Stock Trading Agent using DQN

Aditya Narkar
Kansas Sate University
CIS 730
avnarkar@ksu.edu

Abstract

Deep learning applies hierarchical layers of hidden variables to construct nonlinear high dimensional predictors. My goal is to develop and train deep learning architectures for stock market data.

1 Introduction

I applied DQN on stock market data. I preprocessed data available on www.quandl.com, and used that to train our agent. I developed two agents, one with 2 actions - Buy and Sell, and the other one with 3 actions - Buy, Hold and Sell and tried to see which will be stable and perform better.

2 Background and Related work

I followed research paper by Jonah Varon and Anthony Soroka, titled "Stock Trading with Reinforcement Learning". Their method was to use 2 actions only, Buy and Sell. They calculated their reward as percent change between current value and previous value. They tried to implement their DQN agent on the per minute data. I have implemented the same reward policy but I used daily data instead. I wanted to predict the data for longer investment strategy.

3 Problem Specification

I will be applying Deep Q-Learning to trading in a different and perhaps more straightforward manner. Agent rewards will be defined by stock performance. Actions will be limited to Buy and Sell for one agent and the other agent will be having Buy, Sell and Hold actions. States will be represented with Open, High, Low, Close, Volume, High-Low percent change, Open Close Percent Change and 30 day Moving average.

3.1 What is Q-Learning?

Q-learning is a model-free reinforcement learning technique. Specifically, Q-learning can be used to find an optimal action-selection policy for any given (finite) Markov decision process (MDP). It works by learning an action-value function, often denoted by , which ultimately gives the expected utility of taking a given action in a given state , and following an optimal policy thereafter. A policy, often denoted by , is a rule that the agent follows in selecting actions, given the state it is in. When such an action-value function is learned, the optimal policy can be constructed by simply selecting the action with the highest value in each state. One of the strengths of Q-learning is that it is able to compare the expected utility of the available actions without requiring a model of the environment.

3.2 What are Deep Q-Networks?

Neural networks are exceptionally good at coming up with good features for highly structured data. We could represent our Q-function with a neural network, that takes the state and action(s) as input and outputs the corresponding Q-value. This approach has the advantage, that if we want to perform a Q-value update or pick the action with the highest Q-value, we only have to do one forward pass through the network and have all Q-values for all actions available immediately.

3.3 Experience Replay

It turns out that approximation of Q-values using non-linear functions is not very stable. There is a whole bag of tricks that you have to use to actually make it converge. And it takes a long time, almost a week on a single GPU. The most important trick is experience replay. During iterations, all the experiences < s, a, r, s' > are stored in a replay memory. When training the network, random minibatches from the replay memory are used instead of the most recent transition. This breaks the similarity of subsequent training samples, which otherwise might drive the network into a local minimum. Also experience replay makes the training task more similar to usual supervised learning, which simplifies debugging and testing the algorithm.

I used Replay memory of size 20000, and minibatches of size 64. Minibatches should be chosen from the replay memory at random to not overfit the data.

4 Approach

I used the program available online on GitHub which was developed for training game playing agent using DQN. I modified the program to learn the stock trading daily data. I gathered data for stock named "Asian Paints Ltd" from www.quandl.com. The website has APIs available for Python to directly download the data related to particular stock.

I preprocessed data such that mean should be scaled. I implemented following reward policy for both the agents:

$$r = \text{percent return} = (\text{curr price - prev price})/(\text{prev price})$$

$$r(Buy) = r$$

$$r(Sell) = -r$$

State will be represented with Open, High, Low, Close, Volume, High-Low percent change, Open Close Percent Change and 30 day Moving average for agent with 3 actions. For agent with two action I tried with 8 input parameters but it was not quite satisfactory and hence I trained this agent with only High-Low percent change, Open Close Percent Change and 30 day Moving average these 3 as a state.

4.1 Exploration

If we pick an action with the highest Q-value, the action will be random and the agent performs crude "exploration". As a Q-function converges, it returns more consistent Q-values and the amount of exploration decreases. So one could say, that Q-learning incorporates the exploration as part of the algorithm. But this exploration is "greedy", it settles with the first effective strategy it finds.

A simple and effective fix for the above problem is ε-greedy exploration – with probability ε choose a random action, otherwise go with the "greedy" action with the highest Q-value. In their system DeepMind actually decreases ε over time from 1 to 0.1 – in the beginning the system makes completely random moves to explore the state space maximally, and then it settles down to a fixed exploration rate.

I have used standard exploration policy for DQN. I used epsilon with initial value = 1, where it takes 100% random actions. I gradually decreased the value of epsilon such that it reaches 0.1 where agent only takes 10% actions randomly, whereas 90% actions will be taken using the max Q-value of given options. This approach is called epsilon-greedy.

4.2 DQN architecture

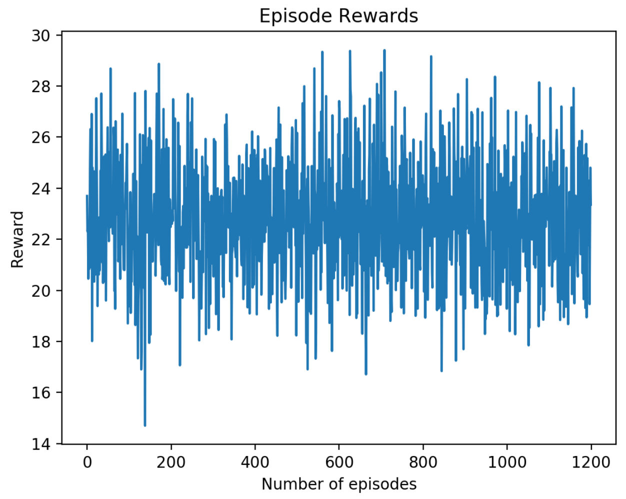
I have used the following architectures while building two different agents. There is not much

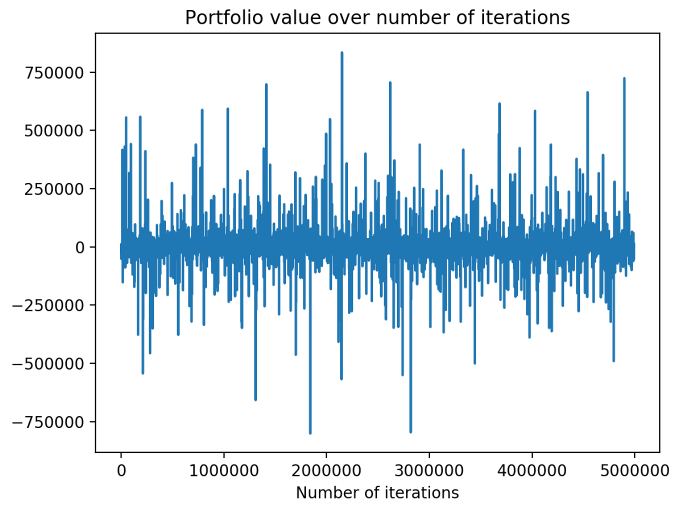
Paramters	Agent with actions = 2	Agent with actions = 3
Input Layer	3	8 (3 + 5 scaled mean)
Hidden layer 1	20	20
Hidden layer 2	20	20
Ouput Layer	2	3
Activation function	RELU	RELU
Implemented using	Tensorflow-PrettyTensor	Tensorflow-PrettyTensor

different between two architectures.

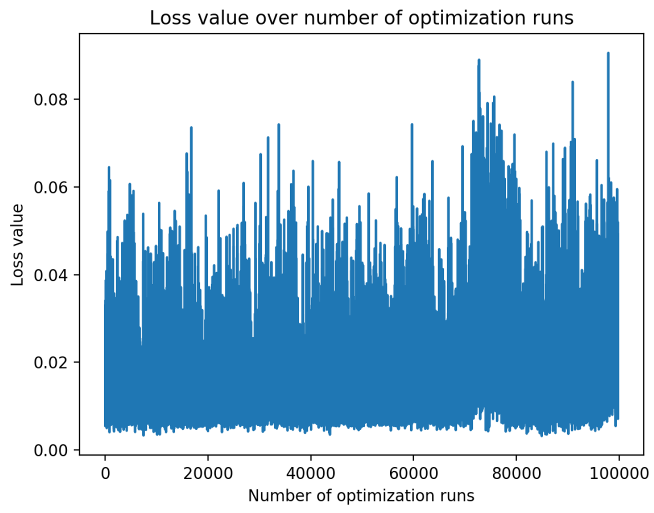
5 Results

I have implemented both of these agents with episode size equal to the size of training data. I have used following graphs to showcase my results. Following results are of the agent with three actions. I have graphed all the results after approximately 6 million iterations (~1200 episodes).



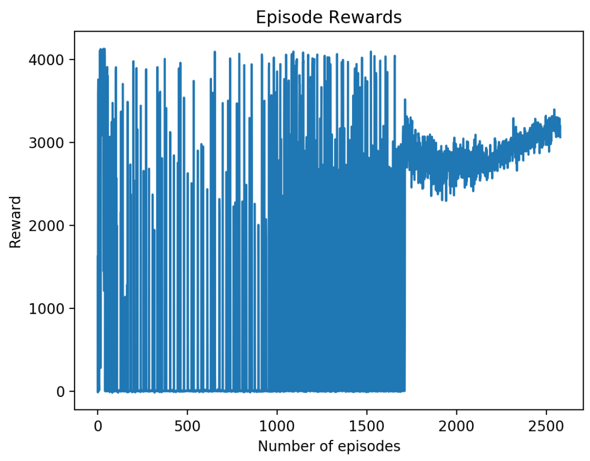


The above graph is for the agent with 3 actions, I have used initial portfolio value as 0. I have plot the portfolio value over every iteration. At the start of each episode I used to reset the portfolio value to 0.

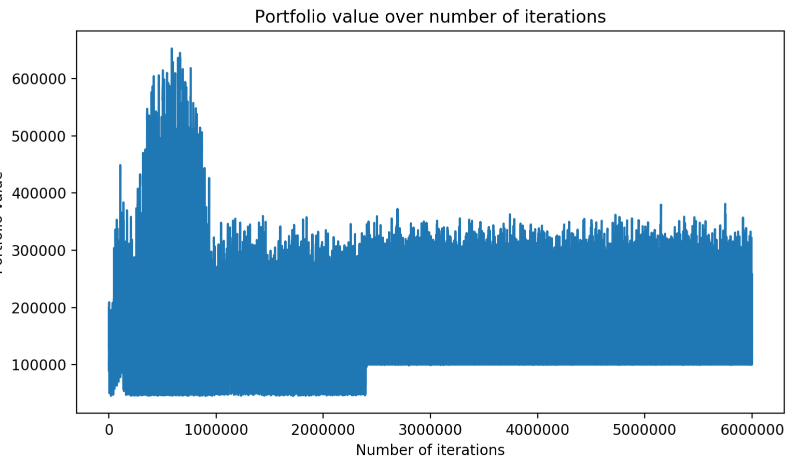


The above graph depicts the oscillation of loss function value over the optimization runs of neural network. I have used mean squared error loss function to calculate loss value.

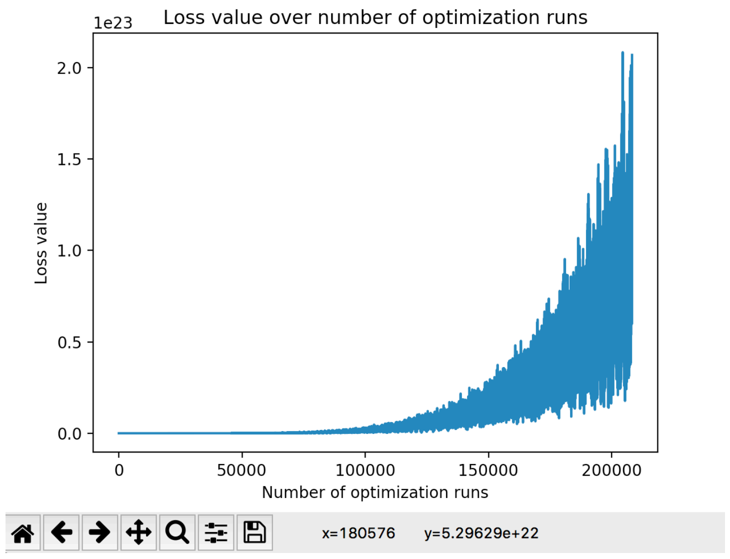
The following graphs will showcase the different scenario I evaluated for the agent with only two actions.



The above graph shows the change in reward value over the episodes. As we can see, reward is not stable over time.



The above graph shows the change in portfolio value over the course of 6 Million iterations. I have used starting portfolio value as 100k.



The above graph shows the increase in loss value of this agent. Loss value starts to increasing after 1 million iteration and reaches positive infinity after 2 million iterations.

6 Conclusion

DQN agent with 3 action was stable in terms of loss, and loss value over time was below 1.0 whereas DQN agent with 2 actions having loss value near to the positive infinity.

Portfolio value change in DQN agent with 2 actions was greater but was not stable, on the other hand, agent with 3 actions produced stable results but unable to impress on the final outcome.

Both the agent were having their final Q-values closer to each other but pretty much biased towards single action only.

7 Future Scope

As conclusion having so many un stable results because of higher loss and biased Q values, I would like to debug the Deep Q Network for Bothe agent to resolve this issue.

I would like to consider different formation of state so that I would be getting efficient results for both the agents.

I want to incorporate sentiment analysis and to see the differences between the result.

For further evaluations, I would also like to conside the incorporation of technical indicators to guide our agent through this process.

8 References

1. Tambet Matiiesen. (2015, Dec 22). *Demystifying Deep Reinforcement Learning.* Retrieved from https://www.intelnervana.com/demystifying-deep-reinforcement-learning/
2. Jonah Varon and Anthony Soroka. (2016, Dec 12). *Stock Trading with Reinforcement Learning.* Retrieved from https://static1.squarespace.com/static/563f45cfe4b0bff8503c3b20/t/58750e10e3df286f96adea20/1484066322582/Varon_Soroka_FinalProject.pdf
3. Q-learning article, retrieved from https://en.wikipedia.org/wiki/Q-learning

8.1 Sources used:

I have used the skeleton provided on following url to build both the agents
Url: https://github.com/Hvass-Labs/TensorFlow-Tutorials/blob/master/16_Reinforcement_Learning.ipynb