

University of Bucharest
Faculty of Mathematics and Computer Science

Graduation Thesis

Reinforcement Learning in Stock Trading

Scientific Coordinator,
Lect.dr. Ciprian Ionuț Păduraru

Graduate Student,
Burdescu Alexandru

I. Contents

II.	Introduction.....	4
1.	Thesis Area of Interest	4
2.	Motivation.....	4
3.	Usage.....	5
4.	Related work	5
1.	Advantage Actor Critic (A2C)	5
2.	Deep Deterministic Policy Gradient (DDPG).....	5
3.	Proximal Policy Optimization (PPO).....	5
III.	Theoretical analysis	7
1.	Used technologies	7
2.	Project overview	8
1.	Config.py file:	9
2.	Replay buffer:.....	10
3.	DQN class:	11
4.	Agent:	11
5.	Trader:	14
6.	Environment:.....	17
7.	DataBase Environment (DbENV):.....	19
IV.	Results.....	22
1.	Apple Stocks	23
2.	Tesla Stocks	25
3.	EURUSD pair	27
4.	Netflix Stocks.....	28
V.	Policy gradient comparison.....	31
1.	The implementation of the Policy Gradient algorithm	31

2.	Results.....	33
1.	Apple Stocks	34
2.	Tesla Stocks.....	35
3.	Algorithm confronting	37
VI.	Conclusion	39
1.	Innovation	39
2.	Further research	40
VII.	References.....	42
VIII.	List of figures	43

II. Introduction

1. Thesis Area of Interest

Artificial Intelligence (AI) is a growing domain of IT with many possibilities of applicability. AI is used in many areas where it performs tasks that traditional programming does not achieve such good results. From spotting and learning patterns impossible for humans to classifying different objects that are easily separable just by looking at them are tasks that can not or can hardly be solved by traditional programming.

Reinforcement learning although, an old branch of AI, just nowadays started growing in popularity with algorithms like AlphaGo and AlphaZero.

The idea of learning by interacting with an environment represents one of the first things you can think of when it comes to the nature of learning [1]. That is what started reinforcement learning and helped it evolve through the years.

2. Motivation

AI has always got my attention as an area of interest, but with the introduction of algorithms such as AlphaZero, AlphaGo, AlphaStar, I was fascinated. Further studies revealed to me the potential that reinforcement learning had in playing games with the help of the DQN algorithm developed by Deepmind [2]. That was when I realized the possibilities of beating every game you can through the implementation of the DQN algorithm. I was also passionate about the stock market that many consider being "a game of luck" and asked myself, can I beat this game? It would also come with the reward of obtaining a passive income that would not be bad from a financial perspective. I also wanted to work with reinforcement learning algorithms as they seem to approach humans more than people think. As I previously said, both learn most naturally, by interacting with an environment. The work I have done for this paper, also made me think of strong AI and the possibilities of it passing the imitation game as described by Alan Turing [3].

3. Usage

The usage of the algorithm is of it creating passive income for the owner. A vast spread of it could not be beneficial for the stock market, as a lot of trades would do the same type of trading resulting in market manipulation which is also illegal. It also has an element of risk, meaning that the algorithm does not take into account the news that can affect the market. This algorithm is made for study purposes only and not for public use.

4. Related work

1. Advantage Actor Critic (A2C)

A2C is an actor-critic algorithm working with multiple agents for a better update of the policy gradients. It also does calculate an advantage function instead of a value function, meaning when it takes an action does not only think about how good it is, but how good it can be. This implementation results in a lower variance of the policy network.

A2C uses copies of the same agent in the same environment but having as input different data samples. After all the agents finish calculating their gradient, an average is made and sent to the global network, which updates both the critic and the actor network.

2. Deep Deterministic Policy Gradient (DDPG)

DDPG is an algorithm made to encourage maximum return for your investment. It combines two well-known reinforcement learning techniques: policy gradient and Q-learning. In contrast to other approaches, it does not learn from Q-tables, which is not considered to be a direct learning method. DDPG considers the stock market a continuous action space and learns directly from the environment itself through policy gradient.

3. Proximal Policy Optimization (PPO)

PPO is used to control the policy gradient update. It assures that the new policy is not so different from the last one. PPO discourages large policy changes and improves the stability of the policy network training.

All the above algorithms were implemented and tested in paper [4]. They were run individually on a Google stock database between 2016/01/04-2020/05/08. The main goal of

the paper was to create a trading strategy using the best-performing algorithm in a given period of time based on their previous performance. Then we are given the final results for that period.

2016/01/04-2020/05/08	PPO	A2C	DDPG
Cumulative Return	83.0%	60.0%	54.8%
Annual Return	15.0%	11.4%	10.5%
Max Drawdown	-23.7%	-10.2%	-14.8%

As you can see above, reinforcement learning in stock trading is a real problem that can be solved and profit-producing algorithms can be produced.

III. Theoretical analysis

1. Used technologies

Python 3.7 Python was invented by Guido van Rossum as an easy-to-learn and powerful programming language. As a high-level programming language, it has implemented high-level data structures and types, flexible arrays, and dictionaries, that in a traditional programming language would take a lot to implement. It also comes with a lot of standard libraries and built-in modules. Python is an interpreted language that also saves a lot of time when it comes to compilation and linking [5].

Xampp – Xampp is an open-source, free project that aims the easy creation of an Apache local server. It allows you to create a database on a local server with much ease.

Navicat – Released in 2002, Navicat is a database management software that supports multiple connections for local and remote databases. It comes with a lot of features and an easy-to-use interface that helps you move much faster with your projects.

Pycharm – Released in 2010, Pycharm is an IDE (integrated development environment) developed specifically for the Python programming language. It provides a lot of features such as graphic debugging, code analysis, refactoring, and many more which helps a lot in developing your projects.

Pip – Pip is a package manager for Python used to manage software packages.

MySQL- Mysql is an open-source database system. It was released in 1995 and it is written in C / C++. It is a good choice when it comes to working with relational databases because of its speed and the help you can get from the community.

Numpy- Numpy is a package for the Python programming language that adds support for multi-dimensional arrays and matrices. It also comes with a lot of mathematical functions and tools to easily operate on these large data structures. Numpy not only helps you with all the above-mentioned but it will boost the speed of your Python program because of its C-memory allocation.

Tensorflow- Tensorflow was released in 2015 as a free open-source machine learning library. In my opinion, Tensorflow is the best when it comes to ease of use and fast implementation of your ideas. It is very intuitive and it can keep up with your "flow" of thoughts.

2. Project overview

My project consists of a Reinforcement Learning algorithm learning to trade stocks. First, I would like to introduce you to my thought process.

General implementation of the stock trading reinforcement learning algorithm consists of it having to take three actions based on their observation. That actions are buying, holding, and selling. I find this method rather inefficient. The hold actions seem to be irrelevant as long as you can think of the tools that are offered to you when you start trading. You can always do a buy or short-sell action that will be automatically on hold as long as they do not reach a target price you set. I took advantage of this fact and introduced my bot to "limit orders". I considered it also to be an easier way to learn for my algorithm. When it wants to buy something it means only and only one thing: that the algorithm thinks the price of that stock will rise. For now, all seem to be very natural even from a human way of thinking. Now with the introduction of limit orders, you will set two additional orders alongside the action chosen by the AI. You set two target prices based on the percent of the actual price. One is when you will take your profit (i.e. you did a buy action, the price has risen over your target price to take profit and will automatically close your order and give you your profit) and the other is when you will stop your losses (i.e. you did a buy action, the price has fallen lower than your target price to stop your losses and will automatically close your order and give you your remaining money).

Concluding, my algorithm will know only 2 actions:

buy - representing the AI thinking of the price going up

short-sell - representing the AI thinking of the price going down

The bets are accordingly adjusted for the given action and carefully managed by the provided environments.

My project has many parts and I will describe them accordingly:

1. Config.py file:

This file contains easy-to-setup variables for the program and for different tests you may want to make. The following variables are:

`ALPACA_API_BASE_URL`

This variable represents the URL for the live trading environment. You can switch it from a real-money account to a "paper" account purposed for testing and interacting with fake money.

`ALPACA_API_KEY_ID`

Represents the public key provided by the Alpaca trading site and it is used to utilize the API tools offered.

`ALPACA_API_SECRET_KEY`

This variable represents the account secret key which is used to sign transactions through the API.

`ACCOUNT_URL`

This variable represents your account URL.

`STATE_DIM`

This is an important variable for training your network. It represents the length of the input string of prices that goes into the main neural network. The bigger, the more information is given for the algorithm to learn, the slower the program works.

Symbol

This variable is a string and represents the symbol of the stock you want to trade on the market.

Quantity

This variable represents the number of stocks you want to trade in a single order. It can maximize profit based on your capital. For example, if you make a profit of 1\$ on a single stock, trading it twice at that time would have made 2\$ profit.

Take_profit_percent

This variable is given as a percentage and represents how much the shift in price should be for you to take your profit in a given trade. It should be assigned only greater than 1 (i.e. if it is 1,1 represents that the price should be 110% of the original price to stop the order if it is a buy order, and 90% if it is a short-sell order)

Stol_loss_percent

This variable is given as a percentage and represents how much the shift in price should be for you to stop your losses in a given trade. It should be assigned only lower than 1 (i.e. if it is 0,9 represents that the price should be 90% of the original price to stop the order if it is a buy order, and 110% if it is a short-sell order)

2. **Replay buffer:**

Probably the least difficult thing to implement in the project. It consists of a double-ended queue (Deque) that stores all the details about a taken action: the states when it was taken, the action, the reward, the next state reached, and if it was the state of one episode. An episode is considered to be a given number of consecutive trades. The replay buffer was class-defined with only one parameter: "buffer_size" which will give the length of the double-ended queue.

It has two member variables defined in the constructor:

```
self.buffer_size = buffer_size
```

This variable defines the size of the double ended queue.

```
self.replayMem = collections.deque(maxlen=buffer_size)
```

This variable represents the double ended queue.

It implements two methods:

```
def add(self, state, action, reward, next_state, done):
```

This method adds the details to the double-ended queue.

```
def sample(self, batch_size):
```

This method samples a random batch-sized length from the replayMem.

3. **DQN class:**

The DQN class contains the neural network architecture we will use for this project. It contains one input, one output layer, and a hidden layer, all being fully connected. The first two layers got a "relu" activation, while the output layer has a softmax activation. Also, the class has overloaded the TensorFlow function "call" for better code management and an easier way to use the neural network.

4. **Agent:**

The Agent class represents the way we communicate with the environment. The agent is the main focus of this program as it will be the one who we will train and the one who will do the trades for us. This class is the implementation of the DQN algorithm with its specific variables and its specific methods.

The class has the following variables:

`main_nn` - It is an instance of the DQN class, representing the main neural network.

`target_nn` - It is an instance of the DQN class, representing the target neural network that performs the learning.

`buffer` - It is an instance of the ReplayBuffer class, which stores all the previous results of our actions.

`batch_size` - It is a customizable variable that represents the length of the buffer needed to start training.

`epsilon` - It is the variable that helps us selecting an "e-greedy policy". It starts as 1 representing the probability of taking a random action and will decay with the progress of training.

`epsilon_decay` - Starting at 0.99 it represents the decay rate of the epsilon factor while training.

`steps_until_sync` - This variable is customizable and represents the number of steps we update the target network weights with the main network weights.

`optimizer` - It represents the optimizer we use for our neural network. In this paper was chosen to be Adam with a learning rate of 0.0001 as being one of the best performing and commonly used. It also can be changed to other known optimizers that TensorFlow offers.

`loss_function` - It represents the loss function used to determine our training loss. In this paper, I used binary cross-entropy as it performs the best when it comes to classifying two actions.

`env`- Represents one of the training environments. It is an online environment build through APIs on the Alpaca website. It gives the possibility of training on a "paper" account for live-action and also it lets you switch to a real account if needed.

`dbenv`- It represents another custom environment that runs on a local database of 3 months-second saved stock prices.

The class implements the following methods:

```
def select_epsilon_greedy_action(self, state, epsilon):
```

This method selects at random for a given state, based on the probability of epsilon an action. If it is chosen to be random, the environment will return at random one of the two implemented actions with a probability of 50/50. If it is not chosen at random, the state is given to the main neural network (main_nn) and action is chosen by the algorithm (which one is considered the best by the algorithm).

```
def train_step(self, states, actions, rewards, next_states, done):
```

This method represents the training of the networks. It has as input a random sample of batch_size variable size from the buffer. The two networks visit the batch-sized sample and try to predict the best actions for the given states. Here we try to predict the best reward for the given states and try to get it as close as possible to the target reward. Further, we calculate the loss function based on our predictions and targets, we calculate the gradients and apply our optimizer. During the train_step method, we lower the value of our epsilon as we start to learn and think of better actions than random ones.

```
def saveModel(self, modelName):
```

This method saved the model at the time of its call. We do not want to lose a well-trained agent. It accepts as a parameter the model name, representing the name of the file saved.

```
def loadModel(self, modelName):
```

This method loads a given model specified by the parameter modelName.

```
def train(self):
```

This method represents the training iteration on the online environment. It is perceived as a continuous loop as the stock market is a continuous moving thing so we have to be up to date with the moves that the market makes and the situations it encounters. Here we give the initial state for the market which represents the last STATE_DIM (configurable parameter) prices of the market taken by seconds. The target network weights are here also updated after several given numbers of trades and the model is saved. After an action is taken by the select_epsilon_greedy_action method, the state is saved into the buffer. When the buffer reached the length of the batch_size variable training is started.

```
def train(self):
```

This method represents the training iteration on the local database environment. It is perceived as a continuous loop as the stock market is a continuous moving thing so we have to be up to date with the moves that the market makes and the situations it encounters. This method acts the same as the above-mentioned one but it moves much faster and gives us faster results as it does not represents live trading. It speeds through the prices database and gives us feedback on how well the algorithm performs.

5. **Trader:**

The trader class incorporates the Alpaca market API through the package alpaca_trade_api. This class provides utility for our live trading environment and incorporates the tools needed for it to work.

The trader class contains the following variables:

key_id - represents the Alpaca trade API public key given in the config file

secret_key - represents the Alpaca trade APO secret key given in the config file

base_url - represents the base URL where the trades are made, also given in the config file

`data_url` - represents the URL to the endpoint where market data are stored. It provides utility to get history prices, actual prices, and more.

`symbol` - represents the config variable `symbol`, that tells us what stock we are trading.

`base_bet` - represents the quantity we are trading. It is the quantity variable from the config file.

`API` - represents the connection to the endpoint needed to make the trades.

`account_info` - represents the response from `self.api.get_account()` that gives details of the account.

`equity` - represents the money we have in the account and it is given by the `account_info.equity` field.

`margin_multiplier` - represents the money available for margin trading and it is given by the `account_info.multiplier` field.

The class implements the following methods:

```
def market_status(self):
```

This method gets the clock from the API endpoint and checks if the market is open so we can move forward with trading if the market is closed

```
def check_if_tradable(self):
```

This method checks if the given symbol is tradable on the Alpaca market.

```
def get_history_data(self, number):
```

This method returns a given number of last prices for the given symbol in the config file. It is used to give us an initial state so we can start trading and is called with the parameter `STATE_DIM`.

```
def buy(self, quantity, symbol_price):
```

This method places a buy order at a given price and for a given quantity. The symbol price is the last price given by the `get_history_data` method so it stays accurate from an algorithmic perspective. It usually gets the price instantly as the algorithm is moving very fast. This method also places the two bracket orders, one for the stop loss and one for the take profit.

```
def buy_bracket_and_return_sold(self, quantity, symbol_price):
```

This method calls the above-mentioned buy method. After the buy method executes, this method waits for one of the two bracket orders to get executed. If the take profit price is reached, we take our profit and return a reward of 1 as we made a profit in this trade. Else if the stop-loss price is reached, we return a reward of -1. This also takes into account orders that for some reason were not executed, and for them return a reward of 0.

```
def sell(self, quantity, symbol_price):
```

Similar to the buy method, this represents the short-sell order. It uses the same parameters and does the same thing, except that instead of buying we are shorting the given position.

```
def sell_bracket_and_return_sold(self, quantity, symbol_price):
```

This method calls the above-mentioned sell method. It works the same as the `buy_bracket_and_return_sold`.

6. Environment:

The environment class is the one that links our agent and the Alpaca market. This is the one with who our agent interacts and gives him all the information and responses needed for training.

It contains the following variables:

trader - This is an instance of the Trader class, that is used to interact with the Alpaca market

action_space - This represents the action space of the environment. It is a list of two numbers each representing an action: 0 is the buy action and 1 is the short sell action. It helps us in choosing a random action from the action space when the epsilon change kicks in. It also lets us easily add new actions to space and implement new methods without affecting the already existing code.

steps_till_done - represents a variable that increments every trade. It tells us when an episode is over when it reaches a certain number of trades.

steps_todo_done - represents the threshold of the steps_till_done variable and tells us after how many trades we should end an episode.

The environment class implements the following methods:

```
def random_action(self):
```

This returns a random action from the action_space variable. This helps the implementation of the e-greedy policy.

```
def get_state(self):
```

This method returns the last STATE_DIM prices and represents the current state or the initial state. It works with the help of the get_history_data method from the Trader class.

```
def step(self, action):
```

This method is the actual step in the environment. Given an action, it performs a trade and increments the `steps_till_done` variable to tell us when the episode is over. Based on the action, it takes the output of the `buy_bracket_and_return_sold` or `sell_bracket_and_return_sold` method from the `Trader` class and sets it as a reward. It also gives us the new state based on the new history data after the current trade is over.

The environment also has an interactive design on the Alpaca site. It shows us a graphic of the balance and the current open orders. It also has a menu for the closed orders that were done in the past.

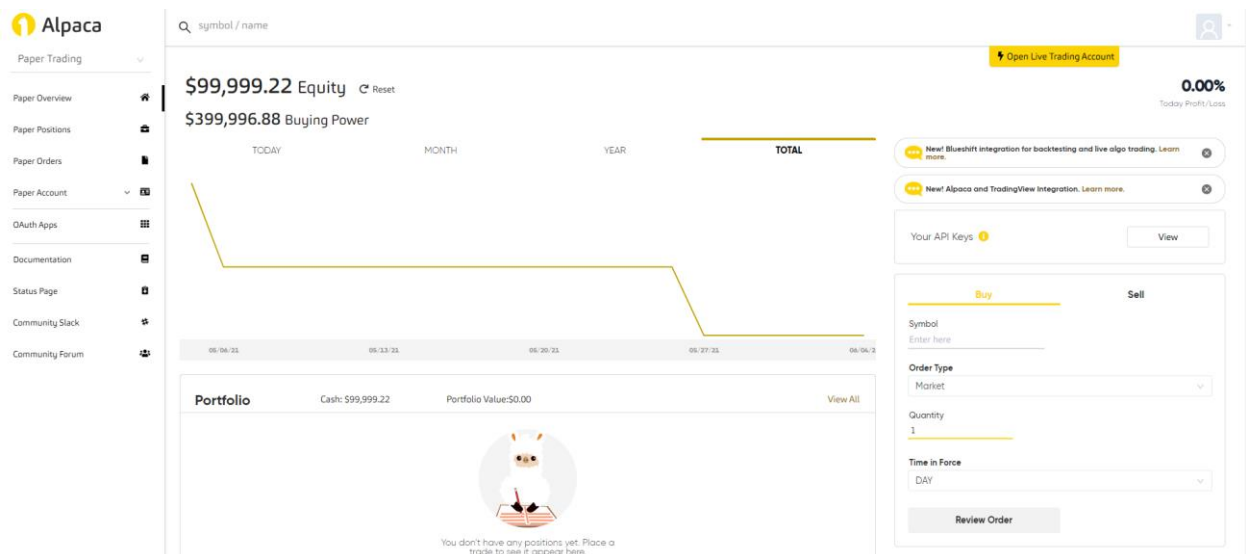


Figure 1 – The Alpaca Trade site overview of the account

The screenshot shows the Alpaca website's 'Orders' menu. On the left is a sidebar with navigation links: Paper Trading, Paper Overview, Paper Positions, Paper Orders, Paper Account, OAuth Apps, Documentation, Status Page, Community Slack, and Community Forum. The main content area is titled 'Orders' and has a search bar. Below the title is a dropdown for 'Closed Orders'. The table below lists 10 orders for AAPL, showing details like quantity, side, type, limit price, stop price, filled average price, notional amount, status, and timestamps for submission, filling, cancellation, and replacement.

Symbol	Qty	Filled Qty	Side	Type	Time in Force	Limit Price	Stop Price	Filled Avg Price	Notional	Amount	Status	Submitted at	Filled at	Cancelled at	Replaced at
AAPL	1	1	sell	stop	day	-	\$126.99	\$126.99	-	\$126.99	filled	2021-05-27 18:05:00	2021-05-27 18:05:00		
AAPL	1	0	sell	limit	day	\$127.25	-	-	-	\$0.00	replaced	2021-05-27 17:52:01			2021-05-27 18:05:00
AAPL	1	1	buy	market	day	-	-	\$127.12	-	\$127.12	filled	2021-05-27 17:52:01	2021-05-27 17:52:01		
AAPL	1	1	sell	stop	day	-	\$127.11	\$127.11	-	\$127.11	filled	2021-05-27 17:52:00	2021-05-27 17:52:00		
AAPL	1	0	sell	limit	day	\$127.37	-	-	-	\$0.00	replaced	2021-05-27 17:51:02			2021-05-27 17:52:00
AAPL	1	1	buy	market	day	-	-	\$127.22	-	\$127.22	filled	2021-05-27 17:51:01	2021-05-27 17:51:01		
AAPL	1	1	buy	limit	day	\$127.26	-	\$127.24	-	\$127.24	filled	2021-05-27 17:50:36	2021-05-27 17:50:42		
AAPL	1	1	sell	market	day	-	-	\$127.32	-	\$127.32	filled	2021-05-27 17:50:35	2021-05-27 17:50:36		
AAPL	1	0	buy	stop	day	-	\$127.52	-	-	\$0.00	canceled	2021-05-27 17:50:35		2021-05-27 17:50:42	
AAPL	1	1	buy	limit	day	\$127.27	-	\$127.27	-	\$127.27	filled	2021-05-27 17:41:30	2021-05-27 17:50:34		

Figure 2 – The orders menu on the Alpaca Trade site.

In the above figure, you can see the buy orders and also the bracket orders for the take profit and stop loss.

7. DataBase Environment (DbENV):

For this local environment to work I had to start a local server with Xampp and make a local database. This environment contains three months of stock prices saved around May 2020.

For this environment class to work, I started by defining the connection to the database. After the connection is established, a cursor is created to fetch through the given symbol prices for better memory management.

The class contains the following variables:

states - is a list of prices, capped at STATE_DIM dimension that updates the last prices.

steps_till_done - how many steps are currently done for a given episode

steps_todo_done - at how many steps an episode ends

start_money - the amount of money we give the algorithm to start within this environment.

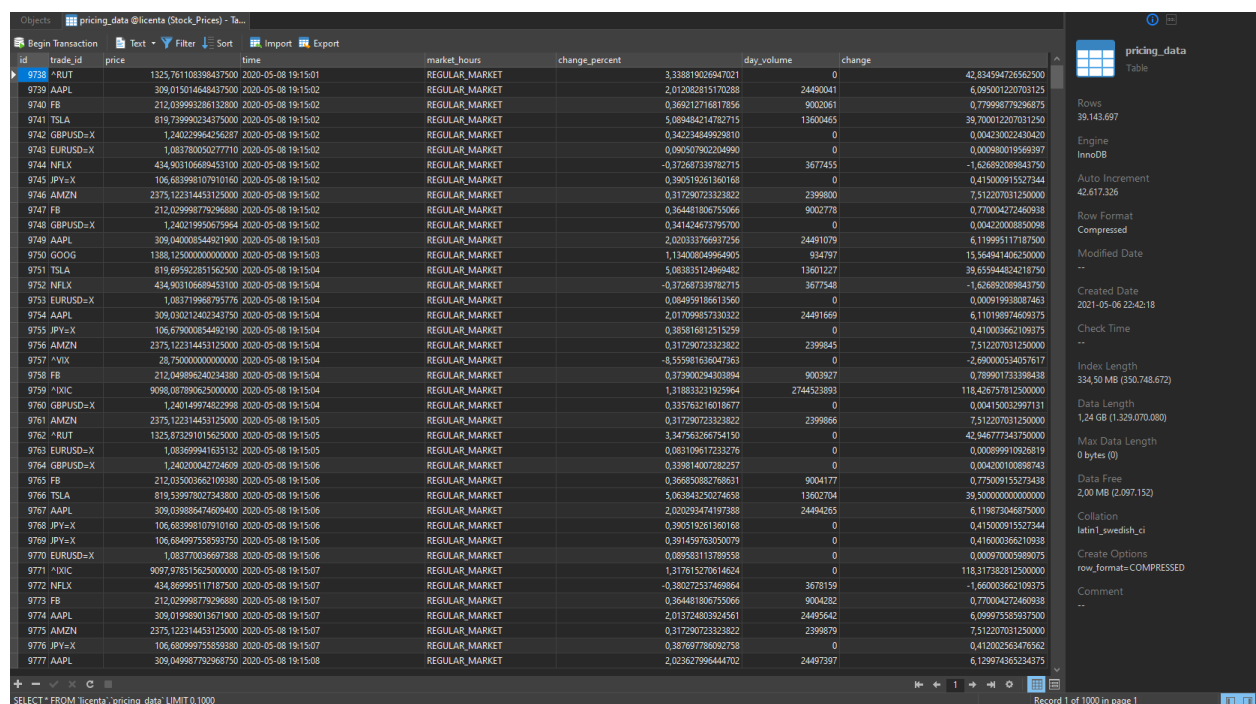
The class contains the following methods:

def state(self):

This method returns the current state. It can be the first state at the start, or after a trade is done a new price is appended to the list for the algorithm to continue where it left.

def step(self, action):

For a given action, this method fetches through the database with the help of the cursor. It simulates the bracket orders for the take profit or stop loss and waits for one of them to complete. This method will return the next state, which will be an updated version of the states list variable, the reward based on which bracket order completed and if this is the final trade of an episode.



id	trade_id	price	time	market_hours	change_percent	day_volume	change
9736	*RUT	1325.761108398437500	2020-05-08 19:15:01	REGULAR_MARKET	3.338819026947021	0	42.834594726562500
9739	AAPL	309.015014648437500	2020-05-08 19:15:02	REGULAR_MARKET	2.012082815170288	24490041	6.095001220703125
9740	FB	212.039993286132800	2020-05-08 19:15:02	REGULAR_MARKET	0.369212716817856	9002061	0.779998779296875
9741	TSLA	819.739993286132800	2020-05-08 19:15:02	REGULAR_MARKET	5.089484214782715	13600465	39.700012207031250
9742	GBPUSD=X	1.240229964256287	2020-05-08 19:15:02	REGULAR_MARKET	0.342234849929810	0	0.004230022430420
9743	EURUSD=X	1.083780050277710	2020-05-08 19:15:02	REGULAR_MARKET	0.090507902204990	0	0.000980019569397
9744	NFLX	434.903106689453100	2020-05-08 19:15:02	REGULAR_MARKET	-0.37268739782715	3677455	-1.626892089843750
9745	JPY=X	106.683998107910160	2020-05-08 19:15:02	REGULAR_MARKET	0.390519261360168	0	0.415000915527344
9746	AMZN	2375.122314453125000	2020-05-08 19:15:02	REGULAR_MARKET	0.317290723323822	2399800	7.512207031250000
9747	FB	212.029998779296880	2020-05-08 19:15:02	REGULAR_MARKET	0.364481806755066	9002778	0.77004272460938
9748	GBPUSD=X	1.24016950675964	2020-05-08 19:15:02	REGULAR_MARKET	0.341424673795700	0	0.004230022430420
9749	AAPL	309.040085448219000	2020-05-08 19:15:03	REGULAR_MARKET	2.02033766937256	24491079	6.119999117167500
9750	GOOG	1388.123000000000000	2020-05-08 19:15:03	REGULAR_MARKET	1.134008049956905	934797	15.56494146620000
9751	TSLA	819.695225152625000	2020-05-08 19:15:04	REGULAR_MARKET	5.083835124968482	13601227	39.6559448216750
9752	NFLX	434.903106689453100	2020-05-08 19:15:04	REGULAR_MARKET	-0.37268739782715	3677543	-1.626892089843750
9753	EURUSD=X	1.083771986795776	2020-05-08 19:15:04	REGULAR_MARKET	0.084959186613560	0	0.000919938087463
9754	AAPL	309.030212402343750	2020-05-08 19:15:04	REGULAR_MARKET	2.017099857330322	24491669	6.110198974609375
9755	JPY=X	106.679000854492190	2020-05-08 19:15:04	REGULAR_MARKET	0.385816812515259	0	0.410003662109375
9756	AMZN	2375.122314453125000	2020-05-08 19:15:04	REGULAR_MARKET	0.317290723323822	2399845	7.512207031250000
9757	*VIX	28.750000000000000	2020-05-08 19:15:04	REGULAR_MARKET	-8.555981636047363	0	-2.69000534057617
9758	FB	212.040986240334380	2020-05-08 19:15:04	REGULAR_MARKET	0.373900294303894	9003927	0.789901733398438
9759	*IXIC	9098.087806250000000	2020-05-08 19:15:04	REGULAR_MARKET	1.318833231925964	2744523893	118.426757812500000
9760	GBPUSD=X	1.240149974822998	2020-05-08 19:15:04	REGULAR_MARKET	0.335763216018677	0	0.004150032997131
9761	AMZN	2375.122314453125000	2020-05-08 19:15:05	REGULAR_MARKET	0.317290723323822	2399866	7.512207031250000
9762	*RUT	1325.873291015625000	2020-05-08 19:15:05	REGULAR_MARKET	3.347563266754150	0	42.946777343750000
9763	EURUSD=X	1.083699941635132	2020-05-08 19:15:05	REGULAR_MARKET	0.083109617233276	0	0.000899910926819
9764	GBPUSD=X	1.24020042724609	2020-05-08 19:15:06	REGULAR_MARKET	0.339814007282527	0	0.004230010898743
9765	FB	212.025003652109380	2020-05-08 19:15:06	REGULAR_MARKET	0.366860882786631	9004177	0.77500155273438
9766	TSLA	819.539978027438800	2020-05-08 19:15:06	REGULAR_MARKET	5.063843230274638	13602704	39.500000000000000
9767	AAPL	309.039886474609400	2020-05-08 19:15:06	REGULAR_MARKET	2.020239474197388	24494265	6.119873046875000
9768	JPY=X	106.683998107910160	2020-05-08 19:15:06	REGULAR_MARKET	0.390519261360168	0	0.415000915527344
9769	JPY=X	106.684997585937500	2020-05-08 19:15:06	REGULAR_MARKET	0.391459763050079	0	0.416000366210938
9770	EURUSD=X	1.083770036697388	2020-05-08 19:15:06	REGULAR_MARKET	0.089583113789538	0	0.000970005989075
9771	*IXIC	9097.978515625000000	2020-05-08 19:15:07	REGULAR_MARKET	1.317615270614624	0	118.317382812500000
9772	NFLX	434.869995117187500	2020-05-08 19:15:07	REGULAR_MARKET	-0.380272537468864	3678159	-1.660003662109375
9773	FB	212.029998779296880	2020-05-08 19:15:07	REGULAR_MARKET	0.364481806755066	9004282	0.77004272460938
9774	AAPL	309.019890136719000	2020-05-08 19:15:07	REGULAR_MARKET	2.01374803924561	24495642	6.099975559375000
9775	AMZN	2375.122314453125000	2020-05-08 19:15:07	REGULAR_MARKET	0.317290723323822	2399879	7.512207031250000
9776	JPY=X	106.68099755859380	2020-05-08 19:15:07	REGULAR_MARKET	0.387687786092758	0	0.412002563476562
9777	AAPL	309.049987929687500	2020-05-08 19:15:08	REGULAR_MARKET	2.023627995444702	24497397	6.12947635234375

Figure 3 – Snapshot of the database from the Navicat database manager

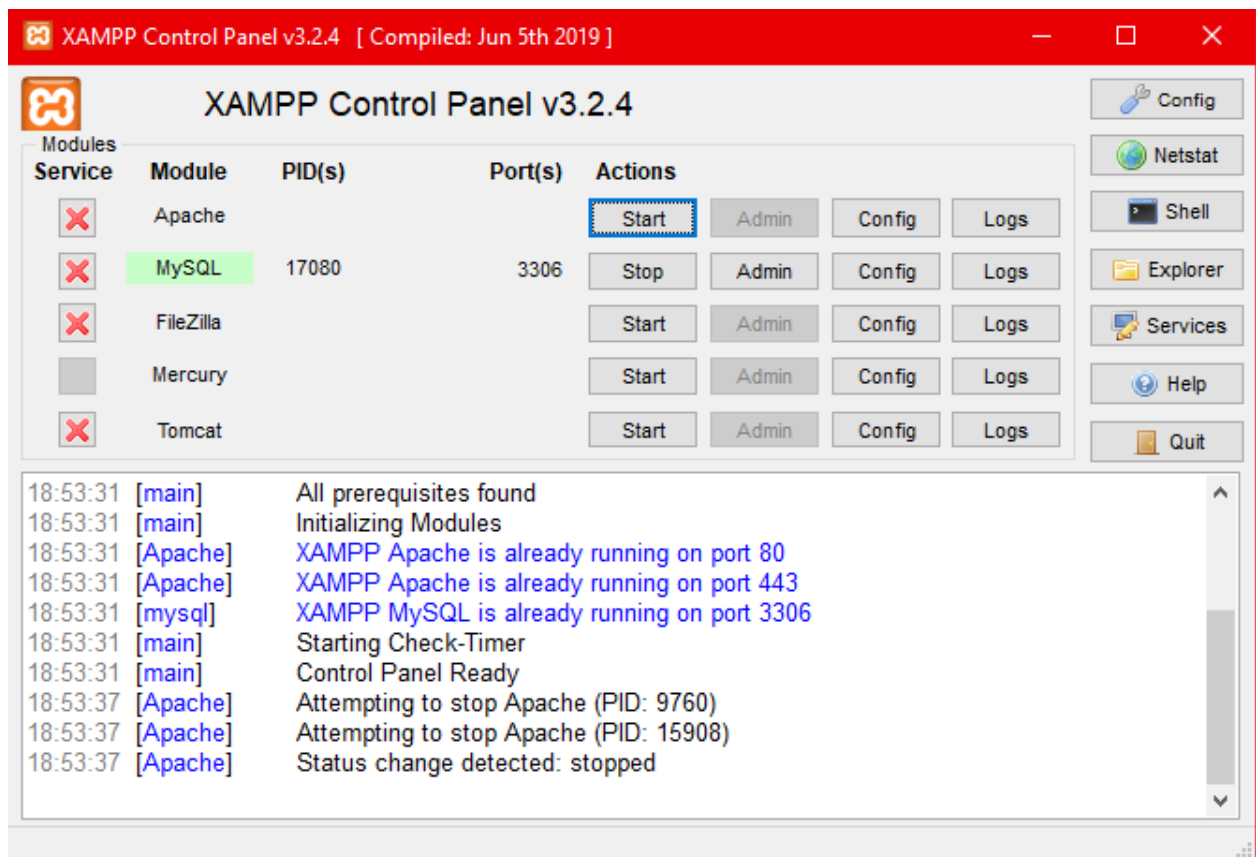


Figure 4 – Snapshot of the XAMPP Control Panel

IV. Results

The results given are from the database environment over three months of saved data. Due to the large amount of time that the algorithm needs to train and the management of the market hours, results on live trading would be nearly impossible. Results on the database can be rapidly obtained and do not need the algorithm to run continuously for a large amount of time.

With the help of a custom-created plot class named `Plotter`, I integrated it into the Database environment to save the evolution of the money and the stock prices for better analysis and a better view of what is happening behind.

Due to the low volatility of the market, all the tests were run with the two bracket orders set as follows:

```
take_profit_percent = 1.001
```

```
stop_loss_percent = 0.999
```

Also, the quantity for the trades was set to 1, as it does not affect the way the algorithm performs, and only the amount of money it gets.

1. Apple Stocks

For a batch_size of 1000 and a STATE_DIM of 60:

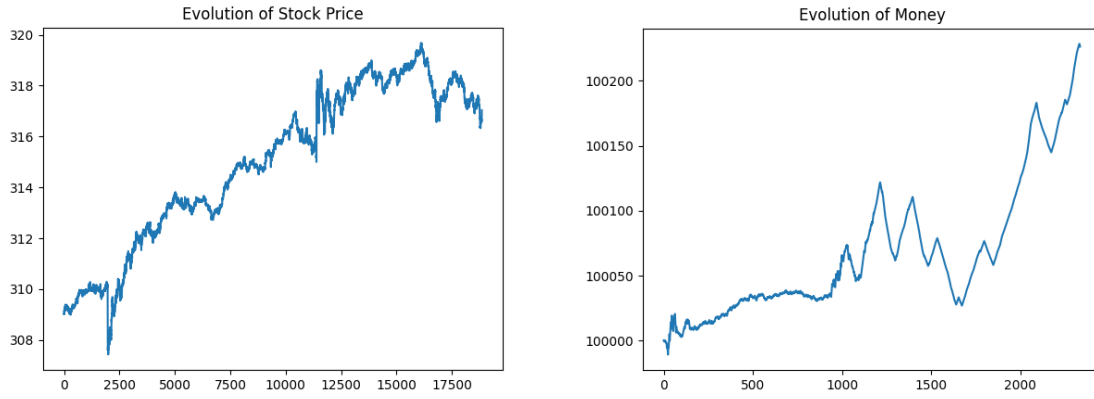


Figure 5 – Evolution of stock price for Apple Stock and Evolution of Money for a batch_size of 1000 and a STATE_DIM of 60

For a batch_size of 1000 and a STATE_DIM of 80:

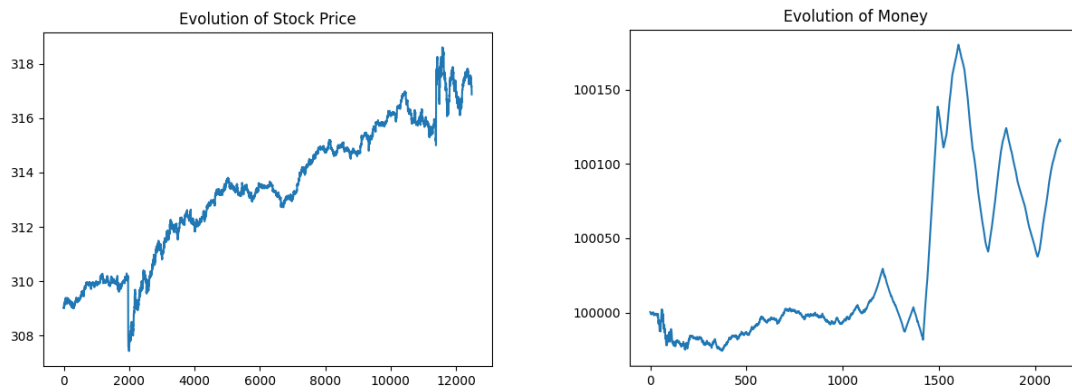


Figure 6 – Evolution of stock price for Apple Stock and Evolution of Money for a batch_size of 1000 and a STATE_DIM of 80

For a batch_size of 100 and a STATE_DIM of 60:

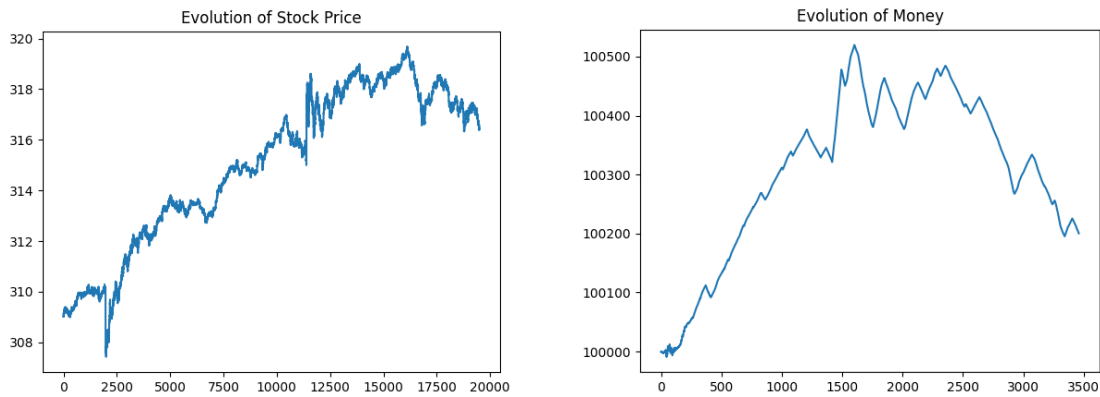


Figure 7 – Evolution of stock price for Apple Stock and Evolution of Money for a batch_size of 100 and a STATE_DIM of 60

We can see that the least we look in the past (represented by the STATE_DIM parameter) and the least we perform random actions (represented by the batch_size parameter) we achieve greater the results on the Apple stock. The result of not taking random actions so much has a clear benefit, and the least we look at the past prices affect the algorithm in such a way that he can predict price trends with much ease. Although it looks promising, further tests are required to see the performance of other stocks.

2. Tesla Stocks

Known for its high volatility the Tesla stock is the next target for testing. The great volatility and the high price of the stock can result in only making more or losing more money.

For a batch_size of 1000 and a STATE_DIM of 60:

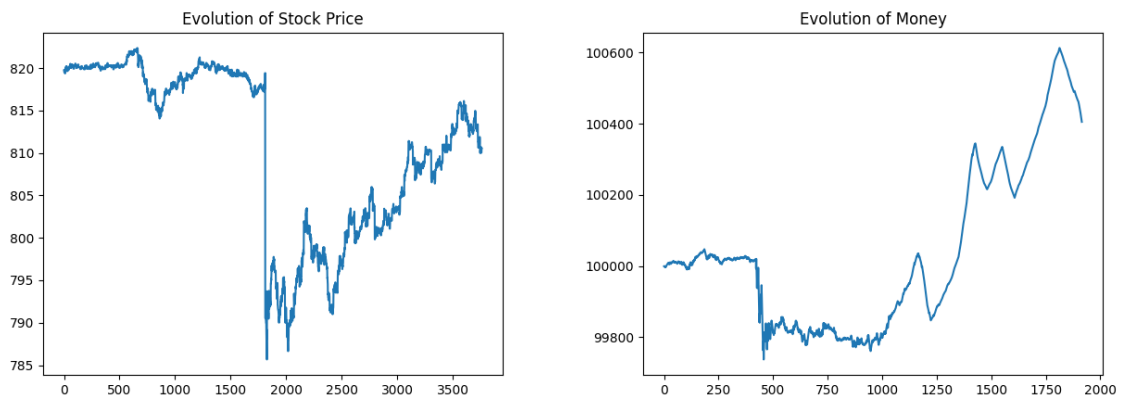


Figure 8 – Evolution of stock price for Tesla Stock and Evolution of Money for a batch_size of 1000 and a STATE_DIM of 60

For a batch_size of 1000 and a STATE_DIM of 80:

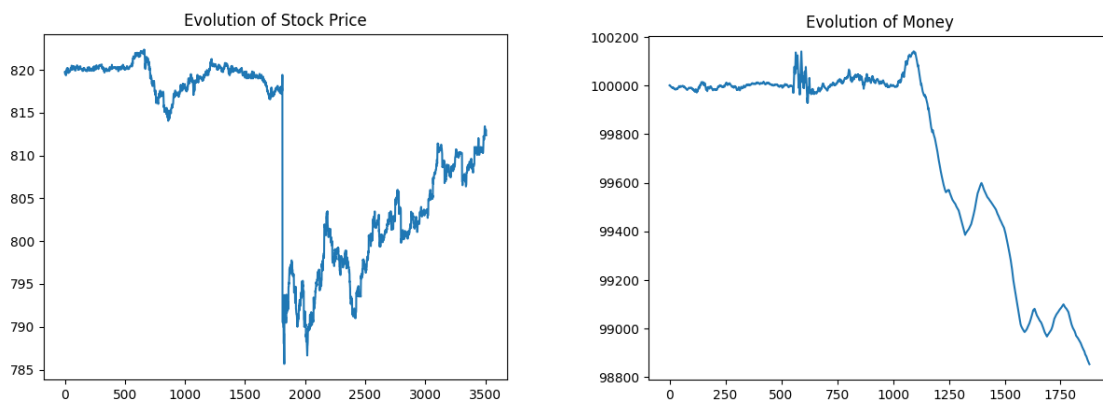


Figure 9 – Evolution of stock price for Tesla Stock and Evolution of Money for a batch_size of 1000 and a STATE_DIM of 80

For a batch_size of 100 and a STATE_DIM of 60:

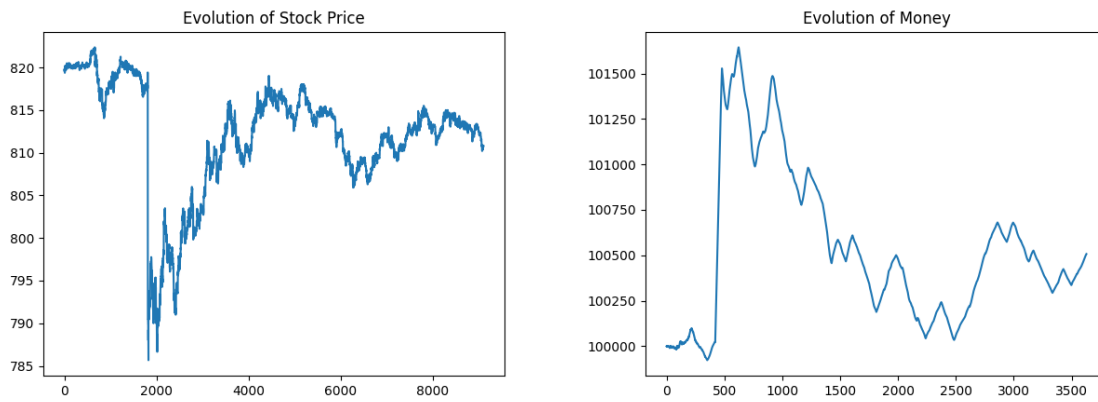


Figure 10 – Evolution of stock price for Tesla Stock and Evolution of Money for a batch_size of 100 and a STATE_DIM of 60

Again the theory of low STATE_DIM and low batch_size has confirmed that this is the best way to use the algorithm.

3. EURUSD pair

Let's try something different now. See how the algorithm performs for a forex pair.

For a `batch_size` of 1000 and a `STATE_DIM` of 60:

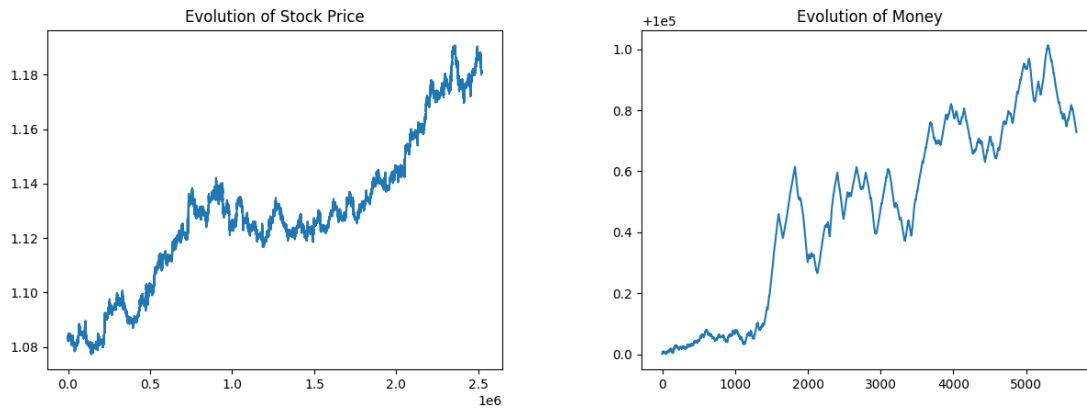


Figure 11 – Evolution of stock price for EURUSD pair and Evolution of Money for a `batch_size` of 1000 and a `STATE_DIM` of 60

For a `batch_size` of 1000 and a `STATE_DIM` of 80:

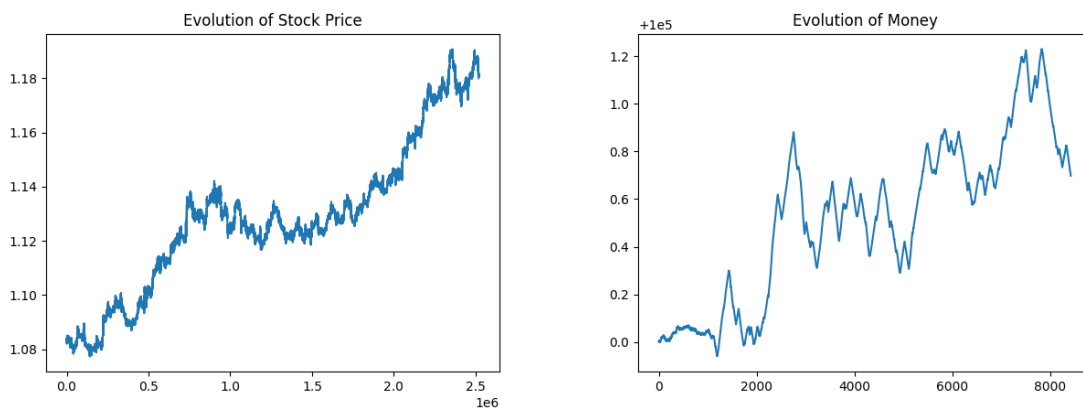


Figure 12 – Evolution of stock price for EURUSD pair and Evolution of Money for a `batch_size` of 1000 and a `STATE_DIM` of 80

For the forex pair EURUSD, the `batch_size` variable does not matter so much as the `STATE_DIM` variable. For this pair, a lot of data is saved and proper training can be done. The money plots are just the power of influence of the `STATE_DIM` variable in the algorithm and his decision-making process. The two plots look very similar, which implies the algorithm working as expected and performing well in the training.

4. Netflix Stocks

Now let's keep an eye on the Netflix stock price, the one that during these months was strongly growing due to pandemics.

For a `batch_size` of 1000 and a `STATE_DIM` of 60:

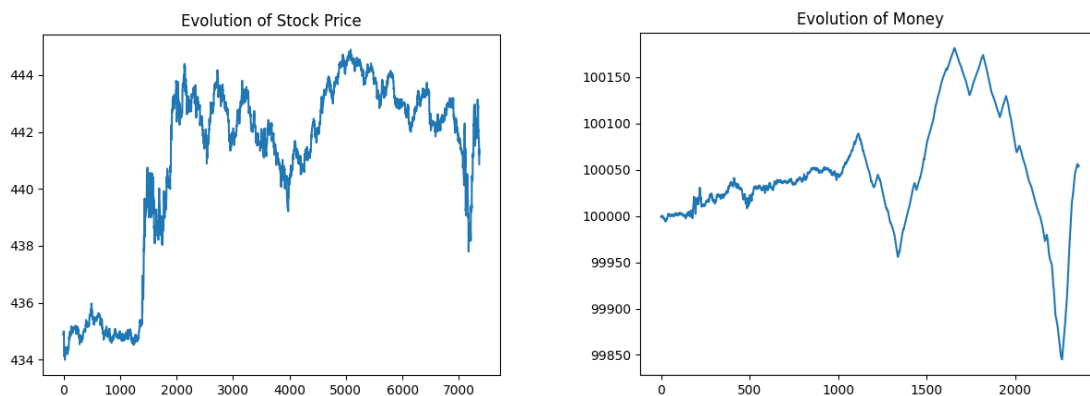


Figure 13 – Evolution of stock price for Netflix stock and Evolution of Money for a `batch_size` of 1000 and a `STATE_DIM` of 60

The effect of high `batch_size` can be seen here. The Netflix stock sky-rocketed and the DQN algorithm learned that it was a good idea to buy the stock. At the same time, he was late for the party, and a small crash affected the balance in a way that it was worse than before the stock jumping so high.

For a batch_size of 1000 and a STATE_DIM of 80:

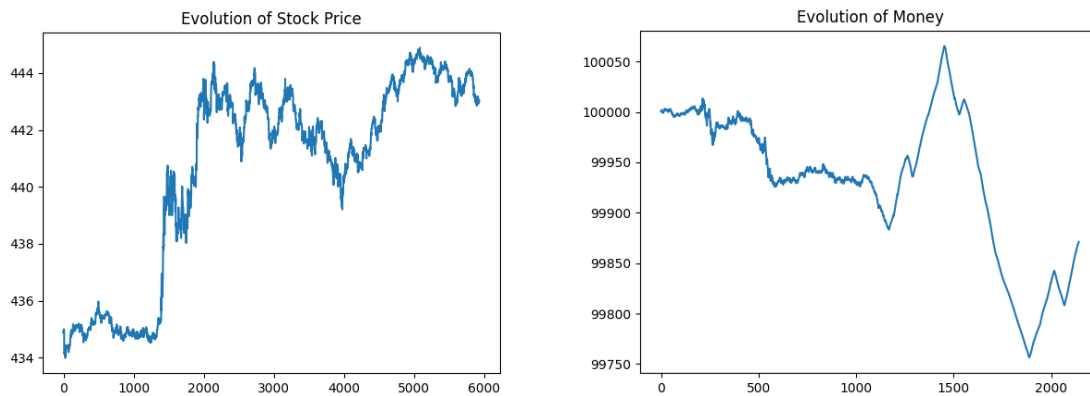


Figure 14 – Evolution of stock price for Netflix stock and Evolution of Money for a batch_size of 1000 and a STATE_DIM of 80

For a batch_size of 100 and a STATE_DIM of 60:

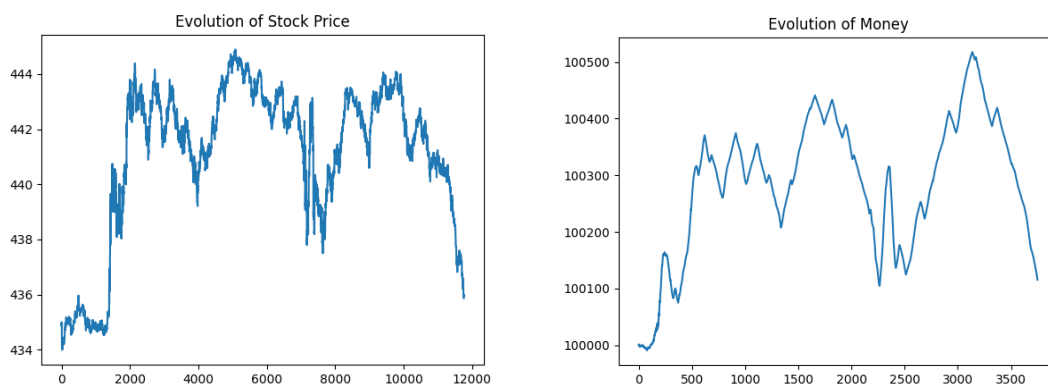


Figure 15 – Evolution of stock price for Netflix stock and Evolution of Money for a batch_size of 100 and a STATE_DIM of 60

The results of these tests were fascinating. We could see how strong the training is on the algorithm by tweaking the `batch_size` parameter. The sooner the training starts the better are the results. Also, we can see how well it learns to follow trends in the stock market and how much the `STATE_DIM` parameter affects it.

The stop loss and take profit bracket orders show us the advantages of their usage too. We could see the massive drop in prices for the Apple and Tesla stocks that did not affect us long-term. A rapid massive drop in price gets the stop loss triggered and helps us not lose money. On the other hand, not a great profit was made because the profit was taken early in the Netflix stock. Things can be seen vice versa as for a short sell on the Netflix stock in that period or a short sell on the Apple and Tesla stocks.

These trend following results come from the fact that the algorithm works with the last prices. Professional traders use special indicators that many are calculated from the last prices such as the MACD indicator [6]. So if a professional trader can make money using the last prices, the task is doable and the algorithm works just as expected.

V. Policy gradient comparison

To compare the DQN results to other very good considered reinforcement learning algorithms I have implemented the Policy Gradient algorithm. The results of the DQN algorithm made me think that Policy Gradient should perform much better. The DQN algorithm had better results with more recent prices so that made me think of an algorithm which "learns" live. The policy gradient algorithm focuses only on the most recent actions and does not count the past actions in training. So the trend following theory should prove that the policy gradient algorithm would suit better the environment of the stock market.

1. The implementation of the Policy Gradient algorithm

The policy gradient class has the following variables:

G - a variable that is calculated during the learning process of the algorithm. It makes a mean of the reward received by the actions and based on that it will take the future actions.

state_memory - a list that stores the states of the current episode.

action_memory - a list that stores the actions of the current episode.

reward_memory - a list that stores the rewards of the current episode.

policy - it represents the neural network that determines the policy taken by the algorithm.

predict - it represents the neural network that returns the probability of what action to take during a given state.

end - this variable represents the environment we are working on. Currently set to the database environment to see how the algorithm performs.

The class implements the following methods:

```
def choose_action(self, observation):
```

This method has an observation parameter that represents the current state (the last STATE_DIM prices). It reshapes the observation so that it fits the input layer of the network. It then feeds the "predict" network the input and gets the probabilities for the two actions (the buy action and the short-sell action) and selects an action.

```
def store_transition(self, observation, action, reward):
```

This method has as parameters the observation, action, and reward for an episode. It is used to store the data in the provided list later used for training. The observation is stored in the state_memory list, the action is stored in the action_memory list and the reward is stored in the reward_memory list.

```
def learn(self):
```

This method represents the learning part for the neural networks. It takes the memory lists and converts them to NumPy arrays. Then it calculates the G factor accordingly to the action-reward relation. The policy network then is trained based on the state and the G factor calculated for that specific state. The state_memory, action_memory, reward_memory are reinitialized to empty lists. This is the reason why policy network learns "live" and the reason that it pays more value to recent actions, therefore on the market trend.

This class also uses some helper functions that give it usability:

```
def build_policy_network():
```

This function defines the neural network used by the predict and policy variables. It uses the same three-layer fully connected network as the DQN algorithm does, so a comparison between the two would be possible. The difference is that a custom loss function is used to better adjust the network prediction of the probabilities. The two even use the same optimizer, Adam, with the same learning $1e-4$. This method returns the two created networks and initializes the class variables.

```
def learnDb(agent):
```

This function is the training loop that makes the policy gradient algorithm learn. It interacts with the environment given to him as a variable. It explores the environment. Based on a given state, it sends the action to the environment and receives the reward, the next state and sets the done variable to true if it is the end of an episode. These are stored in the algorithm lists. At the end of an episode, learning starts.

2. Results

As the DQN tests were run, I did the same with the Policy Gradient algorithm. The database environment was used to see the results in three months. All the tests were run with the two bracket orders set as follows:

```
take_profit_percent = 1.001
```

```
stop_loss_percent = 0.999
```

The only parameter to modify is the STATE_DIM, as the batch_size is irrelevant for the Policy Gradient algorithm.

1. Apple Stocks

For a STATE_DIM of 60 I had the following results:

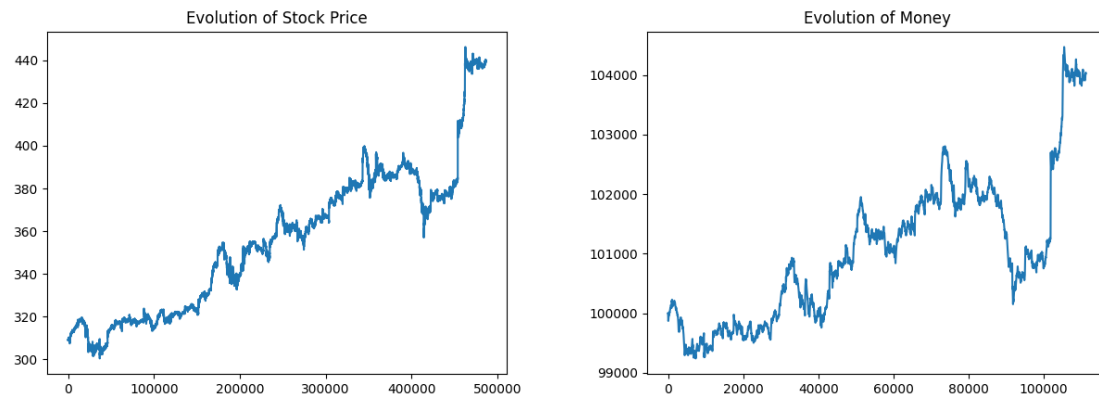


Figure 16 – Evolution of stock price for Apple stock and Evolution of Money for a STATE_DIM of 60

For a STATE_DIM of 80 I had the following results:

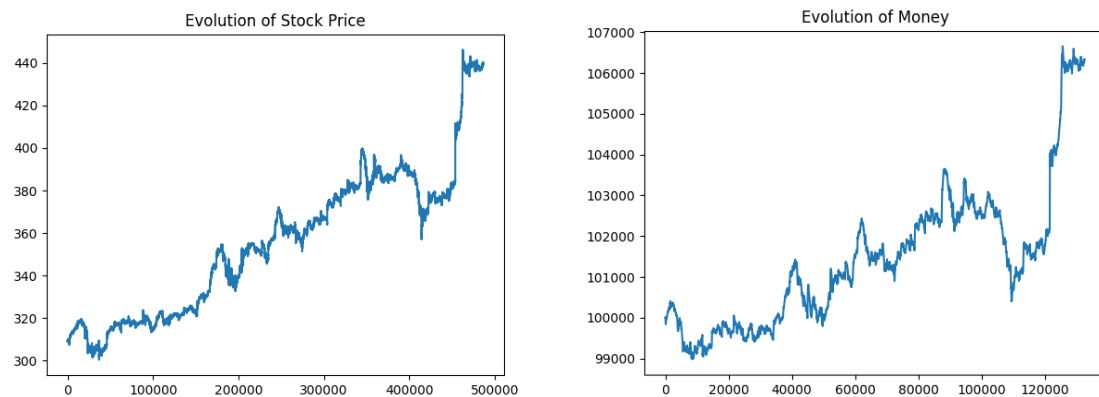


Figure 17 – Evolution of stock price for Apple stock and Evolution of Money for a STATE_DIM of 80

Seems that the higher the STATE_DIM the bigger the amount of money would be generated.

For a STATE_DIM of 200 I had the following results:

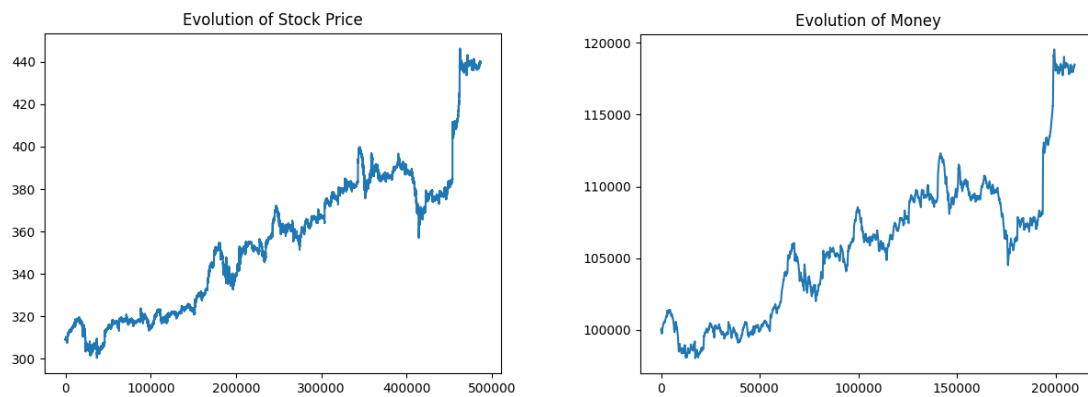


Figure 18 – Evolution of stock price for Apple stock and Evolution of Money for a STATE_DIM of 200

Although the money line follows the line of the stock it is not a bad thing. The algorithm sees the potential of the stock going up and follows the trend. Also, even if not visible by looking at the lines, the higher the STATE_DIM the higher the amount of money made by looking at the numbers.

Running it through more data and with a higher STATE_DIM seem to be beneficial in the learning process, achieving the best results yet.

2. Tesla Stocks

For a STATE_DIM of 60 I had the following results:

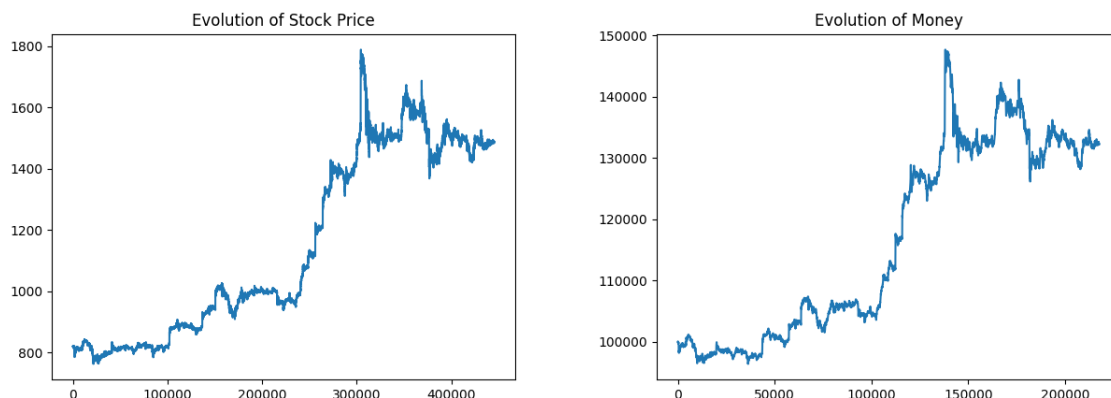


Figure 19 – Evolution of stock price for Tesla stock and Evolution of Money for a STATE_DIM of 60

For a STATE_DIM of 80 I had the following results:

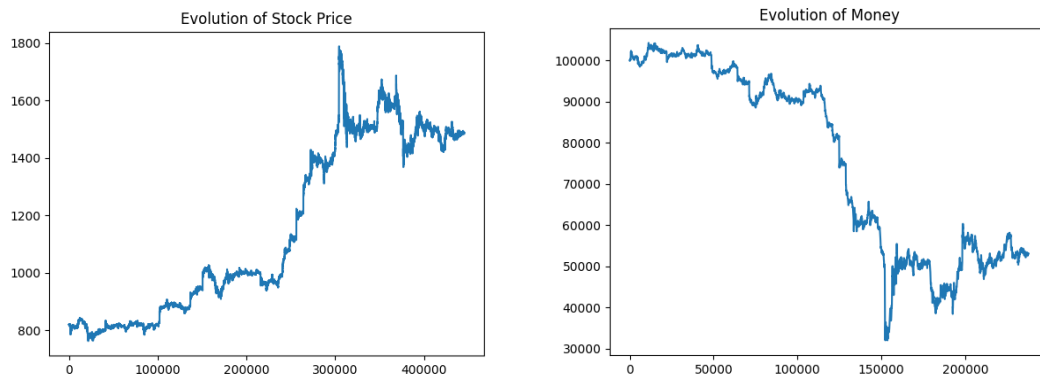


Figure 20 – Evolution of stock price for Tesla stock and Evolution of Money for a STATE_DIM of 80

Even though for the Apple stock was a good idea to use a bigger STATE_DIM, for the Tesla stock it does not pick the trend and it is betting against it. A lower STATE_DIM is the way to switch trends faster, even though it does not make so much profit.

For a STATE_DIM of 200 I had the following results:

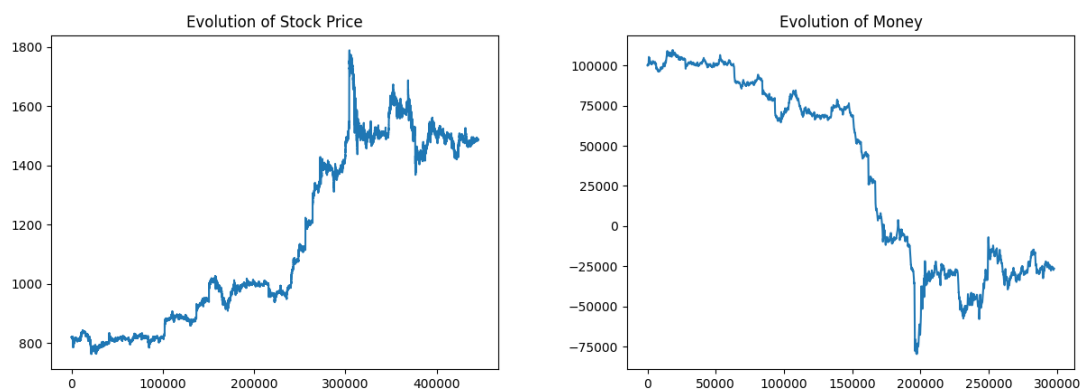


Figure 21 – Evolution of stock price for Tesla stock and Evolution of Money for a STATE_DIM of 200

Even more money is lost with a higher STATE_DIM.

3. Algorithm confronting

Policy gradient is following the stock price and the results prove this fact. This is not a bad thing, as if it identifies a trend, it will follow it as long as it is a good one. I can see the potential of the Policy gradient algorithm in stock trading and I can see why people prefer it instead of the DQN. It identifies a trend much faster and can strongly follow it.

The main difference between those two is the learning process. The DQN with its past knowledge shows us he can pick and recognize trends in the market, but with skepticism. The algorithm knows that nothing is sure reflecting more of a human mind. Even though in an ascending market, the DQN algorithm still makes short-sells resulting in much lower profit.

On the other hand, there is the Policy Gradient algorithm. Its approach to learning is much more greedy and we can see this. It picks a trend that seems profitable and sticks to it, even though it is not the best. Fewer tests were required to see what the Policy Gradient algorithm is up to.

All the learning was done with an episode length of 20. After 20 consecutive trades, the algorithms started learning what was good and what was bad in the actions they took. It may not have a great impact on the strategy they take, even less for the DQN algorithm that knows the past actions. Probably a much bigger episode length is required for the Policy gradient algorithm, as it learns only from the current episode, while the DQN is in a continuous learning process.

In my opinion, the DQN algorithm is the winner. It acts more like a human and even the profit it makes is not such a big number, you have a much higher chance of it making a profit. For sure, the Policy gradient approach can be far more profitable, but my implementation shows me that it can pick far more often the wrong trend of the market. The related work sector of this paper prefers Policy gradient, but as I know, they switch the main algorithm used based on their results on the market. Which one performs the best in a given period, will be the main algorithm for trading for the next period. This shows that sometimes, Policy Gradient algorithms cannot perform well in some situations and have to be replaced by

others. That being said, it gave me confidence that my Policy Gradient algorithm is not a bad one, but it lacks in certain situations. The main focus of this thesis was to give an alternative to Policy Gradient algorithms. The DQN seemed the right option to follow when it comes to reinforcement learning in this sector.

VI. Conclusion

To summarize the thesis it is clear to see the power of reinforcement learning in the stock market. If you trade at random, with a given change of 50/50 and the bracket orders in a unidirectional market you won't make any profit but also don't lose any money. You will only get a certain percentage of money for a given trade or you will lose that amount also with a chance of 50/50. The probability tells us that trading like this to infinity you will end up with the same amount of money. The reinforcement learning algorithm can incline this balance and so it does. From 100 trades where 51 are good trades and 49 are bad trades, with the help of equally distributed bracket orders you will always make a profit and the algorithm takes advantage of this situation.

1. Innovation

The idea of trading with algorithms on the stock market is not new. There have always been people trying to take advantage of this. Even nowadays a lot of trading algorithms can be found online that promise you a good amount of profit by using them. Also, the introduction of reinforcement learning algorithms to the stock market is not new at all. There are lots of papers that focus on this topic.

What has newly been added is the way I have perceived the interaction between the agent and the environment. The introduction of only two actions instead of three seemed to be a brilliant idea. Why would a hold action be useful? How long would you hold? This action was irrelevant to me, so instead of using the three most popular actions of buying, selling, and holding I took advantage of the fact that the market moves both ways. You also bet that a given stock will go down and implemented short selling. You can make a profit both ways. Instead of focusing just on the fact the market is going up and decide when to buy, how much to hold and when to sell, I tried just to predict the trend direction. If you know the trend you know exactly what action to take. Then, because an action that tells you when to stop does not exist, I implemented the idea of bracket orders to the algorithm. Always take a small win or a small loss. It was a good idea as we can see it helped us not to lose big amounts when the market was crashing. Also, the small wins add up, as the algorithm learns what action to take.

Although the DQN algorithm was already used in stock trading, the two environments used are completely new. They are both useful in their ways. The database environment lets you test the implementation of the algorithm to see how it performs, and the live environment gives you the possibility to use the algorithm in a real scenario.

2. Further research

The algorithm is far from perfect. New ideas of a neural network can be implemented instead of the one used. The fully connected network does not take advantage of the fact that the prices represent a time series, so maybe a recurrent network will do better. Probably the integration of an LSTM [7] layer would do a better job handling time series. Networks that use LSTM cells are already in use to predict the movement of stock prices in the far future.

What bothers me is the fact that the algorithm stops early if a trade is profitable. Instead of a take-profit bracket order, a trailing stop one would probably be better. The trailing stop order is an implementation that does not close your order as long as it goes up and does not have pullbacks. It makes an already profitable trade even more profitable.

Also, more inputs can be added to the network as more information can be more useful, such as the trading volume. It shows you how much the stock is traded, which affects the volatility of the price. Maybe an input like Japanese candlesticks [8] would help the understanding of the stock market even further.

The Replay Memory of the DQN algorithm can also be modified. As we have seen, the Policy Gradient algorithm had a great performance in the stock market. By lowering the Replay Memory of the DQN, it can store only more recent actions, that meaning it will pay more attention to them. I don't consider things happening such far in the past to affect the nowadays day-trading from a human perspective.

The Replay Memory of the DQN algorithm can also be modified. As we have seen, the Policy Gradient algorithm had a great performance in the stock market. By lowering the Replay Memory of the DQN, it can store only more recent actions, that meaning it will pay more attention to them. I don't consider things happening such far in the past to affect the nowadays day-trading from a human perspective.

vii. References

- [1] Reinforcement Learning: An Introduction, Richard S. Sutton and Andrew G. Barto, 2014, 2015
- [2] Playing Atari with Deep Reinforcement Learning, Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller, 2015
- [3] Computing Machinery and Intelligence, Alan Turing, 1950
- [4] Deep Reinforcement Learning for Automated Stock Trading: An Ensemble Strategy, Hongyang Yang, Xiao-Yang Liu , Shan Zhong , and Anwar Walid, 2020
- [5] An Introduction to Python, Guido van Rossum, 2003
- [6] A test of the MACD trading strategy, Bill Huang, Yong Soo Kim, 2006
- [7] Long Short-Term Memory, Sepp Hochreiter, Jurgen Schmidhuber, 1997
- [8] The Application of Japanese Candlestick Trading Strategies in Taiwan, Yeong-Jia Goo, Dar-Hsin Chen, Yi-Wei Ckang, 2007

viii. List of figures

Figure 1 – The Alpaca Trade site overview of the account

Figure 2 – The orders menu on the Alpaca Trade site.

Figure 3 – Snapshot of the database from the Navicat database manager

Figure 4 – Snapshot of the XAMPP Control Panel

Figure 5 – Evolution of stock price for Apple Stock and Evolution of Money for a batch_size of 1000 and a STATE_DIM of 60

Figure 6 – Evolution of stock price for Apple Stock and Evolution of Money for a batch_size of 1000 and a STATE_DIM of 80

Figure 7 – Evolution of stock price for Apple Stock and Evolution of Money for a batch_size of 100 and a STATE_DIM of 60

Figure 8 – Evolution of stock price for Tesla Stock and Evolution of Money for a batch_size of 1000 and a STATE_DIM of 60

Figure 9 – Evolution of stock price for Tesla Stock and Evolution of Money for a batch_size of 1000 and a STATE_DIM of 80

Figure 10 – Evolution of stock price for Tesla Stock and Evolution of Money for a batch_size of 100 and a STATE_DIM of 60

Figure 11 – Evolution of stock price for EURUSD pair and Evolution of Money for a batch_size of 1000 and a STATE_DIM of 60

Figure 12 – Evolution of stock price for EURUSD pair and Evolution of Money for a batch_size of 1000 and a STATE_DIM of 80

Figure 13 – Evolution of stock price for Netflix stock and Evolution of Money for a batch_size of 1000 and a STATE_DIM of 60

Figure 14 – Evolution of stock price for Netflix stock and Evolution of Money for a batch_size of 1000 and a STATE_DIM of 80

Figure 15 – Evolution of stock price for Netflix stock and Evolution of Money for a batch_size of 100 and a STATE_DIM of 60

Figure 16 – Evolution of stock price for Apple stock and Evolution of Money for a STATE_DIM of 60

Figure 17 – Evolution of stock price for Apple stock and Evolution of Money for a STATE_DIM of 80

Figure 18 – Evolution of stock price for Apple stock and Evolution of Money for a STATE_DIM of 200

Figure 19 – Evolution of stock price for Tesla stock and Evolution of Money for a STATE_DIM of 60

Figure 20 – Evolution of stock price for Tesla stock and Evolution of Money for a STATE_DIM of 80

Figure 21 – Evolution of stock price for Tesla stock and Evolution of Money for a STATE_DIM of 200