
1. Return space, i.e.
$$r_t = \frac{X_t - X_{t-1}}{X_{t-1}}$$
2. Log return space, i.e.
$$l_t = \log\left(1 + r_t\right) = \log\left(\frac{X_t}{X_{t-1}}\right) = \log(X_t) - \log(X_{t-1})$$
3. Normalized returns
$$r_t = \frac{1}{\hat{\sigma}(X)} \frac{X_t - X_{t-1}}{X_{t-1}}$$
where we divide by the estimated standard deviation.

Return space is convenient in that the results are easily interpretible, as in they are just fractional movements in price. The reasons to prefer log return space are that often returns are not normally distributed, but the log of the returns is, and that log returns have nice computational properties where you can accumulate with sums rather than cumulative products. Normalized returns are mostly useful for doing market basket analysis, as some stocks may have more volatility than others. Most of the following analysis was done with percent returns, sometimes normalized by standard deviation for neural network processing.

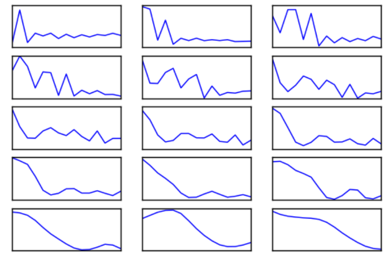
3 Linear Dimensionality Reduction: PCA

An initial idea for improving model performance would be to reduce the dimensionality of the data before the model is fit. One way to do this would be to see if windowed hours can be projected onto a linear subspace. We try a PCA analysis on sliding window price data to see if this has any viability as a dimensionality reduction strategy. The results are shown in figure 1b.

It turns out this projection is not very useful here. First, the eigenvector corresponding to the principal component does not seem to carry more information than any of the less significant eigenvectors. Second, the decay in the eigenvalues is much slower than what we would expect if PCA discovered underlying linear structure. Part of the problem with this analysis is that for a windowed time series,



(a) Decay in the eigenvalues of PCA



(b) The eigenvectors of PCA.

one would expect that features should be somewhat invariant in time. Thus any structure found here would have to persist in sliding through the windows, which it looks like is not the case. We need a model that has more knowledge of the structure of the data.

4 Non-linear: Neural Nets

Another way to reduce the dimensionality for prediction is to let the neural network learn features itself by stacking hidden layers into the network. These approaches are described in Bishop [1]. This is the approach used here.

4.1 Unsupervised Approach: Autoencoder

Although PCA did not seem to recover a useful linear projection in the data, it is possible that there is some non-linear relationships. The technique tried here for to find this structure is to train an autoencoder, a neural network that learns the identity function. The idea of an autoencoder is that there are at least two non-linear layers between the input vector and the output vector. The first layer maps the input vector to some lower dimensional subspace. This lower dimensional subspace hopefully is a more meaningful representation of the data, and can be used to find interesting non-linear patterns.

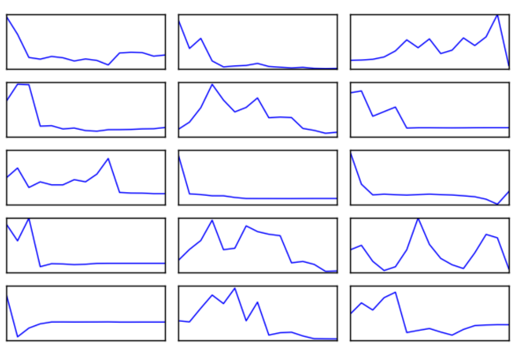


Figure 2: Extracted 15-minute window representations from a a year with of minute price movements on AAPL. These features are the weights on the first layer of a four-layer network, accumulated so that they represent stock-chart price movements.

The results of training an autoencoder are shown in 2. This particular autoencoder had four layers, two non-linear encoding layers, followed by a non-linear decoding layer and a final linear layer that scales the output of the decoded layer back to represent the window. The 15 minute windows were compressed to 10 dimensional space, a number chosen from running different dimension models on a validation set. The objective function was mean square error, trained with an adaptive stochastic gradient descent algorithm ran in batches on about 100,000 price windows for AAPl. There seem to be some interesting learned patterns, such as large movements early in the window that lead to dampened movements later. This visualization is only of the first layer in the data set, and later layers form non-linear combinations of these earlier features.

In **??**, the same autoencoder was trained except now with an $l1$ regularization parameter of .001. $l1$ regularizing, also called LASSO, enforces a sparsity constraint on features. The hope is that the new objective function optimizes away from noise and finds simpler features. It is interesting how the algorithm now seems to pick out a small set of discrete price jumps to form the first layer of its features.

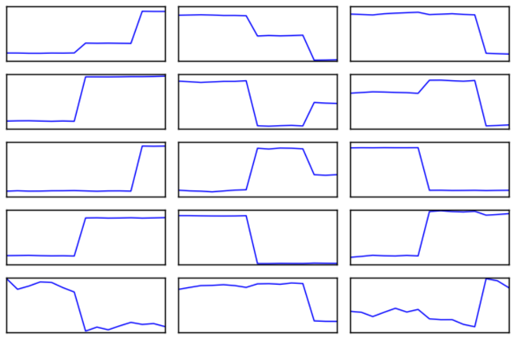


Figure 3: Same network trained on the same data as 2, except now $l1$ regularized to enforce sparsity. Note the more jagged learned features, as flat lines are due to the sparse feature construction

4.2 Supervised Approach: Windowed Multi-Layer Perceptron

There are various network architectures one could use to predict time series with neural networks. The model here is a standard multi-layer perceptron as described in Bishop [1].

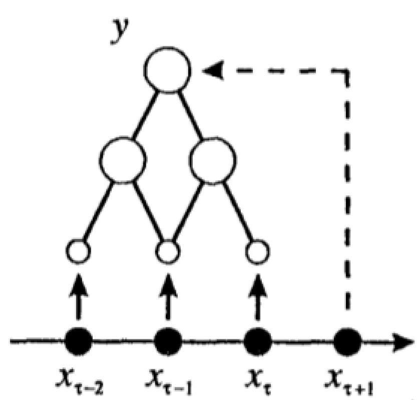


Figure 4: Diagram of the windowed network to predict one step forward in the time series.

As shown in figure 4, the idea is that the time series is windowed into sections. These sections are then used as features for predicting the next element in the time series. The layers stacked in between are nonlinear combinations designed to learn latent features in the data that predict the outcome. The network is trained by sliding a window over a long period of stock data and minimizing the error in predicting one time step forward.

With the autoencoder, the network was set to learn a set of non-linear features that best could replicate any window of returns. By training the network against future returns, we look more for price (and volume) features that best predict forward returns. This more difficult training objective may find different features than an autoencoder.

The learned features on the same data set as PCA and the autoencoder is shown in figure 5.

It is interesting to see how these features seem to have much less interpretible structure than the autoencoder. One part of this is that this network was also trained with volume data, and so more

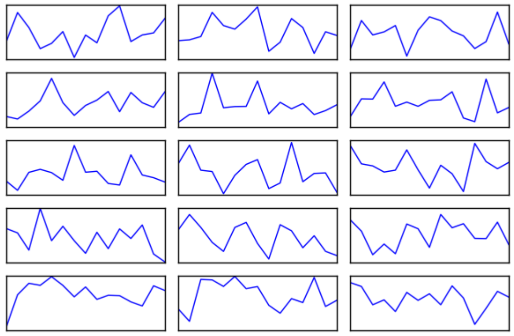


Figure 5: Features learned in the supervised network

of the predictive power may be in a combination of price/volume data which is more difficult to visualize. Another explanation is that this network has not really converged to its optimal point yet. We will see later when this network is trained on millions of samples, it learns more identifiable price features.

5 Bayesian Analysis of The Network

Bayesian methods can be very helpful in training and analyzing neural networks. One of my goals for this project was to use these Bayesian methods to help train better networks. Unfortunately, the hyperparameter estimation worked better by hand and I was unable to effectively compute the Hessian of my network as is needed by the method to analyze error in predictions. But here I lay out the theoretical basis for how Bayesian techniques can be useful for neural network training and evaluation. These are mostly learned from the logistic regression example in class and Bishop [1].

Assume our model has weights w and hyperparameters γ. Our task is to learn an approriate function $p(y|x, w, \gamma)$, so we seek to optimize

$$argmax_{w,\gamma} p(w, \gamma|D)$$

The frequentist approach to fitting this function is to choose a set of possible hyperparameters and search through this parameter space optimizing the weights each time. The results are then checked on a validation set, and the optimal combination of parameters are chosen this way. The problem with this approach is that for neural networks, the hyperparameter space can be huge and so difficult to effectively search. In addition, unless one has a lot of data, it is difficult to not overfit either the training set or validation set. Finally, once one has a model, it is difficult to a sense of what the error bars should be on the models predictions. Approaching this problem with a Bayesian perspective provides some ways to get around these problems.

The first step to using Bayesian techniques on neural networks is to choose a prior on the weights. Here for simplicity we use a Gaussian model

$$p(w) = \frac{1}{Z_W(\alpha)} \exp\left(-\frac{\alpha}{2}||w||^2\right)$$

where α is a hyperparameter and Z is just a normalization constant. Another choice to make is the distribution of error. Here again we use a Gaussian model,

$$p(y|x, w) \propto \exp\left(-\frac{\beta}{2}(f(x, w) - y)^2\right)$$

5.1 Evaluating Error Bars

Assume we have optimal choices for our hyperparameters. We can use the distributions we have estimated so far to get an expression for error bars on the output of our regression. To do so, we wish to compute the integral

$$p(y|x, D) = \int p(y|x, w)p(w|D)dw$$

$p(y|x, w)$ is our model for distribution of noise, and $p(w|D)$ is the model for the weights given the data. The approximation we use is the same approximation given in section 8.4 of Murphy [2]. We approximate this integral with a linear expansion around $f(x; w_{MAP})$. If

$$g = \nabla_w y|_{w_{MAP}}$$

and if H is the Hessian matrix, then

$$p(y|x, D) = \frac{1}{2\pi\sigma_t^2} \exp\left(-\frac{(y - f(x; w_{MAP}}{2\sigma_t^2}\right)$$

where

$$\sigma_t = \frac{1}{\beta} + g^T H^{-1} g$$

This σ_t is our error bar, and evaluating it mostly involves evaluating the Hessian matrix H at a given point. The first term is a standard noise term from our model of the noise on our data. Either term can dominate depending on the Hessian matrix. Computing this Hessian matrix effectively and efficiently is a non-trivial task, and this is why I was not able to apply these methods to get error bars on my supervised neural networks predictions. But given a more sophisticated setup, I would be able to compute this Hessian and thus be able to sample the model posterior in order to get a sense of whether my network has found a predictive set of parameters or not.

6 Implementation and Results

6.1 Implementation

The model was implemented using the standard Python SciPy stack. For the neural network model, the Keras library with a Theano backend was used. This let me train my models on my GPU, which gave higher performance training for faster iteration on trying different models.

6.2 Results

The supervised version of the nerual network showed the most promise for being the basis of a trading strategy. The network was trained on the 30 symbols in the Dow Jones Industrial average over a period of two years, representing approximately 3 million data points. This was done in two ways: first, a network was trained on each symbol. These networks try to learn features unique to each symbol. The predictions from these networks were then used as a trading strategy. The trading strategy is to buy if the predicted future movement is up more than one standard deviation, and to sell if the prediction is down more than one standard deviation. The results from these models are shown in table 1. Without including fees or execution difficulties in buying and selling at the minute time scale, this strategy appears to be profitable for 24 of 30 symbols in the Dow Jones Industrial Average, trained on 2013 and tested on 2014.

Another network was constructed that was trained on all 1.2 million data points to see if the large amount of data would reveal any interesting overall stock market patterns. The features learned are shown in 6.

These features seem to shown that a significant predictor for future returns is some fifteen overall upward or downward trend within the last fifteen minutes. Each of these price features is also paired with volume features, which are much less easy to visualize, but there may be more differentiation between these features within those, i.e. if the price rises on high volumes or if it rises on low volumes and what that says about predictive power. In any case, the large trained network was able to discover useful patterns in price and volume for predicting future returns.

Table 1: Output of neural network trading strategy on Dow Jones Industrial Average.

Symbol	Profit (annualized return)	Sharpe Ratio (annualized return/$\hat{\sigma}$)
AAPL	0.16	0.37
AXP	1.30	3.13
BA	1.16	1.88
CAT	5.73	10.10
CSCO	-0.97	-3.11
CVX	0.58	2.07
DD	0.43	1.19
DIS	-0.16	-0.42
GE	1.49	4.77
GS	-2.26	-6.06
HD	2.01	5.94
IBM	0.68	2.05
INTC	-3.07	-8.21
JNJ	0.67	2.03
JPM	0.35	0.91
KO	0.34	1.06
MCD	-0.88	-2.65
MMM	3.63	10.99
MRK	3.22	7.35
MSFT	0.65	1.55
NKE	2.08	6.04
PFE	2.92	7.19
PG	1.57	4.61
TRV	0.36	1.39
UNH	2.86	6.65
UTX	1.99	4.59
V	4.83	9.29
VZ	-1.32	-3.22
WMT	2.21	7.10
XOM	1.16	3.73

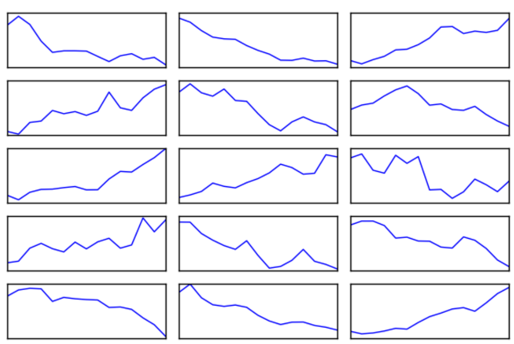


Figure 6: Features learned by training on millions of data points across symbols.

7 Conclusions

In conclusion, on the level of minute data there do seem to be exploitable patterns in stock price data. While linear methods struggle, non-linear neural networks may find subtle structure. I learned quite a few useful tips and tricks for applying these methods doing this project One is that training these models on GPUs saves lots of computational time; a real deep learning setup would be able to crunch even more data faster, and my analysis seems to show that the more data fed to these models the better they seem to learn structure. Another trick I learned was to always unit normalize inputs and outputs to these networks. This helps keep parameters in sensible ranges and improves network performance since they are trained on 32 bit floating points.

Future research would be to investigate the statistical significance of these methods. Although it appears the networks learned interesting structure, I would not actually apply them to trading until I could verify that the structure was not just noise. I would use rolling training/validation/test sets, more than the fixed sets I used here, to verify that the models had learned something interesting. Also, these models could be made more realistic by incorporating many other sources of data relevant to stock price movemetns. But overall neural networks due seem to be a useful tool for discovering stock market features.

References

[1] Christopher M Bishop. *Neural networks for pattern recognition*. Oxford university press, 1995.

[2] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[3] Martin Längkvist, Lars Karlsson, and Amy Loutfi. A review of unsupervised feature learning and deep learning for time-series modeling. *Pattern Recognition Letters*, 42, 2014.