Deep Reinforcement Learning Exam
Team Drunken Policy

Thomas Lagos
Patrick Lucescu
Jakub Polak

12. December 2019

Contents

Contributions

All contributed equally on the take home exam questions. The individual contributions to the project were as following: Patrick did the Problem Setup and Downloaded and Preprocessed the Data, Thomas did the Reinforcement Learning Framework and Problem Setup and Jakub did the Architecture of Policy Network and Problem Setup. As far as the code is concerned all contributed equally.

1 Project: Portfolio Management Approach

Portfolio Management is the actions that a portfolio manager has to take in order to reallocate his wealth according to his views on the financial markets. For a behavioural manager this decisions are specific to his views on the market and thus they could differ from the optimal decision. For a quant manager this decisions are usually encoded into an algorithm that takes actions according to some strict rules.

With the rise of high frequency trading popularity (50% of all trading executed in the US in 2016) and machine learning techniques, more and more papers advertising machine learning approaches in high frequency scenarios appear every day. However, most of these approaches try to predict the future stock price of assets which does not yield acceptable results due to the low accuracy. Furthermore, these approaches are not entirely machine learning based as one has to implement a trading rule based on the predictions obtained.

Jiang, Xu, and Liang, 2017 try to overcome these issues by proposing a Reinforcement Learning framework. Consequently, the trading rule is integrated into the RL environment: the chosen weights represent action taken by the RL agent. They implement this approach in the context of cryptocurrencies markets and show that this approach is able to outperform all the traditional portfolio-selection methods taken into consideration.

As the author claim that their approach can be extended to any market, we test the performance of this RL framework in the context of equity markets. We choose our universe of stock 30 large companies listed on stock exchanges in the United States as they are highly traded. Our results show that the network takes a considerable time to train but it seems that it is able to achieve positive returns. However, we are unable at this stage to say whether or not we are outperform our equally weighted benchmark. Once the experiment is concluded we will discuss in more detail in the results part of the paper.

1.1 Problem Setup

The process of continuous reallocation of capital into a number of financial assets can be mathematically well described. The series of assumptions described here governed our problem parameters and choices of algorithm and architecture.

Another extension we implemented is using the framework on intra-day data with a daily re-balancing. Therefore, at the beginning of the period, the agent chooses weights for different assets in his portfolio. He observes the price changes during the day and at the end of the day we evaluate the portfolio's performance. Then, he may re-balance his portfolio again on the beginning of next day by choosing different weight.

The portfolio consists of m assets. First is so called risk-less asset or cash and the remaining assets are the risky stocks. The prices of all assets for period t are contained in the vector $\boldsymbol{v}_t$. The relative price vector $\boldsymbol{y}_t$ is the element-wise (asset-wise) division of $\boldsymbol{v}_t$ by $\boldsymbol{v}_{t-1}$. The portfolio weight vector $\boldsymbol{w}_t$ contains the proportions of how capital should be allocated into the m assets. This weights are normalized hence sum up to 1. By convention it's initialized as $\boldsymbol{w}_0 = (1, 0, \ldots, 0)^T$ and $\boldsymbol{w}_{t-1}$ contains the weight allocation at the beginning of the period t. The portfolio value at the beginning of period t is p_{t-1}. The portfolio value at the end of period t is then $p_t = p_{t-1}\boldsymbol{y}_t \cdot \boldsymbol{w}_{t-1}$. Finally, the return for the period t is just $r_t := \ln \boldsymbol{y}_t \cdot \boldsymbol{w}_{t-1}$. If there is no transaction cost, the final portfolio value will be $p_\mathrm{f} = p_0 \sum_{t=1}^{t_\mathrm{f}+1} \boldsymbol{y}_t \cdot \boldsymbol{w}_{t-1}$, where p_0 is the initial investment. Hence, we would like to maximize p_f for a given time frame.

Transaction costs are implemented according to Ormos and Urbán, 2013. They argue in favour of so called transaction remainder factor μ_t which essentially replicates paying all transaction fees for the re-balance by shrinking the portfolio value by factor μ_t. This factor is a function of $\mu_t = \mu_t(\boldsymbol{w}_{t-1}, \boldsymbol{w}_t, \boldsymbol{y}_t)$ and can be approximated by iterative estimation.

1.2 Data Preprocessing

Our data set consists of all index constituents of the Dow Jones Industrial Average that were part of the index on the first of July. The period of interest is from the 1^{st} of July 2018 to 29^{th} of November 2019. The last month of data is used for back-testing and evaluation of performance.

As machine learning techniques require large amounts of data to train we decided to use intra-day minute data which was obtained from Bloomberg. The reason behind the choice of this particular index is the higher trading volume which implies better market liquidity of those assets. Furthermore, due to the noise in bid and ask quotes we were forced to use trading data. The data downloaded included Open, Close, High and Low prices (OCHL) for each minute. We used only data which was available during the trading hours of the exchange. The missing data was forward filled: the missing price at time t was filled with the most recent price, which was available at time s, where $s < t$. This was done so we would not introduce any forward looking bias in our analysis.

The price tensor $\boldsymbol{X}_t$ which we use as an input to a neural network at the end of each period t has shape (f, n, m), where m is the number of assets, n is the number of input periods before t, and f is the feature number. Our number of features is $f = 4$ as it corresponds to OCHL prices, on the contrary to original paper which considered only CHL prices. This is due to different nature of markets in which we trade. The number of input periods is also different due to differing re-balancing periods. For daily re-balancing we use $n = 10$ which corresponds to the past 2 weeks of trading. The number of assets is $m = 31$

which is the full Dow Jones index and cash (not investing), but also a smaller subsets of assets is considered.

1.3 Reinforcement Learning Framework

Keeping in mind the problem definition above, we aim to build an automated agent which best allocates the weights of its investment between different stocks. Here we explicitly specify the reinforcement learning framework and deterministic policy gradient algorithm. Agent can be though of as the software portfolio manager performing trading-actions in the environment of financial markets. Environment comprises all available stocks in the market and the implicit expectations of all market participants towards them.

By the setup above, the agent's trading action made at the beginning of period t will affect the reward for period $t+1$ and as a result will affect the decision of its action. Hence, the action at time t can be represented solely by the portfolio vector $\boldsymbol{w}_t$ as:

$$\boldsymbol{a}_t = \boldsymbol{w}_t$$

The previous actions influence the decisions of the current actions. This influence is embedded by considering $\boldsymbol{w}_{t-1}$ as a part of the environment and inputting it to the agent's action making policy. Therefore, the state space at time t is represented as the pair of price tensor $\boldsymbol{X}_t$ and internal state represented by the portfolio vector from the last period $\boldsymbol{w}_{t-1}$. We assume the portfolio amount is negligible compared to the total trading volume and p_t is not included in the internal state. The state representation is then :

$$\boldsymbol{s}_t = (\boldsymbol{X}_t, \boldsymbol{w}_{t-1})$$

The goal of the agent is to maximize the final portfolio value p_f at the final period $t_\mathrm{f}+1$. This is then equivalent to maximizing the average cumulative return R or cumulative reward given as:

$$R(\boldsymbol{s}_1, \boldsymbol{a}_1, \ldots, \boldsymbol{s}_{t_\mathrm{f}}, \boldsymbol{a}_{t_\mathrm{f}}, \boldsymbol{s}_{t_\mathrm{f}+1}) = \frac{1}{t_\mathrm{f}} \sum_{t=1}^{t_\mathrm{f}+1} \ln(\mu_t \boldsymbol{y}_t \cdot \boldsymbol{w}_{t-1}) = \frac{1}{t_\mathrm{f}} \sum_{t=1}^{t_\mathrm{f}+1} r_t$$

We also consider the immediate reward for an individual episode, so called episodic reward which is r_t/t_f. The denominator t_f guarantees the fairness of the reward function between runs of different lengths, enabling it to train the trading policy in mini-batches. This distinguishes our setting from other RL problems. One advantage is that both episodic and cumulative rewards are exactly expressed and can be fully exploited by the agent. Second advantage is that all episodic rewards are equally important to the final return and hence the discounted factor is set to 0. Hence we are taking no consideration of future influence of the action. This make intuitive sense in the financial market due to the assumption of our trading not influencing prices of the assets.

1.4 Architecture of Policy Network

A policy is a mapping from the state space to the action space, $\pi : \mathcal{S} \rightarrow \mathcal{A}$. Due to the full exploitation setting in the current framework an action is deterministically produced by the policy from a state. Then, the optimal policy is obtained using a simple gradient ascent algorithm. The policy is specified by a parameter $\boldsymbol{\theta}$ which governs the optimal choice of action as

$$\boldsymbol{a}_t = \pi_{\boldsymbol{\theta}}(\boldsymbol{s}_t)$$

The performance metric J (value function) is in our case the corresponding reward function for the time interval $[0, t_{\mathrm{f}}]$ given the actions are chosen by the policy. It can be written as

$$J_{[0,t_{\mathrm{f}}]}(\pi_{\boldsymbol{\theta}}) = R\left(\boldsymbol{s}_1, \pi_{\boldsymbol{\theta}}(\boldsymbol{s}_1), \ldots, \boldsymbol{s}_{t_{\mathrm{f}}}, \pi_{\boldsymbol{\theta}}(\boldsymbol{s}_{t_{\mathrm{f}}}), \boldsymbol{s}_{t_{\mathrm{f}}+1}\right)$$

The parameters are continuously updated along the gradient direction with a learning rate λ, such that

$$\boldsymbol{\theta} \rightarrow \boldsymbol{\theta} + \lambda \nabla_{\boldsymbol{\theta}} J_{[0,t_{\mathrm{f}}]}(\pi_{\boldsymbol{\theta}})$$

Furthermore, this setup can be easily extended to mini-batch learning to increase training efficiency, by replacing the full time period $[0, t_{\mathrm{f}}]$ with mini-batch time period $[t_{\mathrm{b}_1}, t_{\mathrm{b}_2}]$. Major benefit of this setting is that it gives an intuitive sense how market works and trader would similarly learn and evaluate his performance and it also gives less hyper-parameters to fine-tune.

The policy function $\pi_{\boldsymbol{\theta}}$ is constructed with a deep neural network. The input to the network is the price tensor $\boldsymbol{X}_t$ and the output of the network is the portfolio vector $\boldsymbol{w}_t$. The last hidden layer l_L contains the scores for each of the stocks m and a cash bias (cash not invested in stocks). The softmax function then applied to these scores and cash bias to obtain actual corresponding portfolio weights for each asset $m + 1$. The second-last hidden layer l_{L-1} contains feature maps of size $(m, 1)$ and a portfolio weight vector $\boldsymbol{w}$ from the beginning of the period. The number of feature maps in this layer is not fixed as it depend on the previous layers. The previous layers are all Convolution layers with one important property, such that the tensor for each of the m stocks (which now will be a matrix) flows through the network identically and independently. This creates a structure of m smaller networks each evaluating the performance of one asset and combining/ensembling them only in the second-last layer l_{L-1}. This provides an additional structure and interpretability of the network, reduces the total number of network's parameters and enables to each individual network learn about their particular stocks behaviour and later combine them. In the initial tests, this structure significantly outperformed classical convolution layers or fully connected layers.

1.5 Training and Testing of Agent

1.6 Discussion of Results

1.7 Conclusion

2 Learning under Exploration and Exploitation

We start by decomposing the individual statements by Larry, Moe and Curly.

Larry: "Learning can always occur, whether exploring or exploiting, as long as the agent encounters something unexpected."

Moe: "Learning can occur whether expectations are met or not, but only during exploration."

Curly: "Learning can occur under any circumstances, but you have to expand the state space"

To make a general statement about their claims, they are all wrong in some way and correct in some way. First and foremost, exploration and exploitation are not two different discrete phases, but rather a continuous trade-off. In general, if agent's actions have more variance and learning rate is lower, then the agent does more of exploration and discovers greater part of state space. Whereas, if the agent's actions have small variance, he does more of exploitation as is likely to follow a path of positive rewards. But the agent learns in either case. This of course depends on a particular Reinforcement Learning Algorithm that is being used, but we try to provide answers that hold in more general frameworks.

First, lets describe what encountering something unexpected means. In our understanding, encountering something unexpected means to observe a state and choosing an action which yields different reward than what we previously experienced or is in contrast to the expected value. So for Larry's statement, if the agent encounters something unexpected and let's say receives a negative reward, then he learns that the action that brought him into that situation was sub-optimal (even though it was previously considered as the action with the highest expected value). Hence, this part is correctly claimed by Larry.

However, Moe correctly points out the gap in Larry's claim by stating that learning can also occur whether expectations are met or not. If the agent expects to receive a positive reward on a next action and this comes true, then it reinforces his learning that the action that brought him to that state was likely optimal. This increases his weights for that particular path or increases the value function for those states even further. That results in learning as he would be more likely to follow that path in next iterations. Although, Moe is wrong in stating that it is exclusive to exploration. If the agent continues to exploit a particular path/strategy, he either converges towards the optimal policy or overfits. In both cases the agent still improves his performance and continues to learn, but for the latter case his strategy just does not generalise well.

Hence, we agree with Curly's first part of the claim that learning *can* occur under any circumstances. The latter part stating that learning is conditioned

on that one has to expand the state space is wrong. In all above examples we weren't expanding state space and still demonstrated the agent's ability to learn. Here, under expanding state space, we understand either including more information in the agent state information or increasing the number of elements in state space which are available to the agent. However, it is correct that expanding the state space can support the agent's ability to learn.

3 Dynamic Problem

To answer this problem we will first break down its description into smaller parts. Initially the problem states that we have a random number of episodes. This means that we do not know when the regime switch happens. Next we are given that our problem includes regime shifts (e.g. change of rewards and transition probabilities in the box world) which makes our problem non-stationary. Afterwards, we are given that the number of regimes are finite, but the way each time the regime is chosen depends on a probability distribution. In case we would try to model the probability with which each regime appears, the finiteness of the regimes would be useful, since we could learn about each regime and store what we learned. However, we encounter an additional problem since we don't know when the regime changes (the number of episodes for each regime are random). So what we could do is to give more weight to more recent rewards compared to past rewards. Hence, we claim that it is possible to have an algorithm that could develop a good policy for the dynamic problem and our algorithm is based on Sutton and Barto, 1998.

Let R_i denote the reward received after the i-th selection of a specific action and Q_{n+1} be the estimate of the value of this action after it has been selected n times. Then in a stationary problem we could say that:

$$Q_{n+1} \doteq \frac{R_1 + R_2 + \cdots + R_n}{n} = \cdots = Q_n + \frac{1}{n}\left[R_n - Q_n\right] \tag{1}$$

Now that we want to give more weight to more recent rewards one standard way to do it is by using:

$$Q_{n+1} \doteq Q_n + \alpha\left[R_n - Q_n\right] = \cdots = (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} R_i \tag{2}$$

where $\alpha \in (0, 1]$ represents the learning rate and is constant.

A more general version of (2) is

$$Q_{n+1} = Q_n + \alpha_n(a)\left[R_n - Q_n\right] \tag{3}$$

and according to Robbins-Monro algorithm we have convergence if the sequence α_n satisfies:

$$\sum_{n=0}^{\infty} a_n = \infty \quad \text{and} \quad \sum_{n=0}^{\infty} a_n^2 < \infty \tag{4}$$

where the first condition guarantees that the steps are big enough to overcome any initial conditions or random fluctuations. The second condition guarantees that the steps become small enough to ensure convergence.

Equation (2) represent a weighted average with more weight towards the more recent rewards. This is clear since $(1-\alpha)^n + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} = 1$ and the weight decreases as the reward becomes older, since $\alpha(1-\alpha)^{n-i}$ increases as i increases. In RL as we learn more and more it is useful to change the learning rate. For instance let $\alpha_n(a)$ represent the learning rate for the n-th

selection of action a. If we choose $\alpha_n(a) = \frac{1}{n}$ then we get the weighted average equation (1), for which we know that (4) is satisfied, since both convergence criteria are satisfied. For different choices of $\alpha_n(a)$ convergence may not occur. More specifically, in our proposed solution, where the learning rate is constant ($\alpha_n(a) = a$), the second condition of (4) is not met. The fact that we do not have convergence is desirable in our case since we constantly switch regimes. In particular, if it were to converge the algorithm would find the optimal solution and exploit always the same optimal path by not taking into account the regime switch.

4 Supervised vs Reinforcement Learning

We split the answer of this question in two parts. One very simplistic and one more detailed which extends the simplistic. If my grandmother would ask me to describe the two different methods in a nutshell, I would provide her with the short answer. If a CEO would ask me for additional information, I would complement the answer with the second paragraph.

4.1 Short answer

Both approaches attempt to learn particular task from data automatically, i.e. without giving them explicit instructions how to do that. A computer using Reinforcement Learning attempts to learn what to do (actions) by trial and error. The goal is to learn the best behavior (optimal strategy) by trying different things (sampling actions) and then observing which behavior leads to the desired outcome. On the other hand, in supervised learning the computer will learn what to do based on examples which contain pairs of "questions" (inputs) with correct answers (labels). It is a learning setting where teacher immediately provides the correct answer. In contrast to the supervised approach, in Reinforcement learning the computer learns the optimal action not from a question and answer (labels) but from a time-delayed revealed answer to the desired question called a reward. The reward, which is modeled by a number, tells us whether the outcome of whatever we did was good or bad. Hence, the goal of Reinforcement Learning is to take actions in order to maximize the reward.

4.2 Complementary answer

Supervised learning tries to infer a functional relationship and produces a generalized formula by learning from the training data. The training data usually contains observations as pairs of features and labels. Features are derived quantities or indirect observations which often significantly compress the information content of measurements and labels are the true conclusions that can be drawn from the measurements which we try to abstract.

Meanwhile, Reinforcement Learning usually solves Markov's Decision process. Markov's decision processes provide a mathematical framework for modeling and decision making situations. It is a tuple that contains states (our situation in the world), actions (things we can do while being in state), transitions (what happens when taking an action in a state), discount factor (how important are future rewards) and rewards (what outcome we receive). The goal is arriving at an optimal policy for maximum discounted rewards over time. The agent interacts with the environment in discrete steps i.e., the agent makes an observation for every time period and receives a reward for every observation with the ultimate goal to collect the highest possible aggregated reward.

The most widely used learning algorithms for both Supervised learning and Reinforcement learning are Linear and Logistic Regression, Bayes Algorithm, Support Vector Machines, Decision trees, Neural Networks and many more which can be applied in problems which include different scenarios. Last but not lest, Deep Reinforcement learning simply means that the agent maps observations to actions via deep neural network.

References

Jiang, Zhengyao, Dixing Xu, and Jinjun Liang (2017). "A deep reinforcement learning framework for the financial portfolio management problem". In: *arXiv preprint arXiv:1706.10059*.

Ormos, Mihály and András Urbán (2013). "Performance analysis of log-optimal portfolio strategies with transaction costs". In: *Quantitative Finance* 13.10, pp. 1587–1597. DOI: 10.1080/14697688.2011.570368.

Sutton, Richard S. and Andrew G. Barto (1998). *Introduction to Reinforcement Learning*. 1st. Cambridge, MA, USA: MIT Press. ISBN: 0262193981.