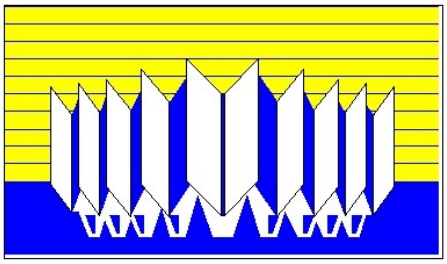
TUNISIAN REPUBLIC
MINISTRY OF HIGHER EDUCATION, OF SCIENTIFIC RESEARCH AND TECHNOLOGY
UNIVERSITY OF TUNIS EL MANAR



Faculty Of Science Of Tunis

Computer Science Department

End Of Year Project (PFA): Algorithmic Trading In Reinforcement Learning

Author:
Ahmed Troudi

Supervisor:
Faouzi Moussa

2017-2018

Abstract

In electronic financial markets, algorithmic trading refers to the use of computer programs to automate one or more stages of the trading process. In this project, reinforcement learning techniques are applied to the problem of algorithmic trading. A dataset relative to a single stock (AAPL of Apple inc.) is used to train the network and test it (70% percent used for training and 30% for testing).While observing the state of the market, the algorithm decides to perform one of three actions: buy, hold or sell. The goal is to come up with a profitable strategy based on a single parameter which is price prediction.Coming up with such a strategy in Supervised Learning can be extremely difficult and requires a certain level of expertise in both Machine Learning and Finance.Emphasis has been given to explain the choice of the framework and the algorithm throughout this report.

Contents

Chapter 1

Interests and Motivations

1.1 Goal:

This area of machine learning consists in training an agent by reward and punishment without needing to specify the expected action. The agent learns from its experience and develops a strategy that maximizes its profits. The goal is to come up with a profitable strategy based on a single parameter which is price prediction by trading a single stock under the reinforcement learning framework.

1.2 The Supervised Learning Approach:

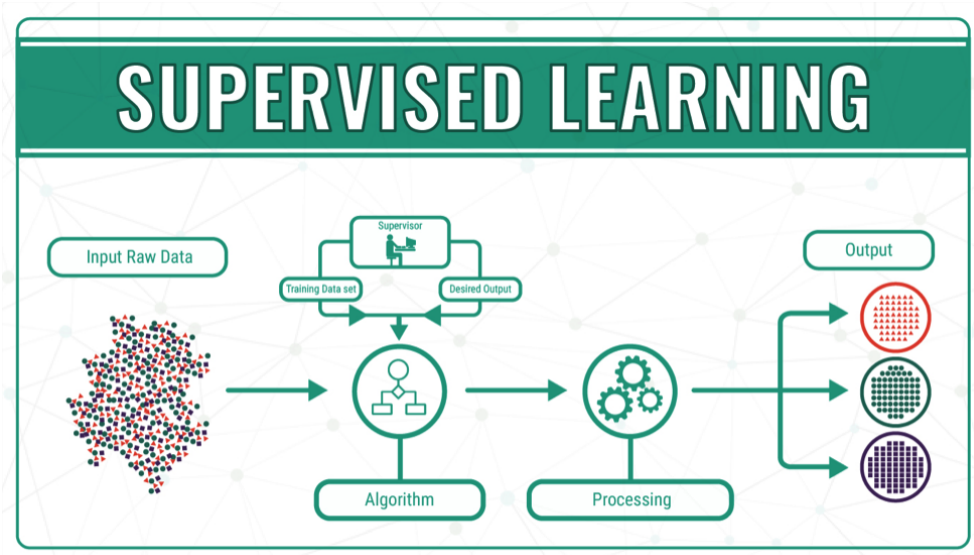


Figure 1.1: Comparison Supervised vs RL

Supervised Machine Learning currently makes up most of the ML that is being used by systems across the world. The input variable (x) is used to connect with the output variable (y) through the use of an algorithm. All of the input, the output, the algorithm, and the scenario are being provided by humans.

The most obvious approach we can take is price prediction. If we can predict that the market will move up we can buy now and then hold.Or, equivalently, if we predict the market goes down, we can sell. However, there are a few problems with this Supervised Learning approach, It does not imply a policy. In a perfect scenario, we buy because we predict that the price moves up, and then it actually moves up. Everything went according to plan. But what if the price had moved down? Would you have sold?

1.3 Problems with this approach:

The rigidity with supervised learning means that you need more than just a price prediction model (unless your model is extremely accurate and robust) and that you can't change strategy mid-trading if you find your predictions to be wrong. In Addition,Stock Data are almost random which makes coming up with a good prediction strategy quite a difficult task especially to someone unexperienced in finance. We also need a rule-based policy that takes as input your price predictions and decides what to actually do

1.4 Solution: Why choose RL?

Here's a list :

- Instead of needing to hand-code a rule-based policy which requires a certain level of expertise in finance, Reinforcement Learning directly learns a policy. There's no need for us to specify rules and thresholds such as "buy when you are more than 75% sure that the market will move up". That's baked in the RL policy, which optimizes for the metric we care about. We're removing a full step from the strategy development process!
- Because the policy can be parameterized by a complex model, such as a Deep Neural network, we can learn policies that are more complex and powerful than any rules a human trader could possibly come up with.
- Because RL agents are learning powerful policies parameterized by Neural Networks, they can also learn to adapt to various market conditions by seeing them in historical data, given that they are trained over a long time horizon and have sufficient memory. This allows them to be much more robust to changing markets.

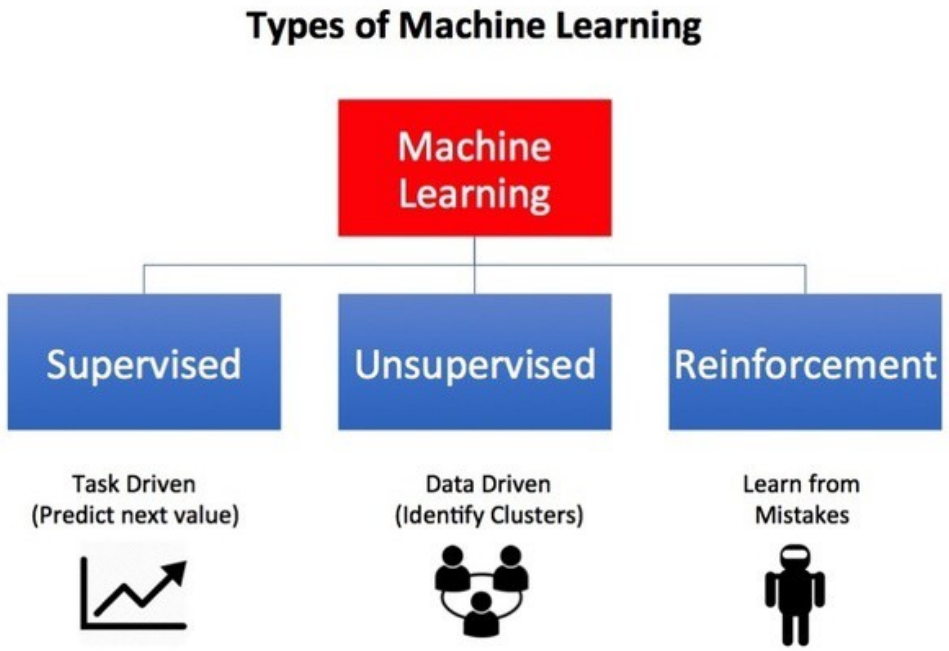


Figure 1.2: Comparison Supervised vs RL

Chapter 2

Theory

2.1 What is Reinforcement Learning ?

Reinforcement learning lies somewhere in between supervised and unsupervised learning. Whereas in supervised learning one has a target label for each training example and in unsupervised learning one has no labels at all, in reinforcement learning one has sparse and time-delayed labels – the rewards. Based only on those rewards the agent has to learn to behave in the environment.

2.1.1 Markov Decision Process

Now the question is, how do you formalize a reinforcement learning problem, so that you can reason about it? The most common method is to represent it as a Markov decision process.

Suppose you are an agent, situated in an environment (e.g. Stock Market). The environment is in a certain state (e.g. market features like prices, whether we're holding the stock). The agent can perform certain actions in the environment (e.g. choose to buy, sell or hold). These actions sometimes result in a reward (e.g. profit). Actions transform the environment and lead to a new state, where the agent can perform another action, and so on. The rules for how you choose those actions are called policy. The environment in general is stochastic, which means the next state may be somewhat random (which is perfect for stock prediction).

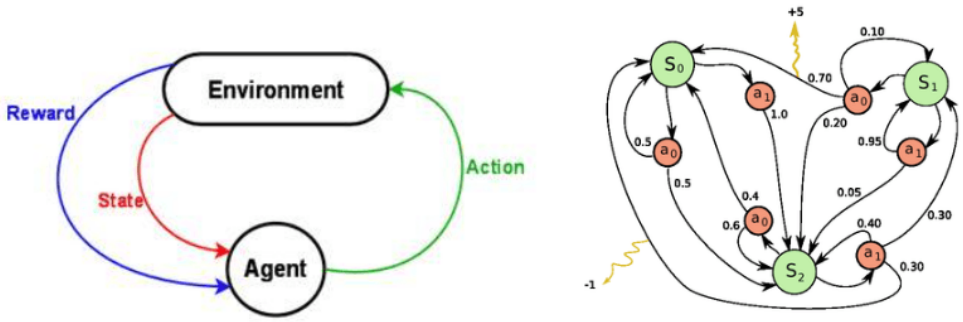


Figure 2.1: Left: reinforcement learning problem. Right: Markov decision process.

The set of states and actions, together with rules for transitioning from one state to another, make up a Markov decision process. One episode of this process (e.g. one iteration of the full dataset) forms a finite sequence of states, actions and rewards:

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, \ldots, s_{n-1}, a_{n-1}, r_n, s_n$$

Here "si" represents the state, "ai" is the action and "ri+1" is the reward after performing the action. The episode ends with terminal state "sn" (e.g. done trading). A Markov decision process relies on the Markov assumption, that the probability of the next state "si+1" depends only on current state "si" and action "ai", but not on preceding states or actions.

2.1.2 Discounted Future Reward

To perform well in the long-term, we need to take into account not only the immediate rewards.
We can easily calculate the total reward for one episode:

$$R_t = r_t + r_{t+1} + r_{t+2} + \ldots + r_n$$

Because our environment is stochastic, we can never be sure, if we will get the same rewards the next time we perform the same actions. The more into the future we go, the more it may diverge. For that reason it is common to use discounted future reward instead:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \ldots + \gamma^{n-t} r_n$$

The discounted future reward at time step t can be expressed in terms of the same thing at time step t+1:

$$R_t = r_t + \gamma(r_{t+1} + \gamma(r_{t+2} + \ldots)) = r_t + \gamma R_{t+1}$$

If we set the discount factor $\gamma = 0$, then our strategy will be short-sighted and we rely only on the immediate rewards. If we want to balance between immediate and future rewards, we should set discount factor to something like $\gamma = 0.9$. If our environment is deterministic and the same actions always result in same rewards, then we can set discount factor $\gamma = 1$.

A good strategy for an agent would be to always choose an action that maximizes the (discounted) future reward.

2.2 Q-learning

Q-learning is a reinforcement learning technique used in machine learning.
In Q-learning we define a function Q(s, a) representing the maximum discounted future reward when we perform action a in state s, and continue optimally from that point on.

$$Q(s_t, a_t) = max\ R_{t+1}$$

The way to think about Q(s, a) is that it is "the highest possible profit at the end of trading after performing action a in state s". It is called Q-function, because it represents the "quality" of a certain action in a given state. This may sound like quite a puzzling definition. How can we estimate the score at the end of trading, if we know just the current state and action, and not the actions and rewards coming after that? We really can't. But as a theoretical construct we can assume existence of such a function

$$\pi(s) = argmax_a\ Q(s, a)$$

Here Π represents the policy, the rule how we choose an action in each state.

OK, how do we get that Q-function then? Let's focus on just one transition <s, a, r, s'>. Just like with discounted future rewards in the previous section, we can express the Q-value of state s and action a in terms of the Q-value of the next state s'.

$$Q(s, a) = r + \gamma max_{a'} Q(s', a')$$

This is called the Bellman equation. The main idea in Q-learning is that we can iteratively approximate the Q-function using the Bellman equation.

2.3 Deep-Q-Network

In the simplest case the Q-function is implemented as a table, with states as rows and actions as columns. Except that, for datasets that are too large, there are way too many states to represent in a table and this data structure becomes unpractical. This is the point where deep learning steps in. 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 as input and outputs the corresponding Q-value.

2.3.1 Network Architecture

Here's a list :

- Layer 1: Dense ($Units = 32$, $input_dimension = self.state_size$, $activation = ReLU$)
- Layer 2: Dense ($Units = 32$, $activation = ReLU$)
- Layer 3: Dense ($Units = 8$, $activation = ReLU$)

These parameters are chosen through trial and error, too many units results in overfitting and too few causes underfitting.

2.3.2 Experience Replay

By now we have an idea how to estimate the future reward in each state using Q-learning and approximate the Q-function using a neural network. But 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.
The most important trick is experience replay. During gameplay 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.

2.4 Exploration-Exploitation

Q-learning attempts to solve the credit assignment problem – it propagates rewards back in time, until it reaches the crucial decision point which was the actual cause for the obtained reward.
A simple way to implement this is ϵ-greedy exploration – with probability ϵ choose a random action, otherwise go with the "greedy" action with the highest Q-value. In this system we 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.

2.5 Deep Q-learning Algorithm

```
initialize replay memory D
initialize action-value function Q with random weights
observe initial state s
repeat
      select an action a
            with probability ε select a random action
            otherwise select a = argmax_a' Q(s,a')
      carry out action a
      observe reward r and new state s'
      store experience <s, a, r, s'> in replay memory D

      sample random transitions <ss, aa, rr, ss'> from replay memory D
      calculate target for each minibatch transition
            if ss' is terminal state then tt = rr
            otherwise tt = rr + γmax_a' Q(ss', aa')
      train the Q network using (tt - Q(ss, aa))^2 as loss

      s = s'
until terminated
```

Chapter 3

Evaluation and Results

3.1 Technologies Used:

Figure 3.1: Anaconda Distribution python 3.6

Figure 3.2: Keras Deep Learning Library with Tensorflow-GPU backend

Figure 3.3: Pandas python

3.2 Dataset:

The Dataset the I used Is "AAPL.csv" relative to the stock of APPLE INC, I split it into two datasets:
"AAPL_Train": contains 70% of the data and used for training
"AAPL_Test": contains 30% of the data and used for testing

3.3 Training:

The training takes a long time; close to 10 hours and is performed on an Nvidia GPU.
States are represented by list that contains prices for the previous n days and moves forward by one day after each action.
Here are the rewards for each actions For action buy $reward = 0$
For action sell $reward = (sellingprice - buyingprice)/buyingprice$
For action hold $reward = 0$

3.4 Testing:

The long training duration makes testing a difficult task. Models (which are versions of the neural network) are logged for every 10 episodes, meaning that for 1000 episodes we have 100 models. During testing, we load one of those models and use it on the testing dataset. The algorithm is profitable, It was tested against random actions and despite the profit being low, it consistently performs better.
Test Case:
Starting with a portfolio of 10000, it ends with a portfolio of 10029, meaning a profit of 29. for model n°200 on "AAPL_Test.csv"
Compared to a random action model which is model n°0 that results in negative profit of around :-3243.

3.5 Conclusion

While the model seems to be a bit unstable and the profit too low for a real life scenario.We may find that the fact that this algorithm produces a profit from price prediction alone on a single stock is quite interesting and shows that reinforcement learning is a very viable framework for algorithmic trading if we take into account additional features about the market

Bibliography

[1] Denny Britz. <http://www.wildml.com/2018/02/introduction-to-learning-to-trade-with-reinforcement-learn
>, 2018.

[2] Keon Kim. <https://keon.io/deep-q-learning//>, 2017.

[3] Tambet Matiisen. <https://ai.intel.com/demystifying-deep-reinforcement-learning/>, 2015.

[4] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.*

[5] Daniel Zakrisson. <https://hackernoon.com/the-self-learning-quant-d3329fcc9915/>, 2016.