CS230

Deep Q-Networks For Multi-Asset Trading Decisions

Shen Gao
Stanford University
shengao@stanford.edu

Abstract

Trading in major global markets have increasingly moved to a programmatic regime. Market micro-structure has changed in ways human traders, without the ability to react to market conditions in a timely manner, are conceding grounds to computers and trading algorithms. In this paper, we examine a deep reinforcement learning approach to trading a portfolio of stocks utilizing a deep q-network that makes trading decisions based on observed financial time series inputs. We then examine the decisions made by such algorithm evaluated by portfolio performance. Based on our naive model construct, we observed significant out-performance in the test set and discuss potential issues with such approach.

1 Introduction

Quantitative investing has become the new norm in today's financial markets. According to a recent study, over 70% of US equities trading by volume is electronic. It has become crucial for not only the savvy institutional investors to leverage complex trading system to compete in the sub-millisecond latency space but also retail investors who could invest effectively in this new dynamic.

The premise of our approach is that trading can be formulated as a complex game. With the groundbreaking development in deep reinforcement learning in recent years[1], deep q-network (DQN) has rise to prominence with many applications involving decision making processes.

In this paper, we formulate trading as a game of maximizing financial gains. A neural-network is used as an agent that interacts with an environment of stock prices and makes optimal decision on which and how many stocks to trade. As to the q-network, we use a multi-layer perceptron (MLP) network which takes stock prices over a set look-back window and return probabilities of a predetermined action space that allows the agent to buy, sell or hold a given stock. Each action would then associate with a reward (e.g. buy 1000 shares of Tesla stock would result in a gain of 1000 times price change, adjusted for risk aversion and trading cost). In training, the model steps through the training time steps. At each time step, the agent act upon the input state and returns an optimal action. Based on the action, the reward is determined and a next state is also returned. The agent then remembers all the state and actions leading up to the current time step and replays the experience. In the replay, a randomly sampled batch of experiences is selected. The model then iterate through the batch and use the Bellman equation for updating the predicted model output with ground truth reward and discounted future optimal outputs. The model is then trained on each batch with X being the replayed current state and Y being the reward-updated model outputs.

2 Related work

There has been numerous attempts on using a q-learning framework for trading applications. Ritter[2] and Gao[3] used DQN framework on simulated asset price data and showed the framework's ability

CS230: Deep Learning, Fall 2019, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

to spot price momentum on single stocks and discover correlation among a leading signal and actual asset price. While results are encouraging, both only acts on single asset. Gao, Chien[4] and Wang et el.[5] used a rather limited action space. Chien claims the first ever application on real financial data. Chien and Kim et el's [6] works are both replete with practical considerations such as reducing reward by estimated transaction costs in the form of spreads, feeding multiple market conditions as states and applying DQN to well-known trading approaches (pairs trading).

[6]

3 Dataset and Features

Daily adjusted close stock prices from the alpha-vantage API is the sole data source of this project. The alpha vantage API provides financial time series data for major US and International stocks, cryptocurrencies and foreign exchange rates, on a interval of minutes, hourly, daily, weekly and monthly.

Eight stocks representing a diverse set of industry sectors are used for training and testing. They are: Tesla Inc (TSLA), Alphabet Inc Class A (GOOGL), Morgan Stanley (MS), IBM (IBM), Coca Cola (KO), Chevron Corporation (CVX), Home Depot Inc (HD), Mcdonald's Corp (MCD)

Price history starts from 2010/06/29. Training set is the first 1500 trading days ending on 2016/06/13. Testing set starts from the first day following training ending day until 2019/12/01.

For dates that one or more stocks don't trade, we removed the date completely. The environment takes the price history data raw. When states variables are generated as model inputs, a time-window is used to obtain last 20-day price history as state. Then the state is normalized with the time-window mean and standard deviation for each stock.

The reward function uses an exponentially weighted moving average (EWMA) with center of mass 9.5 (approximate to 20-day moving average) instead of the raw price time series. Using EWMA would greatly reduce noise in financial time series as some of the stocks used in this exercise are quite volatile (e.g. TSLA).

4 Methods

4.1 State Space

The trading problem is formulated under the Markov Decision Process (MDP) which says outcomes are partly random and partly controlled by a decision-making agent, i.e. the DQN agent in our study. MDP defines future state to be dependent only on current state and completely independent of any previous states.

The State Space is $\mathbb{R}^{20x8}$ with 20-previous period stock price for 8 stocks. The DQN will make decision solely on this short-term price history.

The state input is normalized by in-window mean and standard deviations before fed to DQN agent.

4.2 Action Space

The Action Space is $\mathbb{R}^{5x8}$ with each of 8 stocks able to taking actions {-1000, -100, 0, 100, 1000}, representing sell, hold, buy and the respective number of shares traded in each direction. The actions are intentionally left with uneven space to differentiate small appetite/micro-tuning versus large portfolio re-positions/re-balancing.

The action is flattened to $\mathbb{R}^{40}$ before fed to DQN for training due to efficiency.

4.3 Reward Function and Additional Variables

Additional variables used to evaluate reward are defined as follows:

- Min and max positions: minimum and maximum number of stocks the portfolio can hold for any stock. In an long-only case, the min would be set to 0 as no shorting is allowed.

- Positions: number of stocks held for each stock at each time step
- Transaction cost: $0.5 for each share traded applicable to both buy and sell
- Risk aversion factor: any negative rewards is multiplied by 1.2 to magnify the negative reward to reflect human's innate aversion to risks.
- Portfolio value: value of overall portfolio evaluated with daily price at each time step
- Cash positions: available cash on hand at each time step
- Initial wealth: $1 million dollars in cash
- Allow borrowed cash: Boolean indicating if can use borrowed cash to buy stocks.
- Reward function: for stock i at time t, the economic gain from a trade is defined as follows:

$$\begin{aligned}&(P_{t+1}N_{t+1} - P_tN_t) + (N_t - N_{t+1})(P_{t+1}) - c|N_t - N_{t+1}| \\ &= (P_{t+1} - P_t)N_t - c|N_t - N_{t+1}|\end{aligned}$$

  where
  P_t, P_{t+1}: stock prices at current state and next state
  N_t, N_{t+1}: stock positions (in shares) at current state and next state
  c: fixed transaction price per share traded (buy or sell)

- Additional notes:
  - The reward value is normalized by initial wealth.
  - For long-only cases and no borrowed cash allowed, if DQN's proposed trade cannot be satisfied due to limited cash or position limits, the proposed trades are first clipped based on position limits, then allocate cash available and proceeds from proposed sell trades to buys proportionally to buy costs.

4.4 DQN Layers

Due to speed constraints, use simple 3-Layer neural network:

- Reshape layer to flatten state space of (20, 8) to 160-dimensional input
- Dense layer with 512 neurons and ReLu activation
- Dense layer with 64 neurons and ReLu activation
- Output Dense layer with 40 output units and Hard Sigmoid activation. Using Hard Sigmoid instead of Sigmoid due to faster performance

4.5 Algorithm

See Algorithm 1 for "DQN Trader" psedocode describing how above modules interacts and how the model is trained.

One of the major issues with DQN is that it learns sequentially through time and tend to overfit to the timing of environment. To reduce this effect, we use an experience-replay mechanism to train on randomly sampled experiences rather than on sequential experiences. To further reduce overfitting, we employ an ϵ-greedy action exploration mechanism to add random actions during training. The ϵ starts at 0.7 and reduced by multiplying 0.995 after each time step

5 Experiments/Results/Discussion

Since reinforcement learning works differently from supervised learning, we use total portfolio return as model metric. Please see Figure 1 for portfolio return improvement over training period.

In previous iterations of this model, it has overfitted the environment significantly. To ameliorate this, I have added random exploration (discussed above) and selected a relatively high exploration rate of 0.7.

To put the model performance in perspective, Figure 3 illustrates wealth appreciation paths for the DQN strategy versus two simple strategies.

Algorithm 1 DQN Trader

Initialize environment E, recurrent Q-network, Q_θ, agent A with Q_θ as input
Initialize Simulator with inputs E and A for training and testing
for each episode **do**
  Observe initial state s from E
  while each time step in E **do**
    Select greedy action a w.r.t Q_θ and apply to E
    Receive reward r and next state s' from env E
    Store memory (s, a, r, s') to A's memory M
    Agent A replays memory M up until current time step:
    Randomly sample a batch of memories from M
    for each batch i **do**
      Get (s, a, r, s') from batch i
      Train Q_θ based on the Bellman Equation via TensorFlow
      $L(\theta) = E_{s,a,r,s'}[||r + \gamma Q_\theta(s', argmax_{a'} Q_\theta(s', a')) - Q_\theta(s, a)||^2]$
      where $Q_{\theta'}$ is the Q-network output given the next state s' and the next action a'
      $\theta = \theta - \alpha \nabla_\theta L(\theta)$
    end for
    Update s = s'
  end while
  if Exploration threshold ϵ greater than 0.1 **then**
    Reduce ϵ by multiplying a decay factor 0.995
  end if
end for

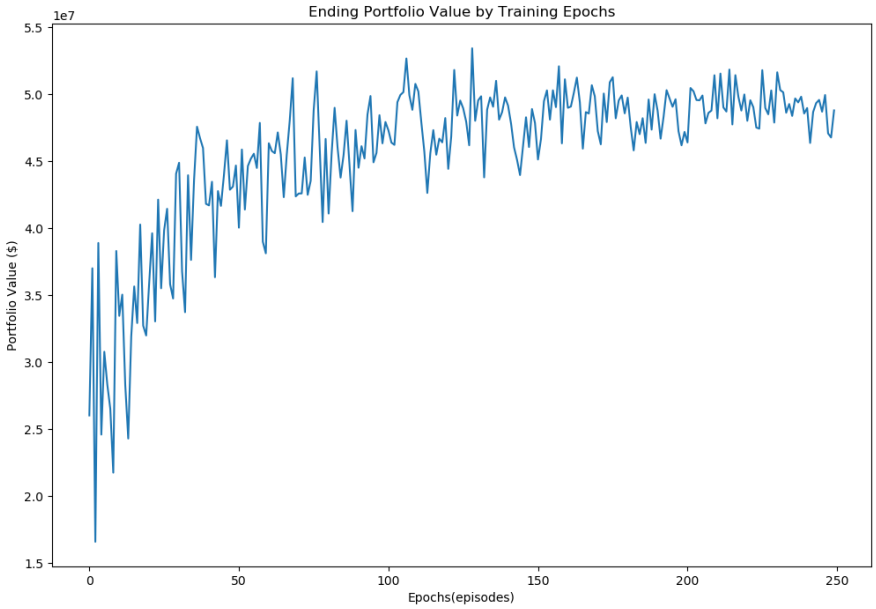

Figure 1: Portfolio value of strategies over training period

- Benchmark 1: An equally weighted portfolio at inception. Buy and hold through time.
- Benchmark 2: Pick the best performing stock among the eight and take full position in that stock and hold till the end

As illustrated the DQN strategy, though underperforms initially, eventually far outperforms the other two.

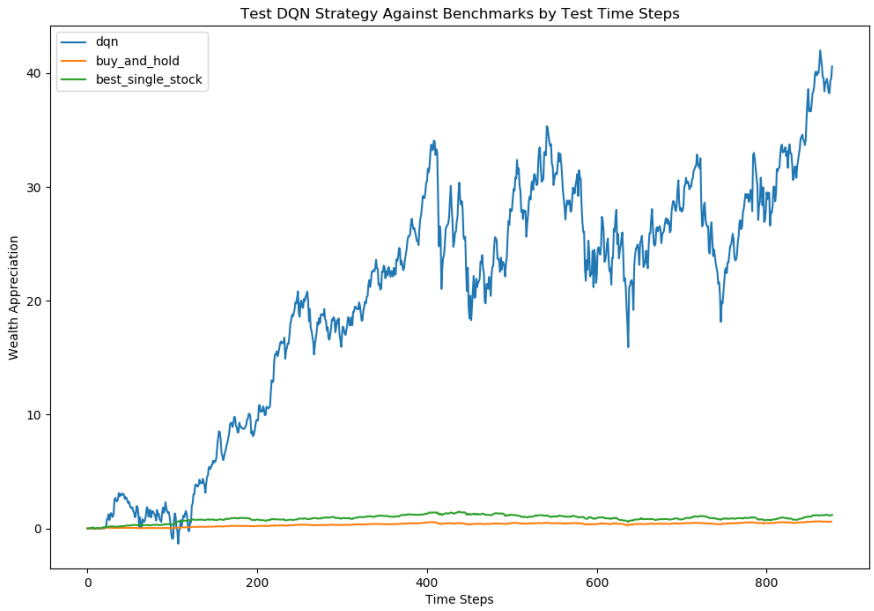

Figure 2: Epoch-ending Reward Kernel Distribution

6 Conclusion/Future Work

In this paper, we examine a DQN approach applied to trading. The model results convince us that it's capable of not only predicting best course of action but also able to be applied to asset allocation or portfolio rebalance.

However, to be used in real world trading environment, the model's action space will need to be augmented greatly to account for not only buy or sell 100 or 1000 stocks but a list of all possible trades in either direction. The action space would be (all practically feasible trade lot sizes, number of all potentially traded assets).

Another potential issue with this approach is that I have yet to test it with more stocks. It is entirely possible that the outstanding results might not work with certain types of assets. It's also possible that since both the train and test periods are in the longest bull market in the history it's much easier for the algorithm to produce positive returns than otherwise. It would be a worthwhile exercise to deploy this algorithm in a distressed period such as the 2008 Financial Crisis and see if the model would still outperform its peers.

On the more technical end, DQNs are very specific to the environment it's in. This makes the model to easily overfit to the environment. I would like to explore other ways to reduce the tendency to overfit in addition to random exploration, as well as improve the algo to be trained on many different environments (e.g. bear market conditions or many different sets of stocks). I would also like explore other related reinforcement learning approaches such as double-q learning, policy gradients, etc. for a more robust framework for application in trading.

7 Appendix

This work is produced using many Python libraries [7, 8, 9, 10, 11, 12]

Project GitHub Repo: `https://github.com/ggaoshen/su_cs230_project`

References

[1] Kavukcuoglu K. Silver D. et al Mnih, V. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 01 2015.

[2] Gordon Ritter. Machine learning for trading. *SSRN Electronic Journal*, 01 2017.

[3] Xiang Gao. Deep reinforcement learning for time series: playing idealized trading games. *Complexity*, 03 2018.

[4] Chien Huang. *Financial Trading as a Game: A Deep Reinforcement Learning Approach*, 07 2018.

[5] Yang Wang, Dong Wang, Shiyue Zhang, Yang Feng, Shiyao Li1, and Qiang Zhou. *Deep Q-trading*, 01 2017.

[6] Taewook Kim and Ha Kim. Optimizing the pairs-trading strategy using deep reinforcement learning with trading and stop-loss boundaries. *Complexity*, 2019:1–20, 11 2019.

[7] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[8] Fernando Pérez and Brian E. Granger. Ipython: A system for interactive scientific computing. *Computing in Science & Engineering*, 9(3):21–29, 2007.

[9] Wes McKinney. Data structures for statistical computing in python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.

[10] John D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[11] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[12] François Chollet et al. Keras. `https://keras.io`, 2015.