Algorithmic Trading Using Reinforcement Learning

Nivedita Ganesh

nive@nyu.edu

Abstract. The project evaluates the efficiency of deep-q-networks in the task of automated trading. A MDP formulation of the trading task is proposed and the optimal action function is learnt using a neural network. Transaction costs and liquidity constraints are taken into account. The agents are trained and evaluated on real-life cryptocurrency instruments like Bitcoin and Ethereum. The results show that the proposed method does well on learning short term trends and are able to profit well.

1 Motivation

The impact of Automated Trading Systems (ATS) on financial markets is growing every year and the trades generated by an algorithm now account for the majority of orders that arrive at stock exchanges. In order to automate and accomplish a level of performance and generality similar to a human trader, our agents learn for themselves to create successful strategies that optimise short-term rewards. Due to the fact that we are looking at historical information, we will not be interested in making investment choices. Algortithmic trading is centralised over making short-term trades rather than investments. Trading decisions are far more dynamic because they have to take into account volatile current information, however investments look at the horizon and are less affected by day to day changes.

We will train an Agent to make automated trading decisions in a simulated stochastic market environment using Reinforcement Learning or Deep Q-Learning which is a form of semi-supervised learning.

A vast majority of Algorithmic trading comprises of statistical arbitrage / Relative Value strategies which are mostly based on convergence to mean, where the mean is derived from a randomly chosen sample of historical data.

Algorithmic trading primarily has two components: Policy and Mechanism. The policy is chosen by the traders and the mechanism is implemented by the machines. It has always been a huge challenge to pick the right data sample for that universal spread measure through regression. The issue here is that, Statistical Arbitrage is not as much of a Regression problem, as it is a behavioral design problem and it has been understood well but quite poorly implemented.

With all the advancement in Artificial Intelligence and Machine Learning, the next wave of algorithmic trading will have the machines choose both the policy as well as the mechanism. Using advanced concepts such as Deep Reinforcement Learning and Neural Networks, it is possible to build a trading system which has cognitive properties that can discover a long term strategy through training in various stochastic environments.

2 Introduction to Deep Q Network

We consider tasks in which an agent interacts with an environment ϵ, in this case the stock emulator, in a sequence of actions, observations and rewards. At each time-step the agent selects an action a_t from the set of legal actions, $A = \{1, ..., K\}$. The action is passed to the emulator and modifies its internal state. In general ϵ may be stochastic. In addition it receives a reward r_t representing the change in the value we are optimising. Note that feedback about an action may only be received after many thousands of time-steps have elapsed.

We therefore consider sequences of actions and observations, $s_t = x_1, a_1, x_2$, ..., a_{t-1}, x_t, and learn game strategies that depend upon these sequences. All sequences in the emulator are assumed to terminate in a finite number of time-steps. This formalism gives rise to a large but finite Markov decision process (MDP) in which each sequence is a distinct state. As a result, we can apply standard reinforcement learning methods for MDPs, simply by using the complete sequence s_t as the state representation at time t.

The goal of the agent is to interact with the emulator by selecting actions in a way that maximises future rewards. We make the standard assumption that future rewards are discounted by a factor of γ per time-step, and define the future discounted return at time t as $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$, where T is the time-step at which the game terminates. We define the optimal action-value function $Q^*(s, a)$ as the maximum expected return achievable by following any strategy, after seeing some sequence s and then taking some action a, $Q^*(s, a) = max_\pi E[R_t | s_t = s, a_t = a, \pi]$, where π is a policy mapping sequences to actions (or distributions over actions).

The optimal action-value function obeys an important identity known as the Bellman equation. This is based on the following intuition: if the optimal value $Q^*(s', a')$ of the sequence s' at the next time-step was known for all possible actions a' , then the optimal strategy is to select the action a' maximising the expected value of $r + \gamma Q^*(s', a)$,

$$Q^*(s, a) = E[r + \gamma max_{a'} Q^*(s', a') | s, a]$$

The basic idea behind many reinforcement learning algorithms is to estimate the actionvalue function, by using the Bellman equation as an iterative update, $Q_{i+1}(s, a) = E[r + \gamma max_{a'} Q_i(s', a')|s, a]$. Such value iteration algorithms converge to the optimal actionvalue function, $Q_i \to Q^*$ as $i \to \infty$.

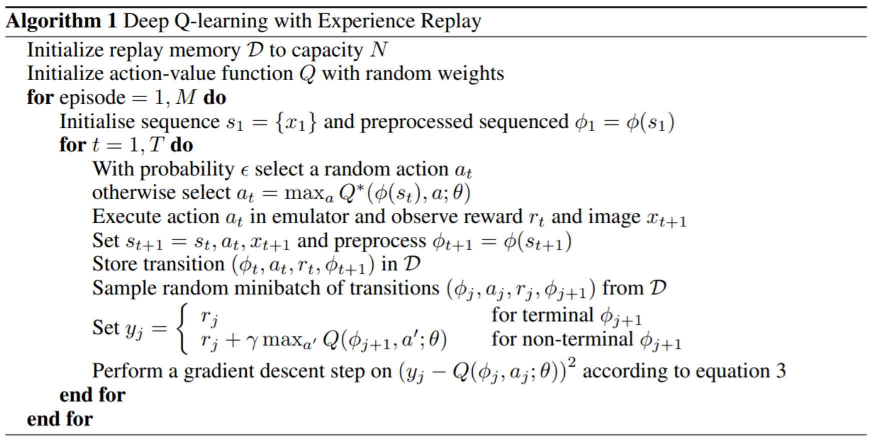
In practice, this basic approach is totally impractical, because the action-value function is estimated separately for each sequence, without any generalisation. Instead, it is common to use a function approximator to estimate the action-value function, $Q(s, a; \theta) \approx Q^*(s, a)$. In the reinforcement learning community this is typically a linear function approximator, but sometimes a non-linear function approximator is used instead, such as a neural network. We refer to a neural network function approximator with weights θ as a Q-network. A Q-network can be trained by minimising a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i,

$$L_i(\theta_i) = E[y_i Q(s, a; \theta_i))^2]$$

where $y_i = E[r + \gamma max_{a'} Q(s', a'; \theta_{i1})|s, a]$ is the target for iteration i and $\rho(s, a)$ is a probability distribution over sequences s and actions a that we refer to as the behaviour distribution. The parameters from the previous iteration θ_{i-1} are held fixed when optimising the loss function $L_i(\theta_i)$. Note that the targets depend on the network weights; this is in contrast with the targets used for supervised learning, which are fixed before learning begins. Differentiating the loss function with respect to the weights we arrive at the following gradient,

$$\nabla_{\theta_i} L_i(\theta_i) = E[(r + \gamma max_{a'} Q(s', a'; \theta_{i1}) Q(s, a; \theta_i)) \nabla \theta_i Q(s, a; \theta_i)]$$

Rather than computing the full expectations in the above gradient, it is often computationally expedient to optimise the loss function by stochastic gradient descent. If the weights are updated after every time-step, and the expectations are replaced by single samples from the behaviour distribution ρ and the emulator ϵ respectively, then we arrive at the familiar Q-learning algorithm.



```
Algorithm 1 Deep Q-learning with Experience Replay
  Initialize replay memory $\mathcal{D}$ to capacity $N$
  Initialize action-value function $Q$ with random weights
  for episode $= 1, M$ do
      Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
      for $t = 1, T$ do
          With probability $\epsilon$ select a random action $a_t$
          otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
          Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
          Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
          Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
          Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
          Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
          Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
      end for
  end for
```

Fig. 1: Pseudocode of deep Q learning

3 Problem Statement

We explore how to find a trading strategy via deep Q networks. We propose a MDP suitable for the financial trading task and solve it using a deep Q network. All our experiments will be based on Bitcoin and Indian Equity Instruments trading data.
We are interested in finding the short term trends rather than long term investments.

4 Problem Formulation

4.1 Input

– n-day window: the model uses n-day windows of closing prices to determine the best action.
– total number of episodes to train the network

4.2 Markov Decision Process

– **Statespace** $\epsilon[0, 1]^n$: Given the historical trading data, and an n-day window, we compute the n consecutive pairwise differences of closing prices, and then apply the sigmoid function.
– **Action-space**: $\{buy, sell, do - nothing\}$
– **Reward**: Profit(transaction cost included) earned every time the agent chooses the sell action, computed by the current price subtracted by the earliest bought price. During other actions, the reward is 0.

4.3 Assumptions

We assume:

– Our impact on the market is negligible
– During live trading, the price at which we buy is the previous day's close price

4.4 Hyperparameters

– **episodes** - a number of iterations we want the agent to play
– **gamma** - aka decay or discount rate, to calculate the future discounted reward.
– **epsilon** - aka exploration rate, this is the probability with which an agent randomly decides its action rather than prediction.
– **epsilon_decay** - the decay of the exploration rate.
– **epsilon_min** - the minimum amount the agent must explore.
– **learning rate** - of the neural network.

5 Model Architecture of Neural Network

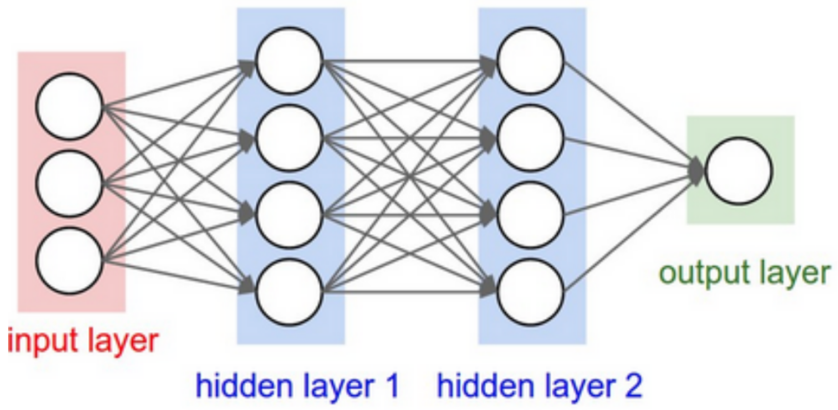


Fig. 2: Example - Architecture of a 4-layer Neural Network

Layer	**Input Dimensions**	**Output Dimensions**	**Activation**
1	n-day window	64	relu
2	64	32	relu
3	32	8	relu
4	8	3	linear

6 Results

After extensive experimentation, the following hyperparameters were used to train the agent :

Hyper-parameter	**Value**
episodes	1000
gamma	0.95
epsilon	1
epsilon_decay	0.995
epsilon_min	0.01
learning_rate	0.001

All the agents are trained using the respective historical post 2010 data with 10-day windows.

Instrument Name	**Backtest Date**	**Total Profit**	**Total Trade**	**Max Liquidity**
BTC	Jan '17 - 18	$38,264.77	463	$270,858
ETH	Jan '17 - 18	$8,338	183	$ 20,098
ITC	Jan '15 -18	INR 4,939	197	INR 11,558

Table 1: Result on unseen data

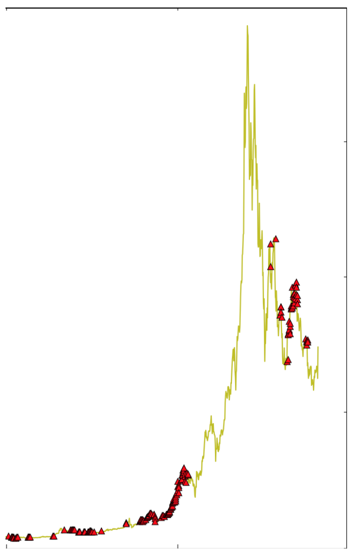


Fig. 3: Bitcoin - Price vs Time graph marked with agent's buy action

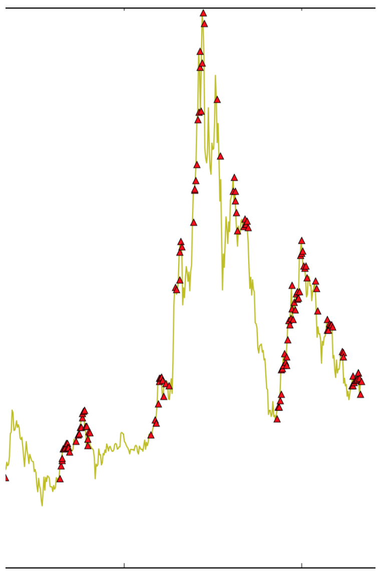


Fig. 4: Ethereum - Price vs Time graph marked with agent's buy action

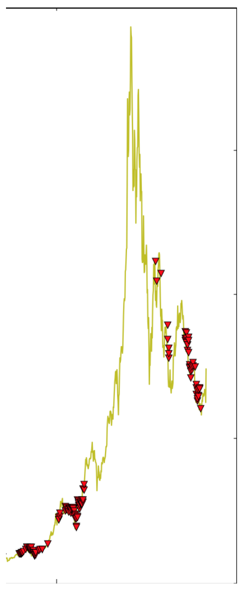


Fig. 5: Bitcoin - Price vs Time graph marked with agent's sell action

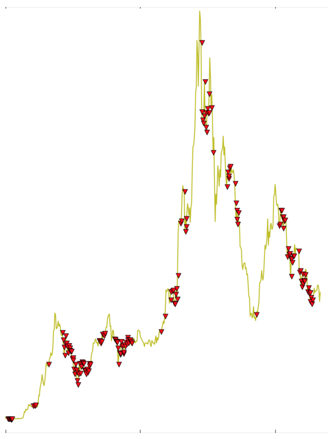


Fig. 6: Ethereum - Price vs Time graph marked with agent's sell action

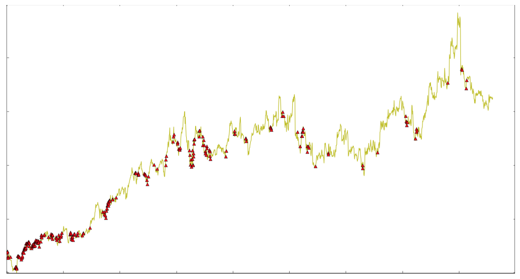


Fig. 7: ITC(NSE) - Price vs Time graph marked with agent's buy action

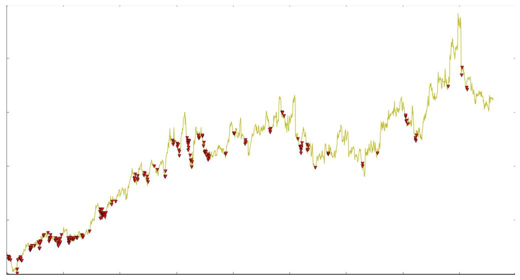


Fig. 8: ITC(NSE) - Price vs Time graph marked with agent's sell action

7 Conclusion

From the above graphs, it is evident that the agent is able to decently learn the local peaks and troughs and makes buy and sell actions based on them. Further studies can address the risk aspects of the trading strategy.

8 References

– Deep Reinforcement Learning: An Overview - Yuxi Li