Q-Learning Stock Trader

Machine Learning Engineer Nanodegree
Capstone Project

David Shahrestani
October 24, 2016

Definition

Project Overview

As of June 2016, the hedge fund industry had more than $3 trillion in assets under management[1]. Of those assets, an increasing portion has been allocated based on quantitative models over the years. Recently, advancements in deep learning – specifically deep neural networks – have opened the door to fully autonomous trading programs. Consider the following excerpt from a Wired Magazine article:

> In recent years, funds have moved toward true machine learning, where artificially intelligent systems can analyze large amounts of data at speed and improve themselves through such analysis. The New York company Rebellion Research, founded by the grandson of baseball Hall of Famer Hank Greenberg, among others, relies upon a form of machine learning called Bayesian networks, using a handful of machines to predict market trends and pinpoint particular trades. Meanwhile, outfits such as Aidyia and Sentient are leaning on AI that runs across hundreds or even thousands of machines. This includes techniques such as evolutionary computation, which is inspired by genetics, and deep learning, a technology now used to recognize images, identify spoken words, and perform other tasks inside Internet companies like Google and Microsoft. (*The Rise of the Artificially Intelligent Hedge Fund*)

Also consider the recent news of Google's AlphaGo beating Lee Sedol – the reigning world champion – in a five-game match of Go[2]. AlphaGo is a computer program that uses reinforcement learning and a deep neural network to improve its play. The version of AlphaGo that beat Lee Sedol used 1,920 CPUs and 280 GPUs.

All of this has inspired me to build a stock trading agent that allocates assets through reinforcement learning and deep neural networks. Simply put, my stock trading agent will be given a set of data (sourced from Yahoo Finance) that it will make actions on, then be rewarded or punished for those actions, and then update its memory based on the state it was in, the action it took, and the reward it received. It will learn a policy that will maximize its rewards at every state.

Since I don't have access to the raw computing power of AlphaGo or the expensive data feeds of hedge funds, my stock trading agent will be dumbed-down appropriately. As such, I will only be looking at nine possible investments (sector ETFs), features that can be extracted from freely available price data (technical data), and a small time window of market data (200 days). Consider it more a framework, or proof of concept, for a trading agent that can be scaled up with better hardware and data feeds.

[1] http://www.barclayhedge.com/research/indices/ghs/mum/HF_Money_Under_Management.html
[2] https://en.wikipedia.org/wiki/AlphaGo_versus_Lee_Sedol

Problem Statement

The goal of this project is to create a stock trading agent that uses reinforcement learning to successfully implement a sector rotation strategy[3]. The trading agent will be trained on 175 days of stock market data and use it to predict which stock market sector will outperform in the following 25 days. The proxy that will be used to represent the different stock market sectors are the SPDR® Sector ETFs[4]:

- XLY: Consumer Discretionary
- XLP : Consumer Staples
- XLE: Energy
- XLF: Financials
- XLI: Industrials
- XLB: Materials
- XLK: Technology
- XLU: Utilities

For example, if "XLU" preformed better than all the other sectors for a given test period, we would expect the trading agent to allocate its funds to "XLU" for that period. As such, the intended solution for any time period can be found by just checking which sector actually outperformed during the given time period.

Accomplishing this will take a strategy of 7 steps:

1. Retrieve and process data (price data, technical indicator data).
2. Create a function to manage data loading for different training and testing periods.
3. Create a portfolio class to keep track of the agent's performance.
4. Create an evaluation function to test the agent on.
5. Design the Q-learning functions (state, action, reward, new state).
6. Build the Q-function out of a neural network model.
7. Train and test agent over multiple epochs in a Q-learning loop.

Metrics

The agent will be evaluated based on three metrics, the first two being in relation to a benchmark which in this case is the S&P500 stock market index represented by the SPY ETF:

- Cumulative Return: The aggregate amount an investment has gained or lost over time, presented as a percentage. This shows us how well our stock trading agent did over the whole time period involved. This helps us avoid issues of doing well in one section and then bombing in another.
    - (Current Portfolio Value) - (Original Portfolio Value) / (Original Portfolio Value)
- Sharpe Ratio: A measure for calculating our risk-adjusted returns. This has become an industry standard for measuring the performance of hedge fund portfolios. Think of it as the average return earned in excess of a risk-free rate divided by the amount of risk taken

[3] http://www.investopedia.com/terms/s/sectorrotation.asp
[4] https://www.spdrs.com/product/index.seam

on. Any stock trading agent can beat the S&P500 by taking on more risk, but that isn't always the optimal solution.
    - (Portfolio Return - Risk free Return) / Portfolio Standard Deviation
- Whether action choices match with what we would expect for time period.

It is important to note that a live stock trading agent would need to account for trading fees and volume limitations. For simplicity, these factors are ignored. It is also worth noting that the standard machine learning metrics, like accuracy, are not great for this problem. For example, the agent might pick the right move 80% of the time, but the 20% of the time that it is wrong might wipe out the portfolio. Or the agent might make the right moves but with too much risk taken on. For these reasons, we use cumulative return and Sharpe ratio instead.

Analysis

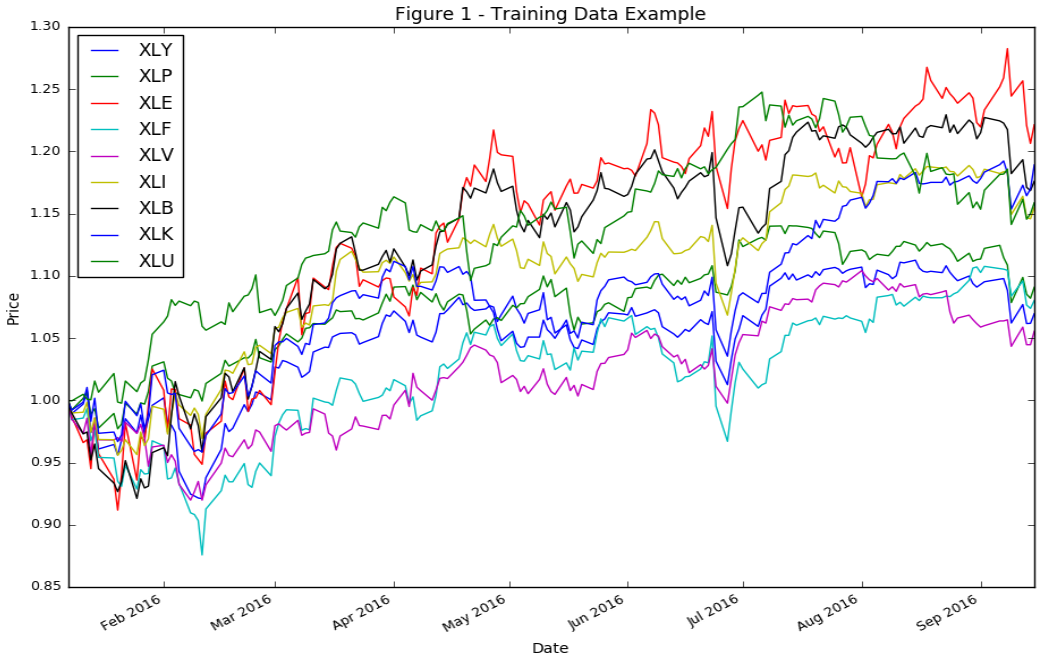
Data Exploration

The data that is extracted from yahoo consists of the daily price and volume information for each requested ETF. For example, the raw data for XLY looks like:

Date	Open	High	Low	Close	Volume	Adj Close
10-24-16	79.99	80.21	79.82	80.13	3537075	80.13
10-21-16	78.58	79.71	78.58	79.59	4295700	79.59
10-20-16	78.93	79.15	78.64	78.96	4909500	78.96

We will only require the “Date” and “Adj Close” columns for this project. “Date” is the day that the market information was recorded. “Adj close” is the closing price on a given day adjusted to reflect past dividends and stock splits. This is important because if the data was not adjusted our agent would view something as harmless as a stock split as a crash in the market price. With the adjusted closing data we can easily calculate technical indicators in preprocessing.

Data has been sourced for the time period February 1, 1999 to present day. The start day was selected because it coincides with the creation of the sector ETFs that this project will be using. It is important to note that some days will be missing data because the stock market was closed on that day. This will be dealt with in preprocessing, as will the isolation of relevant information. The following graphic visualizes the processed price data that will be used for training our trading agent. You can see how different sectors performed relative to each other (Figure 1):



In this specific time range, XLE outperforms the other 8 sector ETFs. If our agent was trained and tested on this time period, we would expect it to learn to allocate its funds to XLE at some point in the range.

Algorithms and Techniques

Q-learning is a model-free reinforcement learning technique that works by learning an action-value function (Q-function) that will give us the expected utility of taking any given action in a given state[5]. Q-learning requires the understanding of a few basic concepts:

1. State (S) – a representation of the current environment. In our model, the state will contain technical indicator data for all 9 sector ETFs.
2. New State (S') – a state one time step later after performing an action.
3. Action (A) – a selected move that can be taken at the current state. In our model, the actions will be choosing one of the 9 sector ETFs to overweight for a time step.
4. Reward (R) – the payoff for choosing an action at a given state. In our model, the reward will be the portfolio's percent increase from the initial state to the new state. A loss will be weighted higher than a gain.
5. Q-function (Q(s,a)) – a function that estimates rewards for given actions at given states. In our model, the Q-function will be a neural network.

Before getting into why this project will use a neural network, let's discuss some of the parameters involved with Q-learning that can be tuned for optimization:

- Gamma – the discount factor determines how important future rewards will be. A factor of 0 means that the agent will only consider immediate rewards, while a factor approaching 1 means the agent will strive for a long-term reward.
- Epsilon – the exploration factor determines how frequent we choose to explore vs exploit. A factor of 0 means the agent will always use our Q-value and won't learn, while a factor of 1 means the agent will always choose randomly and won't use what it has learned.
- Reward – the reward that we use can have a huge impact on what our Q-learner ends up learning.

Now for the Neural Network model that will act as the Q-function of our Q-learner. First, let's explain why there is a need for a Neural Network. A classic Q-learning algorithm will discretize the state space into S states and map a Q-value for every state-action pair. This quickly becomes computationally impossible when the state space is complex. For example, our state space will contain millions of different combinations of technical data for 9 different ETF's. This is why a neural network is needed; it approximates those state-action pairs for us given the complex state space.

I have chosen a Long Short-Term Memory (LSTM) recurrent neural network. A LSTM network is a network that contains blocks that can remember a value for an arbitrary length of time. Its applications range from speech recognition to robot control, but its success in time series prediction is what makes it a viable network for our Q-function. Parameters involved with LSTM neural networks that can be tuned for optimization are:

[5] http://www.gatsby.ucl.ac.uk/~dayan/papers/cjch.pdf

- Training parameters
    - Epochs – length of training loops.
    - Batch Size – how many days to look at for a training step.
    - Memory Size – how many days to store for training.
- Network architecture
    - Number of layers
    - Layer types
- Layer parameters
    - Numerous
- Preprocessing parameters
    - Type of data to include
    - Length of data time window
    - Length of technical indicator time window

A technique known as experience replay will be used for the learning loop that should improve learning speed. I got the inspiration for this technique from *Playing Atari with Deep Learning*. Basically, a minibatch is sampled from our Q-learning experience and our neural network is trained on that minibatch. The neural network will also use a concept called dropout which is a regularization technique that randomly selects neurons to ignore during training[6]. This means other neurons will have to step up to fill in and the model as a whole should become less sensitive to the specific weights of neurons. This helps deal with the overfitting problem in neural networks.

With all of this in mind we can discuss how the Q-learning technique will operate:

1. Initialize state and Neural Net Q-function.
2. For each epoch:
    a. Iterate through stock market data picking random actions for every state and recording the reward and new state.
    b. Add (state, action, reward, new state) to experience replay.
    c. Train Neural Net on minibatches of experience replay.
    d. Evaluate agent on testing data.
    e. Reduce randomness in favor of our neural network Q-function.

The idea is that with every loop, the Neural Network Q-Function will learn more about the environment and be able to approximate the rewards that come with every possible state-action pair. This can be used to optimize our policy at every given state.

Benchmark

There are two key benchmarks that will be used to evaluate our stock picking agent. The first is a portfolio that is 100% allocated to the SPY ETF. The SPY ETF tracks the S&P500 stock market index which can be thought of as the average of all 9 sector ETFs. If our stock picking agent can't beat the SPY, it clearly is not able to correctly identify sectors that outperform.

[6] http://machinelearningmastery.com/dropout-regularization-deep-learning-models-keras/

If the stock picking agent does beat the SPY, it doesn't mean it is learning the optimal sector allocations. To make sure, I will compare the actions taken by the stock picking agent to the actions that would have resulted in the best performance over the period.

Methodology

Data Preprocessing

Originally, I wanted to have two main features for the Q-learner's state, technical data and fundamental data, but finding an accurate feed of historical fundamental data proved costly. The cheapest options from Quandl are around $1 thousand per year[7]. Instead, I went solely with technical data that could easily be calculated from free data sources.

I used many of the introductory techniques taught in Udacity's Machine Learning for Trading course for this section. The main chunk of code can be found in qlearning_trader.ipynb under Data Preprocessing.

The main cleaning that had to be done was removing days when the market was closed. Since SPY is one of the most heavily traded vehicles in the market, it is safe to say that if it wasn't trading then the market was closed. After dropping these dates, the adjusted closing data for the 9 sector ETFs could be added to a single dataframe. The function returns this price dataframe and an SPY price dataframe to be used as a benchmark for evaluation.

Now that we have the price data, we can start to calculate the technical data that will be used as the State for our Q-learner. There are many different technical indicators that could be used for this project, but I chose to go with the percentage difference between the price of a sector ETF and its simple moving average and whether or not the price of a sector ETF was above or below its Bollinger Bands ®. The main chunk of code can be found in qlearning_trader.ipynb under Data Preprocessing.

As you can see in the code, each technical indicator requires a window of time to be calculated on. After experimentation, I settled on the 25 and 100 day simple moving average and the 50 and 200 day Bollinger Bands ®. After combining the technical data for every sector ETF into one dataframe, our final technical data contains 4 technical indicators for each sector ETF across the entire time period.

Implementation

There were 5 main steps for implementation of the stock trading agent. For each step, if sufficiently complicated, I will show the initial and final code (after refinement) and briefly talk about the refinement process and any complications that occurred. In the next section, I'll go into more detail on the process of refining the trading agent. (For full code see qlearning_trader.ipynb)

Step 1 – Create a function to manage data loading for different training and testing periods.

Initially, my plan was to train the trading agent on all 4,000 days of stock market data, but I abandoned this strategy when the training time took multiple days. Instead, I modified the data to load 200 days of data (175 training, 25 testing). The idea was to initialize the data at random periods in order to perform a kind of random rolling cross-validation.

[7] https://www.quandl.com/data/SF1/pricing

Step 2 – Create a portfolio object to keep track of the trading agent's performance.

Most of this code came from Udacity's Machine Learning for Trading course. The portfolio object is initialized with an empty port_tracker dataframe. This is added to as our trading agent executes actions. This class also contains functions for calculating performance of the portfolio and plotting it.

Step 3 – Create the Q-learning functions for state, action, reward, and new_state.

The take_action code simply decides which stock market sector to overweight. Note that the portfolio never puts 100% of its assets in any one sector. Consider this a safety measure if the agent ever goes haywire.

Initial get_reward code:

```
value_new = port.get_value(price_data, new_state[1], time_step)
value_init = port.get_value(price_data, initial_state[1], time_step-1)
reward = (value_new / value_init) - 1
```

Final get_reward code:

```
if time_step < 20:
    reward = (port.port_tracker['value'][time_step-1] /
              port.port_tracker['value'][0])-1
    # Modify reward to weight gains and losses differently
    if reward > 0:
        reward = (reward + 1)
    else:
        reward = (reward - 1)**3
else:
    reward = (port.port_tracker['value'][time_step-1] /
              port.port_tracker['value'][time_step-20])-1
    # Modify reward to weight gains and losses differently
    if reward > 0:
        reward = (reward + 1)
    else:
        reward = (reward - 1)**3
```

The get_reward code evolved as I tested the learner. I started with a simple percent return from the previous day to next. I ended with a 20 day cumulative reward that punishes losses more than it rewards gains. Also, a large loss is more punishing than a small loss.

Step 4 – Build the Q-function out of a neural network.

Initial neural network:

```
model = Sequential()
model.add(LSTM(64,
               input_shape=(1,9)))
model.add(Dense(10, init='lecun_uniform'))
model.add(Activation('linear'))

adam = Adam()
model.compile(loss='mse', optimizer=adam)
```

Final neural network:

```
# Initialize sequential model
model = Sequential()

# Build first layer
model.add(LSTM(64,
               input_shape=(steps, num_features),
               return_sequences=True,
               stateful=False))
model.add(Dropout(0.50)) # drop inputs to avoid overfitting

# Build second layer
model.add(LSTM(64,
               input_shape=(steps, num_features),
               return_sequences=True,
               stateful=False))
model.add(Dropout(0.50)) # drop inputs to avoid overfitting

# Build third layer
model.add(LSTM(64,
               input_shape=(steps, num_features),
               return_sequences=False,
               stateful=False))
model.add(Dropout(0.50)) # drop inputs to avoid overfitting

# Build fourth layer
model.add(Dense(10, init='lecun_uniform'))
model.add(Activation('linear'))

# Build optimizer and compile
adam = Adam()
model.compile(loss='mse', optimizer=adam)
```

This code is a bit complicated. It requires that both Theano and Keras are installed on your workstation. A link to instructions for installing Theano can be found here and to install Keras here. I started out with a two layer neural net and ended with a 4 layer neural net. I also added multiple dropout layers to reduce overfitting. Dropout works by randomly dropping inputs at the given layer. I chose the Adam optimizer because it is well suited for problems that are large in terms of data, parameters, and noise[8].

Step 5 – Train and test in a Q-learning loop.

This code is too long to reproduce in its entirety but I will walk you through it. First, we initialize our training parameters (epochs, gamma, epsilon, batch_size, mem_limit). Next, we iterate through our epochs. For each epoch, we initialize our state and choose an action either based on our Q-function or randomly based on the level of epsilon. The results of this state-action pair are added to our memory. Next, a minibatch from our memory is used to train our neural network. The neural network takes technical data for the 9 sector ETFs as input and outputs the predicted reward for each action. Finally, we test our trading agent on the test set and repeat for the next epoch.

[8] https://arxiv.org/abs/1412.6980v8

Refinement

My refinement process can be broken down into three sections: reward refinement, neural network refinement, and training parameter refinement. Initially, this process ran into a speed issue as every change in parameters would take hours to train and test. To limit the time taken, I refined my trading agent on only the last 200 days of trading data in our dataset.

Before any refinement could take place, I had to design an evaluation framework for every iteration of the trading agent. To accomplish this, I built a function that would test the trading agent on a test set at the end of every epoch. This allowed me to see how the trading agent is progressing over the course of training and to compare its results across iterations.

The code for the evaluation function can be found in qlearning_trader.ipynb under Implementation.

This function uses our neural network Q-function to decide on actions at each step in the testing set. At the end, it returns the cumulative return of our trading agent and its Sharpe ratio. For example, in our initial iteration of the stock trading agent (code seen above in the implementation section), it consistently underperformed the SPY Benchmark in cumulative return.

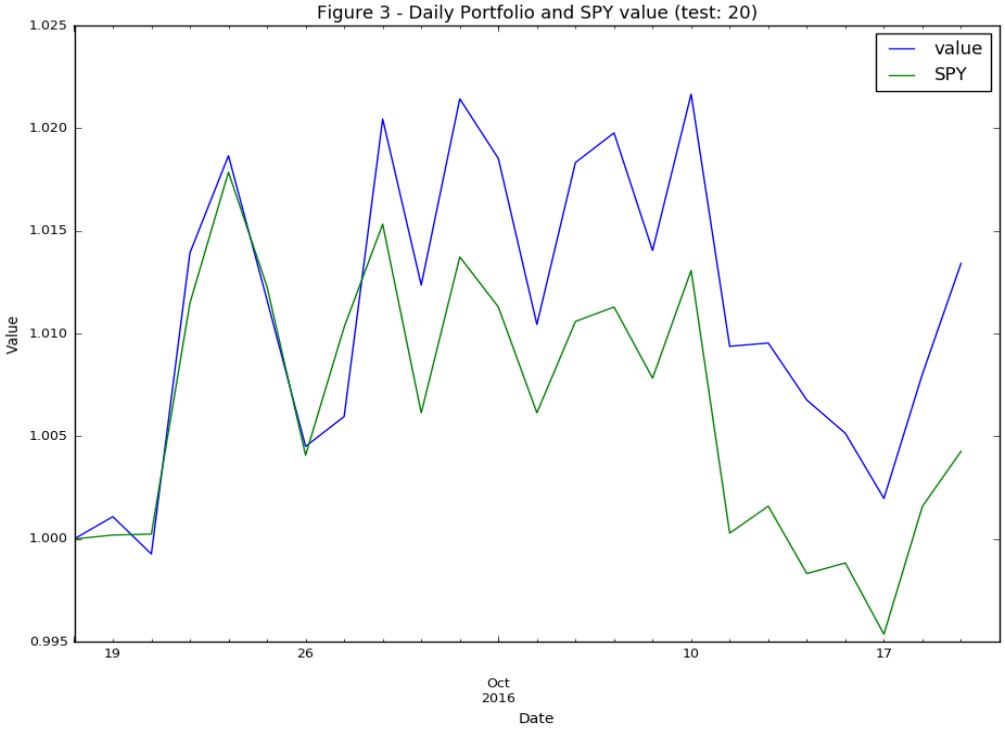
To adjust this, I first dealt with the reward parameters. I wanted my stock trading agent to avoid large losses so I modified to reward to disproportionately punish large losses. This was done by cubing the loss.

Second and where the major improvements came, I made my neural network more complicated. I added more layers and reduced the issue of overfitting by adding dropout to each layer. I experimented with many different layers and parameters, and settled on a four layer neural network.

Third, I modified the training parameters. I tried different levels of gamma, epsilon, and epochs. The main problem I had here was the running time of the trainer. I had to settle on lower levels of epochs even though it may not be optimal. Ideally with better hardware, a true rolling cross validation could be completed on the whole data set, giving us better insight into the best training parameters.

Consider this example of parameters that did poorly (Figure 2):

And this example of parameters that did well (Figure 3):



Results

Model Evaluation and Validation

The final model was judged on three criteria:

1. Did it beat the SPY Benchmark?
2. Did it pick the best actions?
3. Is it robust enough to generalize to other time periods?

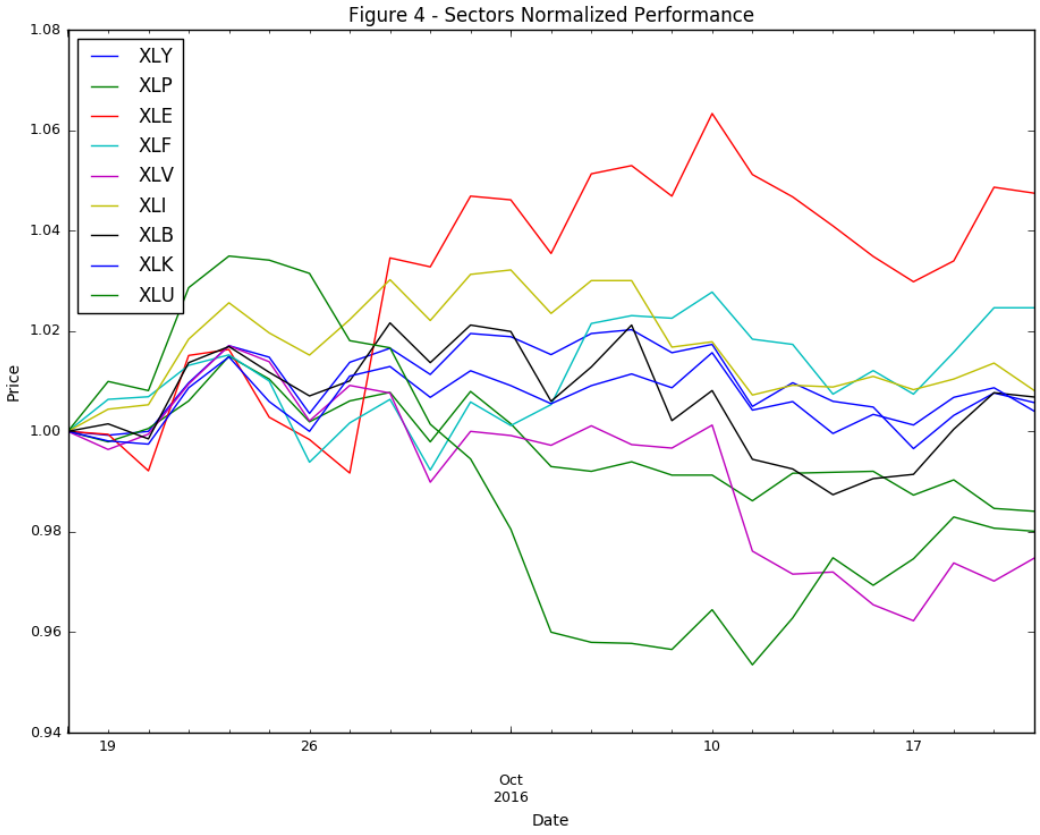
Final model result log:

Test 1/25: Relative Return/Sharpe: -0.3751%/-0.21
Test 2/25: Relative Return/Sharpe: -0.3751%/-0.21
Test 3/25: Relative Return/Sharpe: -0.3751%/-0.21
Test 4/25: Relative Return/Sharpe: 0.1376%/0.32
Test 5/25: Relative Return/Sharpe: 0.1376%/0.32
…
Test 20/25: Relative Return/Sharpe: 0.9153%/1.02
Test 21/25: Relative Return/Sharpe: 0.9153%/1.02
Test 22/25: Relative Return/Sharpe: 0.9153%/1.02
Test 23/25: Relative Return/Sharpe: 0.9153%/1.02
Test 24/25: Relative Return/Sharpe: 0.9153%/1.02
Test 25/25: Relative Return/Sharpe: 0.9153%/1.02

For the first time period tested (2016-09-19 – 2016-10-19), the trading agent beats the SPY benchmark's cumulative return by 0.9153% and its Sharpe ratio by 1.02. This satisfies criteria #1. Let's look at the action choices and compare them to the optimal actions (Figure 4) for the period:

2016-09-19 00:00:00 ACTION: 3 (1.001)
2016-09-20 00:00:00 ACTION: 3 (-1.002)
2016-09-21 00:00:00 ACTION: 3 (1.014)
2016-09-22 00:00:00 ACTION: 3 (1.019)
…
2016-10-03 00:00:00 ACTION: 3 (1.019)
2016-10-04 00:00:00 ACTION: 3 (1.010)
2016-10-05 00:00:00 ACTION: 3 (1.018)
2016-10-06 00:00:00 ACTION: 3 (1.020)
2016-10-07 00:00:00 ACTION: 3 (1.014)
2016-10-10 00:00:00 ACTION: 3 (1.022)
2016-10-11 00:00:00 ACTION: 3 (1.009)
2016-10-12 00:00:00 ACTION: 3 (1.010)
2016-10-13 00:00:00 ACTION: 3 (1.007)
2016-10-14 00:00:00 ACTION: 3 (1.004)
2016-10-17 00:00:00 ACTION: 3 (1.003)
2016-10-18 00:00:00 ACTION: 3 (-1.018)
2016-10-19 00:00:00 ACTION: 3 (-1.016)

Optimal actions:



Our model selects action 3 (XLE) which matches the best performing sector of the period.

Criteria #1 and #2 passed, but these results could be specific to the time period. Cross-validation can be problematic with time-series data since a pattern can emerge in one year and disappear in the next. To really know how well our model generalizes we will need to test it on other training and testing windows. To accomplish this, I initialized the Q-learner on a random start date between now and the late 1990's. I then evaluated how well the model did on this random time period in the same way as above.

The first random testing time period was 2004-03-11 (results found in logs under '2004-03-11'). The agent was trained on 175 days before that date and tested on 25 days after that date. The agent successfully passed criteria #1 and #2.

The second random testing time period was 2006-04-07 (results found in logs under '2004-03-11'). Once again the agent successfully passed criteria #1 and #2.

The third random testing time period was 2015-01-14 (results found in logs under '2015-01-14'). This time the trading agent failed to pass either criterion. Comparing the training data (Figure 5) to the testing data (Figure 6) shows us where the problem is occurring:

Figure 5 - Training Sector Normalized Performance

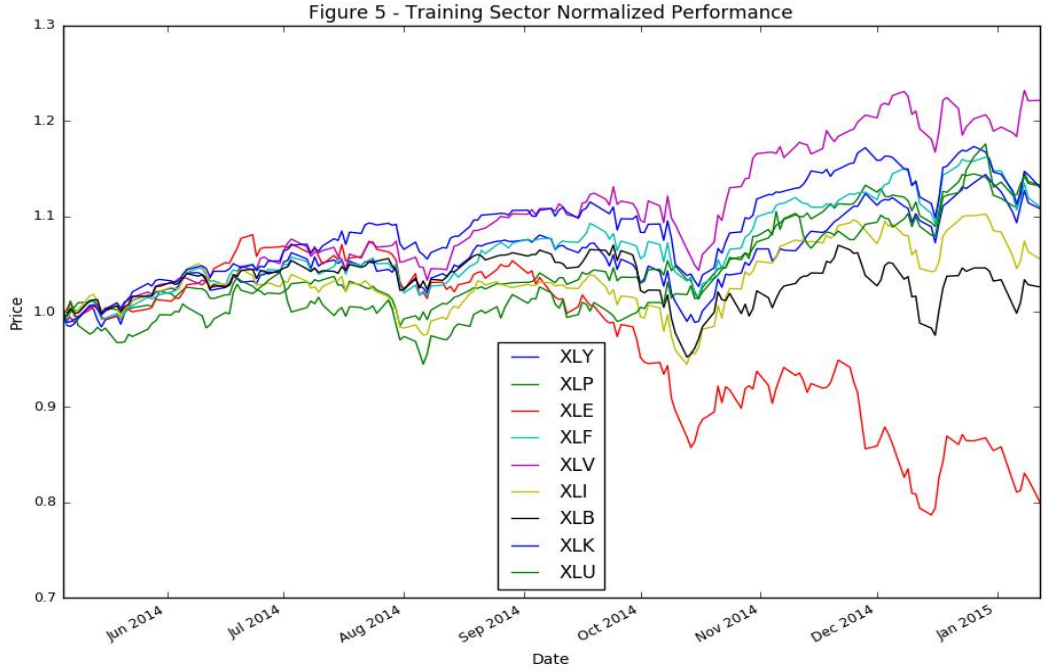
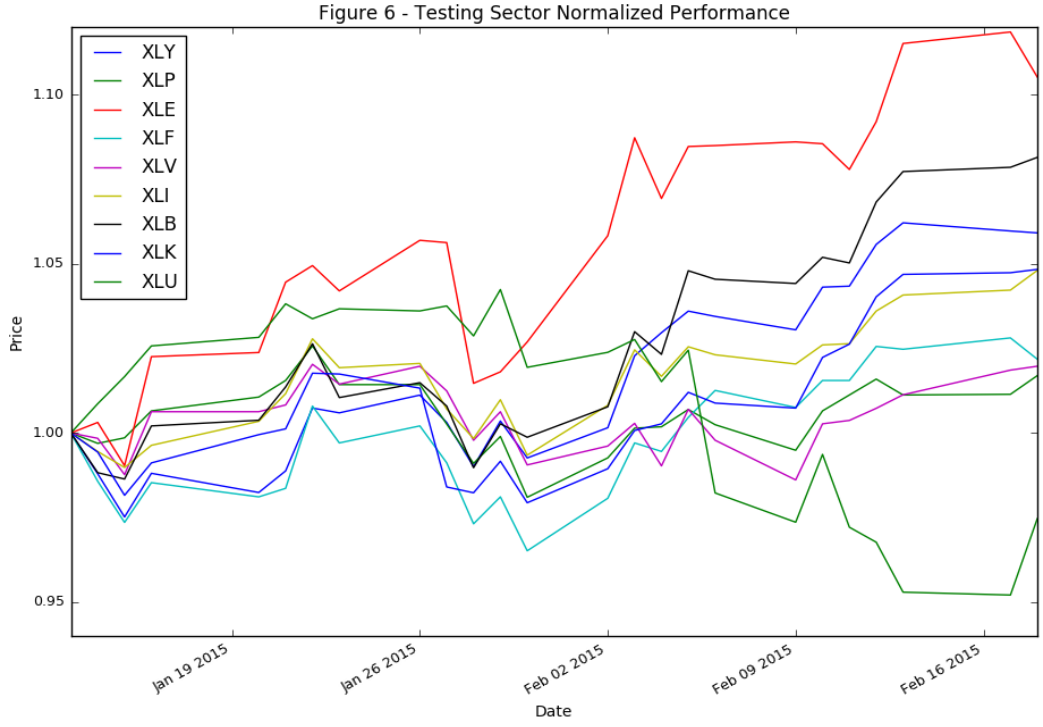


Figure 6 - Testing Sector Normalized Performance



The trading agent was trained on a time period that included a huge crash in oil prices[9]. The energy ETF XLE suffered as a result. The trading agent was then tested on a time period where XLE performed best. Let's look at the actions it chose:

2015-01-14 00:00:00 ACTION: 8 (-1.015)
2015-01-15 00:00:00 ACTION: 8 (-1.038)
2015-01-16 00:00:00 ACTION: 7 (1.003)
2015-01-20 00:00:00 ACTION: 8 (1.003)
2015-01-21 00:00:00 ACTION: 8 (1.010)
2015-01-22 00:00:00 ACTION: 5 (1.023)
…
2015-01-29 00:00:00 ACTION: 7 (1.005)
2015-01-30 00:00:00 ACTION: 8 (-1.026)
2015-02-02 00:00:00 ACTION: 7 (1.007)
2015-02-03 00:00:00 ACTION: 5 (1.018)
2015-02-04 00:00:00 ACTION: 5 (1.011)
2015-02-05 00:00:00 ACTION: 5 (1.024)
2015-02-06 00:00:00 ACTION: 5 (1.017)
2015-02-09 00:00:00 ACTION: 5 (1.011)
2015-02-10 00:00:00 ACTION: 5 (1.023)
2015-02-11 00:00:00 ACTION: 5 (1.026)
2015-02-12 00:00:00 ACTION: 5 (1.042)
2015-02-13 00:00:00 ACTION: 5 (1.030)
2015-02-17 00:00:00 ACTION: 5 (1.032)

The trading agent seemed confused on what the best course of action to take in the testing period was. It definitely failed to select XLE.

Justification

As we saw, the trading agent was able to beat the bench mark and choose the best actions for certain periods but failed at others.

	Relative Return	Relative Sharpe
2016-09-19	0.9153%	1.02
2004-03-11	0.2840%	0.08
2006-04-07	1.3841%	2.53
2015-01-14	-0.4109%	-0.35

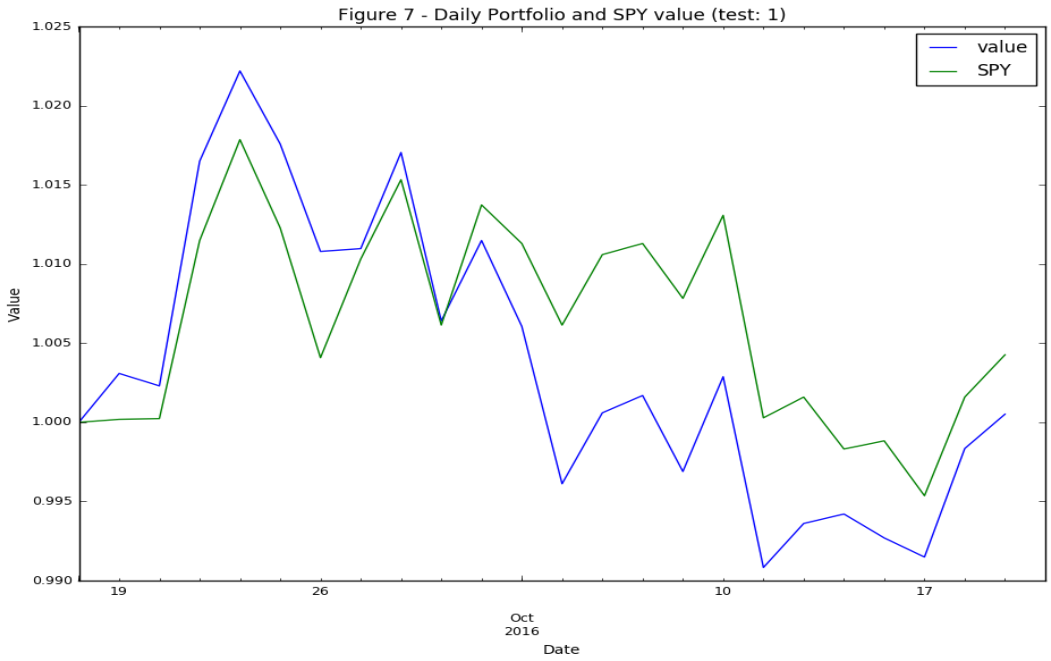
The trading agent is not robust enough to generalize to all time periods and solve the original problem statement of using 175 days of training data to predict the best performing stock sector of the next 25 days. However, the original purpose of the project was to provide a framework, or proof of concept, for a stock trading agent that uses reinforcement learning paired with neural networks. In this regard, the stock trading agent was successful. I could see this framework doing well when scaled up with fundamental data and faster hardware.

[9] http://www.investopedia.com/ask/answers/030315/why-did-oil-prices-drop-so-much-2014.asp

Conclusion

Free-Form Visualization

Let's take a look at the first training period, specifically at how our agent improved over the course of the training epochs. The first epoch's results are shown in Figure 7 and the final epoch's results are shown in Figure 8.



You can see that the agent didn't really know what it was doing for the first epoch of training. It even consistently chose actions that performed worse than the benchmark.

You can see that the agent performed better in the last epoch of training. These two plots that we logged over the course of training summarize the reinforcement learning process. They show that through reinforcement learning, the stock trading agent can improve its performance and find an optimal policy.

Reflection

In order to create a stock trading agent that can successfully implement a sector rotation strategy, I have employed the technique of Q reinforcement learning paired with a neural network that acts as the Q-function.

I trained the agent by feeding it 175 days of stock market technical data and then letting it choose actions (1-9) which decided which stock market sector it would overweight. These actions were either chosen randomly (depending on epsilon) or chosen based on the maximum reward predicted by the neural network Q-function.

In order to train the neural network, a minibatch of experience (state, action, reward, new state) was repeatedly fed into it as training data. This process was repeated for the length of the epochs.

At the end of each training epoch, the trading agent was evaluated based on a test set of 25 days immediately proceeding the testing set.

The main issue that this process ran into was the time of training. To train the model over 100 epochs on all 4000 days of market data would have taken multiple days. Even when the data is scaled down, it still can take hours to run an iteration of the trading agent. This caused problems for parameter tuning, as only so many combinations could be attempted in a given day.

The final results were promising for how scaled down our stock trading agent was (in the next section I will talk about scaling it up). However, our stock trading agent failed to generalize to every time period in our dataset. On top of that, the trading agent does not take into account transaction fees or volume limitations in the stock market. This trading agent should not be used in a general market setting to implement a sector rotation strategy.

Improvement

At the outset of this project, I wanted to create a framework, or proof of concept, for a fully autonomous stock trading agent that could learn on its own. I was inspired by the work Google did with AlphaGO and the fun I had in the last project of training a smartcab to drive.

The stock trading agent that was created through this process performed well for how dumbed-down it was. To improve on these results I offer 5 suggestions.

1. Include transaction costs in the reward and volume limitations in the actions.
2. Include fundamental data from a paid subscription in the state.
3. Increase the layers of the neural network as hardware allows.
4. Increase the training time period as hardware allows.

5. Explore all parameters through a type of gridsearch rolling cross-validation as hardware allows.
6. Implement concept drift detection (ADWIN, CUSUM) which can modify the training window based on changes in the background of the learning problem (i.e. a crash in oil prices).

If these suggestions are executed on, I believe that this framework can be a powerful tool for building a fully autonomous stock trading agent.