Project on Stock Trading using Reinforcement Learning

Report

Abstract

We are making an AI project to predict the stock prices and accordingly decide on buying, selling or holding stock. The AI algorithm should be flexible to consider various trading environmental factors like stock price changes, latency, gaining protection from future price movements.

1 Introduction

Reinforcement Learning is a type of Machine Learning algorithm, which maps a particular action to each input state for a given set of data. A reward is obtained for every action, and actions are taken in such a way that rewards are maximized.

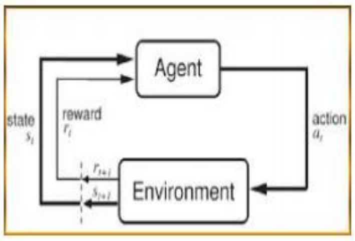


Figure 7.1: Basic Reinforcement Learning Model

The first step to reinforcement learning, ideally, is characterizing the stock data into a finite number of 'states'. To do this, we use the various technical factors derived, and they characterize the stock data from the past 'n' days. The number of possible states is thus dependant on the number of factors used, and the number of steps each factor is quantized into. If we use too many factors and/or too many quantization steps, the number of states shoots up drastically. This case is unwanted, as the amount of historical data is limited, and each state should occur a significant number of times throughout the history. On the other hand, if the number of states is too less then we don't have a significant basis to characterize the idiosyncrasies of the stock at any point of time. The optimal number of states is generally the number of training samples divided by the order of 10^ 2.

Now that we defined the state 's', we define 'π', the policy function which represents taking an action. Hence π(s) is an action taken on state 's'. Generally π(s) as a function just evaluates the value of all possible actions given the state s and returns the highest value action. We call the function that accepts a state s and returns the value of that state vπ(s). This is the value function. Similarly, there is an action-value function Q(s, a) that accepts a state 's' and an action 'a' and returns the value of taking that action given that state. The number of actions we can take at any point of time are 3, i.e. buy, sell and hold.

Q-learning, like virtually all RL methods, is one type of algorithm used to calculate state-action values. It falls under the class of temporal difference (TD) algorithms, which suggests that time differences between actions taken and rewards received are involved. With TD algorithms, we make updates after every action taken. In most cases, that makes more sense. We make a prediction (based on previous experience), take an action based on that prediction, receive a reward and then update our prediction. Now, there's a need to define a method to calculate rewards. To provide a simple idea to how our algorithm calculates rewards for a particular action taken at a given state, we shall consider the benefits of taking that particular action over the remaining actions. Hence, if our algorithm decides to buy shares at a particular time, the reward is calculated as a scaled version of the difference between our net worth when we buy and our net worth if we would have held, after a fixed number of days called the reward extent. If our algorithm decides to sell shares at any given point, the reward is a scaled version of the difference between our net worth when we sell and our net worth if we would have held shares until a fixed number of days. If our algorithm decides to hold shares, we subtract the net worth after a few days from the net worth after the more profitable of the two other actions.

2. RELATED WORKS

Recent applications of deep reinforcement learning in financial markets consider discrete or continuous state and action spaces, and employ one of these learning approaches: critic-only approach, actor-only approach, or actor-critic approach [20]. Learning models with continuous action space provide finer control capabilities than those with discrete action space. The critic-only learning approach, which is the most common, solves a discrete action space problem using, for example, Deep Q-learning (DQN) and its improvements, and trains an agent on a single stock or asset. The idea of the critic-only approach is to use a Qvalue function to learn the optimal action-selection policy that maximizes the expected future reward given the current state. Instead of calculating a state-action value table, DQN minimizes the error between estimated Q-value and target Q-value over a transition, and uses a neural network to perform function approximation. The major limitation of the critic-only approach is that it only works with discrete and finite state and action spaces, which is not practical for a large portfolio of stocks, since the prices are of course continuous.

3. PROBLEM DESCRIPTION

We model stock trading as a Markov Decision Process (MDP), and formulate our trading objective as a maximization of expected return .
+
MDP Model for Stock Trading

To model the stochastic nature of the dynamic stock market, we employ a Markov Decision Process (MDP) as follows:

• State s = [p, h, b]: a vector that includes stock prices p ∈ R D + , the stock shares h ∈ Z D + , and the remaining balance b ∈ R+, where D denotes the number of stocks and Z+ denotes non-negative integers.
• Action a: a vector of actions over D stocks. The allowed actions on each stock include selling, buying, or holding, which result in decreasing, increasing, and no change of the stock shares h, respectively.
• Reward r(s, a, s0): the direct reward of taking action a at state s and arriving at the new state s 0 .• Policy π(s): the trading strategy at state s, which is the probability distribution of actions at state s.
• Q-value Qπ(s, a): the expected reward of taking action a at state s following policy π.

The state transition of a stock trading process is shown in Figure 2. At each state, one of three possible actions is taken on stock d (d = 1, ..., D) in the portfolio.

• Selling k[d] ∈ [1, h[d]] shares results in ht+1[d] = ht[d] − k[d], where k[d] ∈ Z+ and d = 1, ..., D.

• Holding, ht+1[d] = ht[d].

• Buying k[d] shares results in ht+1[d] = ht[d] + k[d]. At time t an action is taken and the stock prices update at t+1, accordingly the portfolio values may change from "portfolio value 0" to "portfolio value 1", "portfolio value 2", or "portfolio value 3", respectively, as illustrated in Figure 2. Note that the portfolio value is p T h + b.

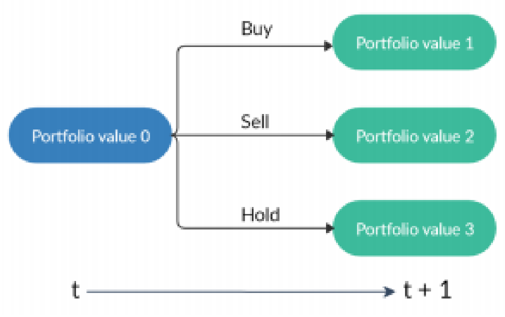


Fig. 2. A starting portfolio value with three actions result in three possible portfolios. Note that "hold" may lead to different portfolio values due to the changing stock prices.

4. Q-learning

Q-learning is a model-free reinforcement learning algorithm to learn the value of an action in a particular state. It does not require a model of the environment (hence "model-free"), and it can handle problems with stochastic transitions and rewards without requiring adaptations.

For any finite Markov decision process (FMDP), Q-learning finds an optimal policy in the sense of maximizing the expected value of the total reward over any and all successive steps, starting from the current state.[1] Q-learning can identify an optimal action-selection policy for any given FMDP, given infinite exploration time and a partly-random policy.[1] "Q" refers to the function that the algorithm computes - the expected rewards for an action taken in a given state.

4.1 Algorithm

After Δt steps into the future the agent will decide some next step. The weight for this step is calculated as $\gamma\Delta t$, where γ (the *discount factor*) is a number between 0 and 1 ($0 \leq \gamma \leq 1$) and has the effect of valuing rewards received earlier higher than those received later (reflecting the value of a "good start"). γ may also be interpreted as the probability to succeed (or survive) at every step Δt.

The algorithm, therefore, has a function that calculates the quality of a state–action combination:

$$Q : S \times A \rightarrow \mathbb{R}.$$

Before learning begins, **Q** is initialized to a possibly arbitrary fixed value (chosen by the programmer). Then, at each time **t** the agent selects an action $\mathbf{a_t}$, observes a reward $\mathbf{r_t}$, enters a new state $\mathbf{s_{t+1}}$ (that may depend on both the previous state $\mathbf{s_t}$ and the selected action), and is updated. The core of the **Q** algorithm is a Bellman equation as a simple value iteration update, using the weighted average of the old value and the new information:

$$Q^{new}(s_t,a_t) \leftarrow \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\bigg(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t,a_t)}_{\text{old value}}\bigg)}^{\text{temporal difference}}$$

where $\mathbf{r_t}$ is the reward received when moving from the state $\mathbf{s_t}$ to the state $\mathbf{s_{t+1}}$, and α is the learning rate ($0 < \alpha \leq 1$).

Note that $\mathbf{Q}^{new}$ (s_t,a_t) is the sum of three factors:

- **(1-α) Q(s_t,a_t):** the current value weighted by the learning rate. Values of the learning rate near to 1 made faster the changes in Q.
- **α r_t**: the reward $\mathbf{r_t = r(s_t,a_t)}$ to obtain if action is taken when in state (weighted by learning rate)
- **αγ max Q(s_{t+1},a):** the maximum reward that can be obtained from state (weighted by learning rate and discount factor)

An episode of the algorithm ends when state s_{t+1} is a final or *terminal state*. However, *Q*-learning can also learn in non-episodic tasks.[*citation needed*] If the discount factor is lower than 1, the action values are finite even if the problem can contain infinite loops.

For all final states $\mathbf{s_f}$,$\mathbf{Q(s_f,a)}$ is never updated, but is set to the reward value r observed for state s_f. In most cases $\mathbf{Q(s_f,a)}$, can be taken to equal zero.

4.2 Influence of variables

Learning Rate

The learning rate or *step size* determines to what extent newly acquired information overrides old information. A factor of 0 makes the agent learn nothing (exclusively exploiting prior knowledge), while a factor of 1 makes the agent consider only the most recent information (ignoring prior knowledge to explore possibilities).

In practice, often a constant learning rate is used, such as $\alpha_t = 0.1$ for all t.

Discount factor

The discount factor γ determines the importance of future rewards. A factor of 0 will make the agent "myopic" (or short-sighted) by only considering current rewards, i.e. (in the update rule above), while a factor approaching 1 will make it strive for a long-term high reward. If the discount factor meets or exceeds 1, the action values may diverge. For $\gamma = 1$, without a terminal state, or if the agent never reaches one, all environment histories become infinitely long, and utilities with additive, undiscounted rewards generally become infinite.

Initial conditions (Q_0)

Since *Q*-learning is an iterative algorithm, it implicitly assumes an initial condition before the first update occurs. High initial values, also known as "optimistic initial conditions",[7] can encourage exploration: no matter what action is selected, the update rule will cause it to have lower values than the other alternative, thus increasing their choice probability. The first reward **r** can be used to reset the initial conditions.[8] According to this idea, the first time an action is taken the reward is used to set the value of Q. This allows immediate learning in case of fixed deterministic rewards.

4.3 Implementation

Q-learning at its simplest stores data in tables. This approach falters with increasing numbers of states/actions since the likelihood of the agent visiting a particular state and performing a particular action is increasingly small.

Function approximation

Q-learning can be combined with function approximation.[9] This makes it possible to apply the algorithm to larger problems, even when the state space is continuous.

One solution is to use an (adapted) artificial neural network as a function approximator.[10] Function approximation may speed up learning in finite problems, due to the fact that the algorithm can generalize earlier experiences to previously unseen states.

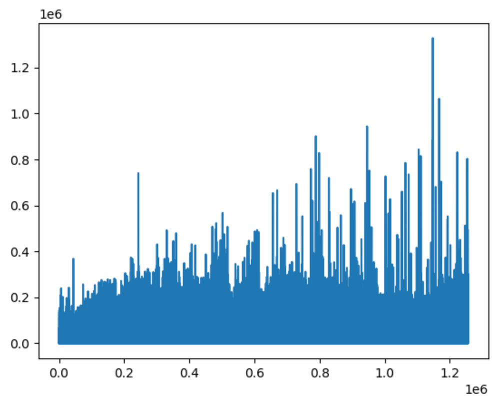
Quantization

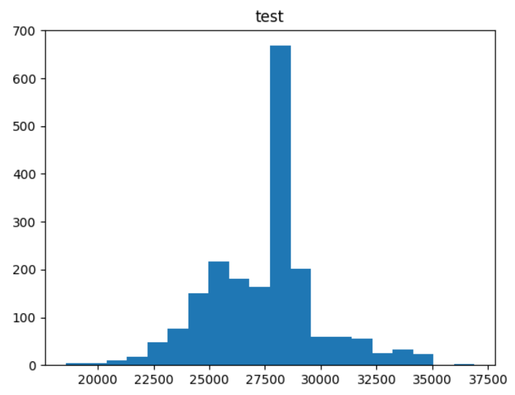
Another technique to decrease the state/action space quantizes possible values. Consider the example of learning to balance a stick on a finger. To describe a state at a certain point in time involves the position of the finger in space, its velocity, the angle of the stick and the angular velocity of the stick. This yields a four-element vector that describes one state, i.e. a snapshot of one state encoded into four values. The problem is that infinitely many possible states are present. To shrink the possible space of valid actions multiple values can be assigned to a bucket. The exact distance of the finger from its starting position (-Infinity to Infinity) is not known, but rather whether it is far away or not (Near, Far).+

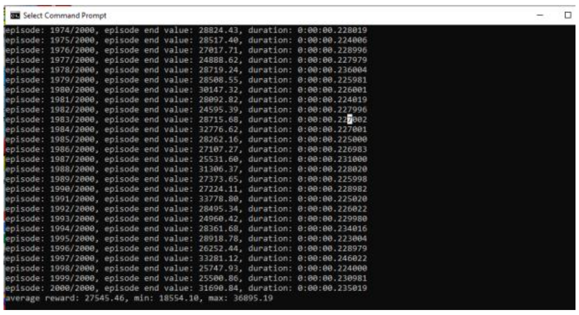
5. Testing and results

Now, we move on to the testing period. As the Q-table contains the weighted average of rewards, a higher maximum Q value for a particular action should mean that the action taken should be more rigorous. Hence, we should buy more if the max Q value for buy is relatively higher for one state than another. Using this property to the best of our interest, we basically normalize the maximum Q-value amongst that of the 3 actions, and convert it to an integer between 0 and 1. Once this is done, we multiply this normalized value to a particular amount of cash we have set aside for transactions, and buy/sell this multiplied value. Once this is done, the transaction is complete for the particular day.

In the same way, the algorithm runs for each day in the training period and updates the Qtable. According to this table, our algorithm takes actions for each day in the testing period. Our results plot the stock price change throughout time (both testing and training periods) and the net worth change in the testing period. These results are plotted for stock data of different companies.

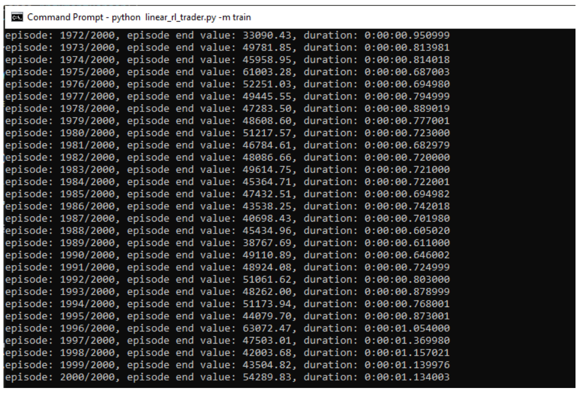






Select Command Prompt

```
episode: 1974/2000, episode end value: 28824.43, duration: 0:00:00.228019
episode: 1975/2000, episode end value: 28517.40, duration: 0:00:00.224006
episode: 1976/2000, episode end value: 27017.71, duration: 0:00:00.228996
episode: 1977/2000, episode end value: 24888.62, duration: 0:00:00.227979
episode: 1978/2000, episode end value: 28719.24, duration: 0:00:00.236004
episode: 1979/2000, episode end value: 28508.55, duration: 0:00:00.225981
episode: 1980/2000, episode end value: 30147.32, duration: 0:00:00.226001
episode: 1981/2000, episode end value: 28092.82, duration: 0:00:00.224019
episode: 1982/2000, episode end value: 24595.39, duration: 0:00:00.227996
episode: 1983/2000, episode end value: 28715.68, duration: 0:00:00.227002
episode: 1984/2000, episode end value: 32776.62, duration: 0:00:00.227001
episode: 1985/2000, episode end value: 28262.16, duration: 0:00:00.225000
episode: 1986/2000, episode end value: 27107.27, duration: 0:00:00.226983
episode: 1987/2000, episode end value: 25531.60, duration: 0:00:00.231000
episode: 1988/2000, episode end value: 31306.37, duration: 0:00:00.228020
episode: 1989/2000, episode end value: 27373.65, duration: 0:00:00.225998
episode: 1990/2000, episode end value: 27224.11, duration: 0:00:00.228982
episode: 1991/2000, episode end value: 33778.80, duration: 0:00:00.225020
episode: 1992/2000, episode end value: 28495.34, duration: 0:00:00.226022
episode: 1993/2000, episode end value: 24960.42, duration: 0:00:00.229980
episode: 1994/2000, episode end value: 28361.68, duration: 0:00:00.234016
episode: 1995/2000, episode end value: 28918.78, duration: 0:00:00.223004
episode: 1996/2000, episode end value: 26252.44, duration: 0:00:00.228979
episode: 1997/2000, episode end value: 33281.12, duration: 0:00:00.246022
episode: 1998/2000, episode end value: 25747.93, duration: 0:00:00.224000
episode: 1999/2000, episode end value: 25500.86, duration: 0:00:00.230981
episode: 2000/2000, episode end value: 31690.84, duration: 0:00:00.235019
average reward: 27545.46, min: 18554.10, max: 36895.19
```



Command Prompt - python linear_rl_trader.py -m train

```
episode: 1972/2000, episode end value: 33090.43, duration: 0:00:00.950999
episode: 1973/2000, episode end value: 49781.85, duration: 0:00:00.813981
episode: 1974/2000, episode end value: 45958.95, duration: 0:00:00.814018
episode: 1975/2000, episode end value: 61003.28, duration: 0:00:00.687003
episode: 1976/2000, episode end value: 52251.03, duration: 0:00:00.694980
episode: 1977/2000, episode end value: 49445.55, duration: 0:00:00.794999
episode: 1978/2000, episode end value: 47283.50, duration: 0:00:00.889019
episode: 1979/2000, episode end value: 48608.60, duration: 0:00:00.777001
episode: 1980/2000, episode end value: 51217.57, duration: 0:00:00.723000
episode: 1981/2000, episode end value: 46784.61, duration: 0:00:00.682979
episode: 1982/2000, episode end value: 48086.66, duration: 0:00:00.720000
episode: 1983/2000, episode end value: 49614.75, duration: 0:00:00.721000
episode: 1984/2000, episode end value: 45364.71, duration: 0:00:00.722001
episode: 1985/2000, episode end value: 47432.51, duration: 0:00:00.694982
episode: 1986/2000, episode end value: 43538.25, duration: 0:00:00.742018
episode: 1987/2000, episode end value: 40698.43, duration: 0:00:00.701980
episode: 1988/2000, episode end value: 45434.96, duration: 0:00:00.605020
episode: 1989/2000, episode end value: 38767.69, duration: 0:00:00.611000
episode: 1990/2000, episode end value: 49110.89, duration: 0:00:00.646002
episode: 1991/2000, episode end value: 48924.08, duration: 0:00:00.724999
episode: 1992/2000, episode end value: 51061.62, duration: 0:00:00.803000
episode: 1993/2000, episode end value: 48262.00, duration: 0:00:00.878999
episode: 1994/2000, episode end value: 51173.94, duration: 0:00:00.768001
episode: 1995/2000, episode end value: 44079.70, duration: 0:00:00.873001
episode: 1996/2000, episode end value: 63072.47, duration: 0:00:01.054000
episode: 1997/2000, episode end value: 47503.01, duration: 0:00:01.369980
episode: 1998/2000, episode end value: 42003.68, duration: 0:00:01.157021
episode: 1999/2000, episode end value: 43504.82, duration: 0:00:01.139976
episode: 2000/2000, episode end value: 54289.83, duration: 0:00:01.134003
```

As seen from the figures, Reinforcement learning is beneficial in most cases, as net worth is trending positive most of the time. However, a major issue faced on simulating the algorithm over a large range of companies was that the net worth tends to follow the stock data in quite a few cases. Hence, the parameters we feed for every stock are different and relative to the properties of each stock. This leaves a scope for improvement in the future, as the parameters fed to the RL algorithm can be made adaptive, and can change with respect to the fundamental factors of each stock.

Reinforcement learning is ideally a case of unsupervised learning and hence requires an ample amount of data to ensure proper training. In case data seems to be less we propose a method of combining a supervised learning algorithm to be used to obtain a model for the reward to develop a relationship with the state. In the case of trading, majority of the points occur as beneficial holding points as compared to buying and selling points since the technical indicators or statistical oscillators (states for Reinforcement Learning model) that are used in finance lag behind the price fluctuations. Since the occurrence of buying and selling points is very less as compared to the length of the data a model for calculating the reward can be derived based on a linear relationship between technical factors and profit over the portfolio. Thus Multivariate Gradient Descent Algorithm [3] can be used to train on a data74 that we can log from the branching model [6.12] to establish a relationship between rewards and states.

Where, θi for i=1, 2…, number of factors are the weights that will be assigned by the Gradient Descent Algorithm.

6. References

- https://www.analyticsvidhya.com/blog/2021/01/bear-run-or-bull-run-can-reinforcement-learning-help-in-automated-trading/
- https://towardsdatascience.com/deep-reinforcement-learning-for-automated-stock-trading-f1dad0126a02
- https://github.com/danx12/kalimba-sublime
- https://github.com/lazyprogrammer/machine_learning_examples/tree/master/rl
- https://github.com/henryboisdequin/AI-Stock-Trader
- https://www.udemy.com/course/artificial-intelligence-reinforcement-learning-in-python/

7. Equations

$$v(t) = \mu v(t-1) - \eta g(t)$$

$$g(t) = gradient\ at\ time\ t$$

$$\mu = momentum\ term\ (0.9,\ 0.99,\ etc.)$$

$$\eta = learning\ rate$$

$$\theta(t) = \theta(t-1) + v(t),\ \theta = \{W, b\}$$

$$\boxed{\begin{aligned} J &= \frac{1}{NK}\sum_{n=1}^{N}\sum_{k=1}^{K}(y_{nk} - \hat{y}_{nk})^2 \\ shape(y) &= shape(\hat{y}) = (N, K) \end{aligned}}$$

$$\frac{\partial J}{\partial \theta} = \frac{2}{NK}\sum_{n=1}^{N}\sum_{k=1}^{K}(\hat{y}_{nk} - y_{nk})\frac{\partial \hat{y}_{nk}}{\partial \theta},\ for\ \theta = \{W, b\}$$