RL Trading Report

Marko Vukovic

June 28, 2021

1 Code

All the code is available online at `https://github.com/mvkvc/RL_trading`. The data is hosted at and experiments were run at `https://dagshub.com/mvkvc/RL_trading`. These results can be recreated by following the instruction in the README using DVC.

2 Assignment

The objective of the assignment is to use reinforcement learning technique to design an algorithmic trading strategy on the Standard and Poor 500 index EFT. The restrictions on the environment taken from the assignment are as follows:

- Initial wealth is $10 on Jan. 3, 2012.
- Short sell is allowed, but you cannot hold more than 3 units of the ETF short positions in your portfolio.
- The transaction fee is 0.2% for each trade.
- You only do one trading per day, and are not allowed to sell or buy more than 5 units per day, i.e., our trading volume is between -5 to 5 units. If the trading volume is zero, it means no trade on that day. No fractional unit is traded.

The data is provided in two files train.csv and test.csv. The train file is used to train the agent which we then evaluate using the test file and record the Sharpe ratio and cumulative returns. The trading target is to maximize the 3-month return, but minimize the variance of the 3-month returns, i.e., we aim for a high and stable 3-month return in a long time horizon.

3 Background

Reinforcement learning is about agents learning how to make optimal decisions through experience. Agents learn to maximize a long-term reward that is the calculated discounted sum of future rewards. Formally, at time t, the agent at current state st $\in$ S makes an action at $\in$ A, obtains a reward rt $\in$ R and the state of the world becomes st+1 $\in$ S. A policy is way to map from state to actopm, and the goal is to learn from past data (past actions, past rewards) how to find an optimal policy. Since data is available at sequential order. But the actions depends on the environment, thus an action at a certain state could give a different reward re-visiting the same state. More specifically, at time t

the learner takes an action at $\in$ A - the learner obtains a (short-term) reward rt $\in$ R - then the state of the world becomes st+1 $\in$ S The

The states S refer to the different situations the agent might be in. The policy can be deterministic, or probabilistic, so in many cases, agents will compute expected values of rewards, conditional on states and actions.

After time step t, the agent receives a reward rt. The goal is to maximize its cumulative reward in the long run, thus to maximize the expected return.

$G_t = \sum_{k=t+1}^{T} r_k$

The cumulative reward is computed starting from t. Sometimes the agents can receive running reward, associated to tasks where there is no notion of final time step, so we introduce the discounted return where 0 ¡ y ¡ 1 is the discount factor which gives more importance to recent reward (and can allow Gt to exist)

$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+1+k}$

To quantify the performance of an action, we introduce, as in the previous section, the action-function, or Q-value on S ×A:

$Q^{\pi}(s_t, a_t) = \mathbb{E}_{\mathbb{P}}[G_t \mid s_t, a_t, \pi]$

In order to maximize the reward, as in bandits, the optimal strategy is characterized by the optimal policies

$\pi^{\star}(s_t) = \underset{a \in \mathcal{A}}{\operatorname{argmax}} \{Q^{\star}(s_t, a)\}$

That function can be used to derive an optimal policy, and the optimal value function producing the best possible return (in sense of regret):

$Q^{\star}(s_t, a_t) = \max_{\pi \in \Pi} \{Q^{\pi}(s_t, a_t)\}$

4 Environment

We will now formulate the stock trading process and a reinforcement learning problem. At each time step t we select an action fron the set of all possible actions [-N, N] with N being the maximum allowed number of shares to trade per timestep. This action will affect our total position which can be long where hold some positive number of shares, neutral where we hold no shares, and short where we borrow a fixed volume under the obligation to buy it later at a future price.

At each step we choose the action a A from policy $(a|s)$, that is, the probability to choose a in s. $P(s'|s, a)$ is the transition probability of the assumed Markov process. R(s, a) is the reward function. At each step the agent becomes the reward depending, not only on thecurrent, but also on the previous actions

This environment can be structured in a standard OpenAI gym instance which create which takes actions as inputs and outputs information including the reward, the next state, and if the episode has terminated. Building on this framework there is an open-source library named FinRL that is specialized in the stock trading environment. The stock trading process is modelled as a Markov Decision Process (MDP) and we then formulate our trading goal as a maximization problem. The relevant components are as follows:

- Action: The action space describes the allowed actions that the agent interacts with the environment. Normally, a ∈ A includes three actions: a ∈ −1, 0, 1, where −1, 0, 1 represent selling, holding, and buying one stock. Also, an action can be carried upon multiple shares. We use an action space −k, ..., −1, 0, 1, ..., k, where k denotes the number of shares. For example, "Buy 10 shares of AAPL" or "Sell 10 shares of AAPL" are 10 or −10, respectively

- Reward function: r(s, a, s′) is the incentive mechanism for an agent to learn a better action. The change of the portfolio value when action a is taken at state s and arriving at new state s', i.e., r(s, a, s′) = v′ − v, where v′ and v represent the portfolio values at state s′ and s, respectively

- State: The state space describes the observations that the agent receives from the environment. Just as a human trader needs to analyze various information before executing a trade, so our trading agent observes many different features to better learn in an interactive environment.

5 Algorithms

5.1 DQN

Double Q-Networks (DQN) is an algorithm that adapts the concept of Q-learning while addressing known limitations. A Q-network is a network that takes any state-action pair Q(s, a)and returns the expected reward. DQN's utilize deep learning to calculate this expected reward and have been shown to be useful on a large variety of tasks. Double DQN's were developed to address the problem of DQN overestimating action values under certain conditions. This is addressed by having a target Q-network that executes the policy and is updated according to the training Q-network. By not training on the online agent we increase the stability of the policy.

Algorithm 1 Double Q-learning

Initialize Q^A,Q^B,s
repeat
  Choose a, based on $Q^A(s,\cdot)$ and $Q^B(s,\cdot)$, observe r, s'
  Choose (e.g. random) either UPDATE(A) or UPDATE(B)
  if UPDATE(A) **then**
    Define $a^* = \arg\max_a Q^A(s',a)$
    $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)\left(r + \gamma Q^B(s',a^*) - Q^A(s,a)\right)$
  else if UPDATE(B) **then**
    Define $b^* = \arg\max_a Q^B(s',a)$
    $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)(r + \gamma Q^A(s',b^*) - Q^B(s,a))$
  end if
  $s \leftarrow s'$
until end

5.2 DDPG

Deep Deterministic Policy Gradient (DDPG) is an algorithm which learns both a Q-function as well as a policy at the same time. It uses off-policy data to learn the Q-function, and then uses the Q-function to learn a policy. One main motivation of this approach is that with Q-networks that they need to evaluate the Q-value for every possible action which can be very computationally expensive. DDPG addresses this by having a policy network that learns a policy over the continuous action space to maximize the reward of the agent.

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

6 Results

Insert tables here ...

7 Future Work

This project was an initial look at using reinforcement learning for a trading environment. There are a large number of ways to improve and extend both the environment and agents used in this work. The first way is to increase the granularity of the data to hour or minute level data. If the data granularity increases then you canexplore the use of recurrent neural networks as opposed to the fully connected networks used here. Next you could utilize other algorithms such as Twin Delay DDPG (TD3). Another possibility is to frame the problem as a continuous portfolio optimization problem rather than discretely buying individual stocks. There a re endless possibilities for improvement and this is only one way to approach this problem.