Machine Learning for Algorithmic Trading

Aryan Mathe

January 2023

Contents

1 Introduction

Here we start!!

1.1 What is this project about ?

This project basically involves using various **Reinforcement Learning** techniques to train an agent which will analyse the stock market environment and build a strategy to act accordingly, so as to maximize the total profits.

We will consider the stock market as a highly volatile, complex but deterministic environment and formalize that in the form of a Markov Decision Process. Since we have an MDP, we can use **Bellman** equation to train a bot which develops a strategy for trading in stock market.

1.2 How does the agent achieve that ?

In simple words, our agent is given an environment to act. Initially, it doesn't know anything, so it just acts randomly and then it receives some rewards (may be positive or negative), which directs its actions so as to achieve a maximum reward and in doing so it tends to develop an optimal strategy to play in the environment.

Generally, the **reward function** is designed so that the agent could be trained to reach the most optimal state of the environment and that could be a pretty daunting task.

1.3 How it is useful in Trading ?

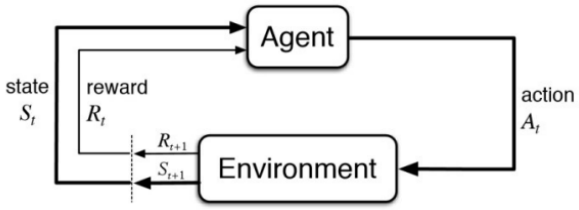
The idea to build a bot which learns a trading strategy model to act in the stock market is used because of the challenges that traders face while trading in the stock market, which involves

- Financial markets are highly dynamic and turbulent structures. Due to the complexity of the trading environment, it is hard to identify patterns and make quick decisions.
- Trading is a partially observable **Markov Decision Processes**, which makes it a suitable environment for our reinforcement learning agent.
- Error-free handling of large volumes of nearly continuous data and gathering unbiased representative financial data is one of the major challenges that traders face.
- Various short-term impacts, leading to a really complex trading environment.

2 Important Concepts Involved

2.1 Markov Decision Processes

- The formalism of state, policy and action in any environment is defined as an **MDP***(Markov Decision Process)*
- MDP can be stochastic as well as deterministic. Stochastic MDPs involve action over a state whose outcome is probabilistic. On the other hand, a deterministic MDP involves actions over the state of the environment whose outcomes are known with **certainity**.
- The following figure shows the designs of a simple MDP which forms the basis of many **reinforcement learning processes**



2.2 Reinforcement Learning Techniques

This section involves various reinforcement learning techniques which can be used to train our agent to act in a deterministic MDP environment, although they can be deployed in a stochastic MDP environment with some use of probabilities, But, here we are only dealing with deterministic environments.

2.2.1 Value Iteration

The following pseudo-code represents the **Value Iteration** method which is used to train the agent and optimize **Policy and Discounted returns**.

This is an **on-policy** reinforcement learning algorithm.

Algorithm 1 Value Iteration

Policy π: A data structure to store **optimal action values** corresponding to each states.
Value V: The **Expected Return** corresponding to each state
initialize **Policy** and **Value** with random values within the domain
while (**policy** not converges) **do**
  while (**Value** not converges) **do**
    for each $s \in S$ **do**
      $V(s) = max_{a \in A}\{R(s,a) + \gamma * V_*(s')\}$
    end for
  end while
  for each $s \in S$ **do**
    $\pi(s) = argmax_a\{R(s,a) + \gamma * V_*(s')\}$
  end for
end while
return (π, V)

2.2.2 TD(Lambda) Learning

Temporal Difference Learning has got many versions, here we will show just one variant of that algorithm, which is **TD(λ)**, the ultimate variant.

This is also an **on-policy** reinforcement learning algorithm.

Algorithm 2 TD(**Lambda**)

Policy π: A data structure to store **optimal action values** corresponding to each states.
Value V: The **Expected Return** corresponding to each state
initialize **Policy** and **Value** with random values within the domain
while (**policy** not converges) **do**

  while (**Value** not converges) **do**

    for each $k \in \{1..n\}$ **do**
      $R^n = \sum_{j=0}^{n} \{\gamma^j * r_{k+j} + \gamma^{n+1} * V(s_{k+n+1})\}$
      $R^\lambda = (1-\lambda) * \sum_{n=1}^{\infty} R^n$
      $V(s_k) = V(s_k) + \alpha * \{R^\lambda - V_*(s_k)\}$ (Updation Step)
    end for
  end while

  for each $s \in S$ **do**
    $\pi(s) = argmax_a\{R(s,a) + \gamma * V_*(s')\}$
  end for
end while
return (π, V)

2.2.3 Q-Learning

It is one of the most important algorithms used to train models because this is an **off-policy** reinforcement learning algorithm which doesn't require any policy to learn.
Q-Learning, DQN, Double-DQN are various variants of **Q-Learning** which gives pretty decent results after training through sufficient number of iterations.

Algorithm 3 General Q-Learning

1: Q-Value Q: Parameter to store the **expected returns** corresponding to each **state-action** pair
2: initialize **Q-Value** parameter with random values within the domain
3: **while** (**Q-Value** not converges) **do**
4: **for** each $s \in S$ **do**
5: Choose action a according to $\epsilon - greedy$ strategy
$Q(s,a) = Q(s,a) + \alpha * \{R(s,a) + \gamma * max_{a' \in A} Q_*(s',a') - Q(s,a)\}$
6: **end for**
Update ϵ following the exponential decay step
7: **end while**
8: return (Q)

There are some implementation differences in each of the Q-Learning techniques described below

- **Q-Learning**

  In this algorithm, the parameter for **Q-value** is in the form of a **(state** x **action)** matrix, which stores the discounted returns for each of the state - action pair.

- **DQN**

  Deep Q-Network, as the name suggests, it involves the use of a deep neural network to train the model parameters. The input and output of the neural network can be defined in various ways.
  One of them is to define the input as the current state and let the neural network output the **Q-value** corresponding to each of the actions that can be taken within this state.
  For the updation step, we make a copy of the online network called **target network** whose weights are updated every τ step. The weights of the target network are used to update the **Q-values** of the state-action pair according to the **Bellman Equation**.

$$Q^t - - > TargetNetwork$$

$$Q - - > OnlineNetwork$$

$$Q(s,a) = Q(s,a) + \alpha * \{R(s,a) + \gamma * max_{a' \in A} Q^t(s',a') - Q(s,a)\}$$

These neural networks can be trained using an optimizer to learn the optimal discounted returns.

- **Double-DQN**

  This technique was an improvement to the problem of overestimation for some non-optimal actions that occur in DQN. The following changes were made in **D-DQN**

  - The target network is only used to find the optimal action from the state s' and the **Q-value** for that action was found using the online network. The changes in the updation equation have been depicted in the equation below

    $$Q(s,a) = (1-\alpha)Q(s,a)+\alpha*\{R(s,a)+\gamma*Q(s', argmax_{a' \in A} Q^t(s', a'))\}$$

  - Using the above changes in the **Bellman Equation**, DDQN achieves **state-of-the-art** model in some of the **Atari** games.

3 Paper Reading

3.1 Deep Reinforcement Learning with Double Q-Learning

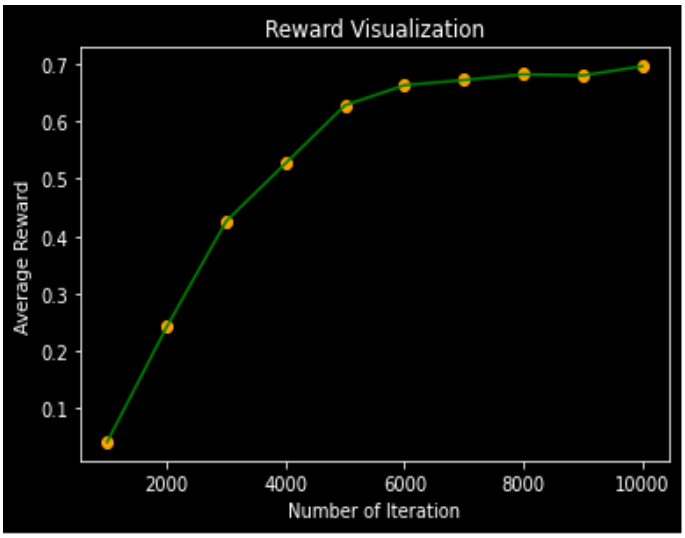
—Google DeepMind

- Understanding the difference between various Q Learning techniques and subtle differences in their implementations.
- The overestimation of non-optimal policies problem with previous Q-Learning algorithms and understanding the idea of Double Q-Learning to improve upon that.
- The paper shows that using DDQN can remove the bias from the optimal Q-function learnt by a significant amount, which leads to the achievement of **state-of-the-art** model on the Atari domain.

4 Implementation

4.1 Frozen Lake

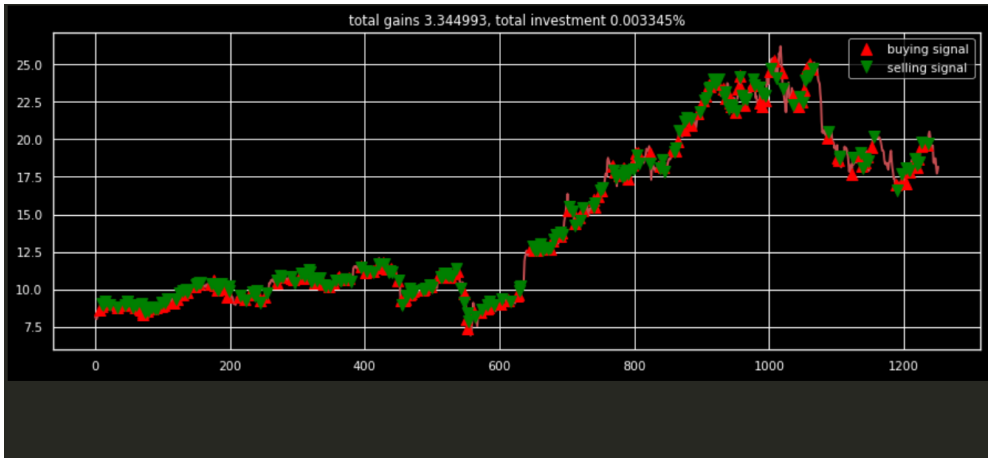
A model was trained using **Deep Q-Network** to play in the deterministic environmental game consisting of several **holes** (*high negative reward positions*) and a winning position(high positive reward position).



Game Sample

4.2 Analytics Vidya Trading Bot

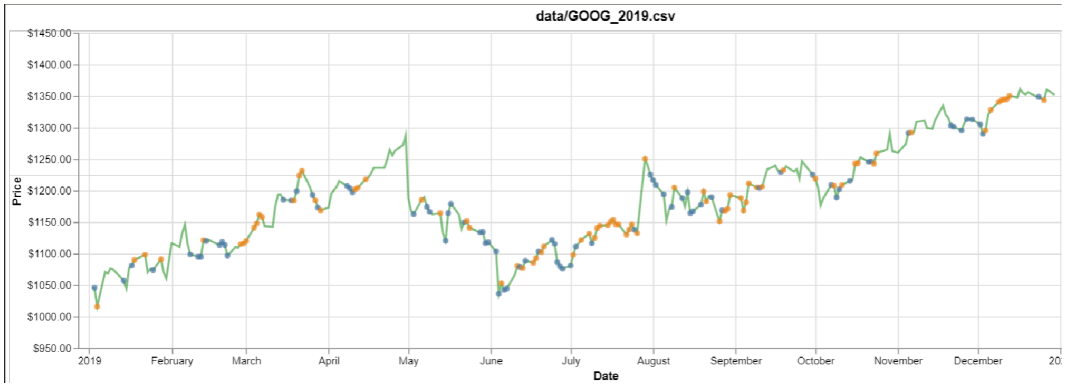
Implementation of a trading strategy model using **DQN** reinforcement learning algorithm. The aim is to trade on past data from **INOSYS** stocks over a fixed amount of time using the action space of [**buy, sell, hold**] and maximize the total profit.



Test Results

4.3 Impleneting Various Trading Strategy Q-Learning Techniques

Implementing and analysing the results of various reinforcement learning strategies like **dqn, t-dqn** and **d-dqn** by training and testing the model on the past stock data.



Test Results