Technical University of Crete
School of Electrical and Computer Engineering

Reinforcement Learning Project Report
Stock Market Trading

Konstantinos Palaiologos

July, 2024

Phase I: Policy iteration

1. **MDP environment for binary-state stock trading**

We assume that there is a selection of N different stocks and each one of them can be either in state *High* (H) or in state *Low* (L). We can formulate this problem of stock day-to-day trading as a Markov Decision Process with the following components:

- **States**:
  A state should include the state of every stock at current round, plus the number of the stock we are currently holding.

$$\mathcal{S} = \{(i, s_1, s_2, \ldots, s_N) \mid s_i \in \{H, L\}, i \in \{1, 2, \ldots, N\}\}$$

  e.g.

$$S_t = \{1, H, L, \ldots\}$$
$$S_{t+1} = \{3, H, H, \ldots\}$$

- **Actions**:
  An action should denote which stock we are holding (or plan to hold) at each round. This will determine whether a transaction fee will be payed or not.

$$\mathcal{A} = \{1, 2, \ldots, N\}$$

- **Transition probabilities**:
  Every stock has 4 transition probabilities for its 2 Markov states. The total transition probability depends on the N-Markov chains (not the action).

$$\text{Stock } 1 : p^1_{HH} = \ldots, \quad p^1_{HL} = 1 - p^1_{HH}, \quad p^1_{LL} = \ldots, \quad p^1_{LH} = 1 - p^1_{LL}$$

$$\text{Stock } 2 : p^2_{HH} = \ldots, \quad p^2_{HL} = 1 - p^2_{HH}, \quad p^2_{LL} = \ldots, \quad p^2_{LH} = 1 - p^2_{LL}$$

$$\vdots$$

e.g.

$$s = \{\cdot, H, H, L, \dots\}$$
$$s' = \{\cdot, L, H, H, \dots\}$$
$$\mathcal{P}(s, \cdot, s') = p^1_{HL} \cdot p^2_{HH} \cdot p^3_{LH} \cdot \dots$$

- **Rewards**:

$$\text{Stock } 1 : R^1_H = \dots, \quad R^1_L = \dots$$
$$\text{Stock } 2 : R^2_H = \dots, \quad R^2_L = \dots$$
$$\vdots$$
$$\text{Let} \quad S_t = \{i, s_1, s_2, \dots\} \quad \text{and} \quad S_{t+1} = \{i', s'_1, s'_2, \dots\}$$

The immediate reward (knowing *s'*) will be

$$\mathcal{R}(s, a = i', s') = R^{i'}_{s'_{i'}}$$

The expected reward will be

$$\mathbb{E}[\mathcal{R}(s, a = i')] = \begin{cases} p^{i'}_{s_i H} \cdot R^{i'}_H + p^{i'}_{s_{i'} L} \cdot R^{i'}_L & \text{, if } i = i' \quad \text{(keep stock)} \\ p^{i'}_{s_i H} \cdot R^{i'}_H + p^{i'}_{s_{i'} L} \cdot R^{i'}_L - 0.01 & \text{, if } i \neq i' \quad \text{(switch stock)} \end{cases}$$

- **Terminal states**:
There are no terminal states in this process.

1. **Small scenarios: experimenting with the environment**

- We assume that N=2, $\gamma = 0$
We set $R^1_H = 2 \cdot R^2_H$, so that:

stock	R_L	R_H
0	-0.0376	0.2354
1	0.0696	0.1177

The possible states in this environment are:

index	state
0	[0, 0, 0]
1	[0, 0, 1]
2	[0, 1, 0]
3	[0, 1, 1]
4	[1, 0, 0]
5	[1, 0, 1]
6	[1, 1, 0]
7	[1, 1, 1]

By also setting a transaction fee of 13% (0.13) and $\gamma = 0$ and using the transaction probabilities matrix:

stock	p_{HH}	p_{HL}	p_{LL}	p_{LH}
0	0.1560	0.8440	0.1560	0.8440
1	0.0581	0.9419	0.8662	0.1338

we get:

$$\pi^* = [0, 0, 0, 0, 1, 1, 1, 1]$$

This is an "**always stay**" policy, which is expected as any transactions are largely penalized due to the chosen fee. Additionally, the $\gamma = 0$ makes the algorithm "myopic", looking only at the next-step reward, thus enhancing the "always stay" policy.

- Keeping the same rewards and probabilities, but setting a transaction fee of 1% and $\gamma = 0.9$, we get:

  $$\pi^* = [0, 0, 1, 1, 0, 0, 1, 1]$$

  This policy chooses to **switch in some states** (2,3,4,5) and to stay in others (0,1,6,7). To check that this is a correct policy, the expected reward of each (state, action) pair was calculated in the specific environment, using the formula stated in the theoretical section above. For all 8 states, the action decided by the optimal policy indeed yielded a higher expected reward than the alternative, indicating that policy iteration works as it should.

2. **Larger scenarios: runtime vs. stocks number**
The runtime t of the algorithm has an exponential increase with the number of stocks N, which defines the number of states as $N \cdot 2^N$. The suggested environment was first implemented ($price \in (-0.02, 0.1)$, specific probabilities). By experimenting with N up to 8 we collected the following pairs of $N - t$:

N	**time (s)**
2	0.07
3	0.34
4	1.22
5	10.86
6	72.96
7	281.09
8	1868.51

The data points were transformed with a log transformation and fitted with a linear regression model. The predicted values were then exponentiated to get the true time predictions:

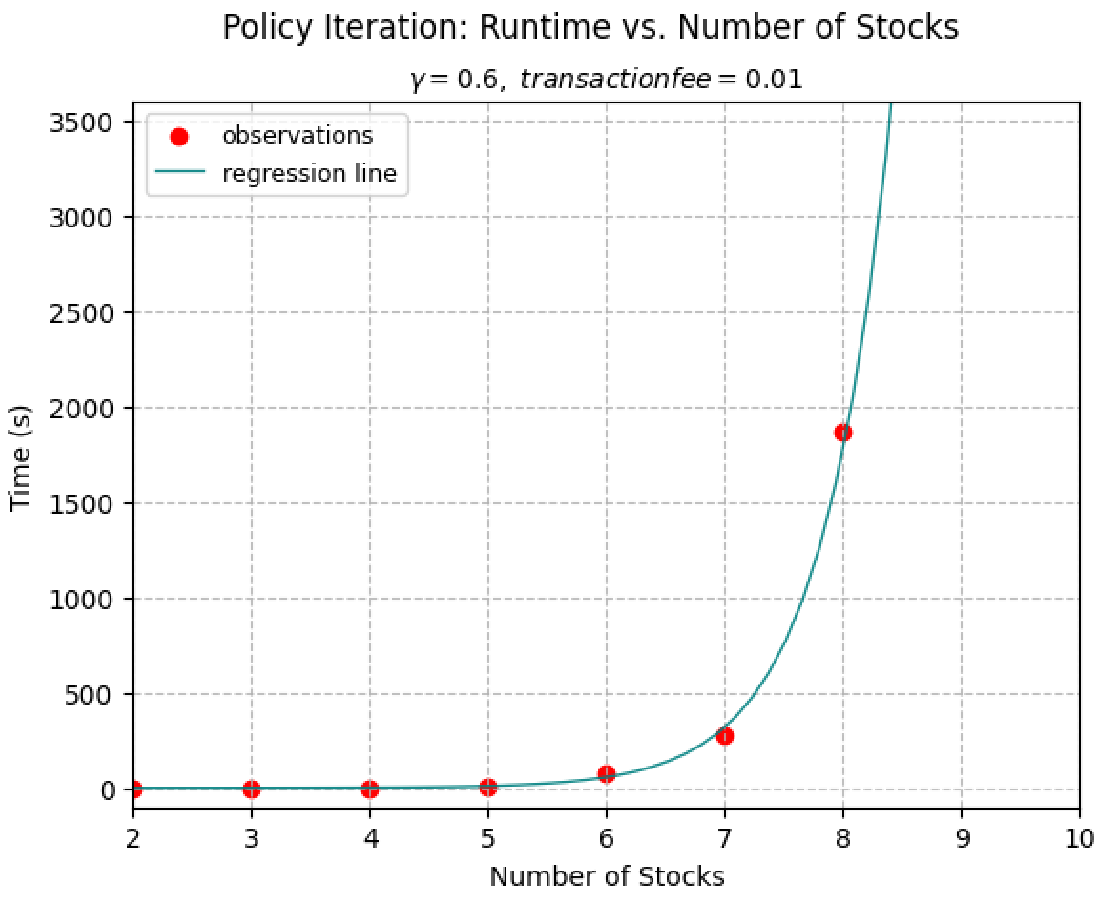

Figure 1: Runtime for different stock numbers

Prior to executing for $N = 8$, the model predicted a runtime of 1703.8 s. The table was then updated with the true value for $N = 8$ and a new regression was performed. Then, policy iteration was executed and converged in 3 iterations. For $N = 9$, $|S| = 4608$ and the updated model predicts a total runtime of 9911.07 s.

Phase II: RL & Deep-RL

Comparing Policy Iteration and Q-learning performance:
Since Q-learning attempts to "learn" the MDP environment over the allowed number of episodes and steps, the optimality of its policy is not guaranteed, unless a convergence condition is used for its termination. For this reason, the performance of its produced policy should be tested against that of Policy Iteration.

The first and most obvious way to do this is by directly comparing the 2 algorithms policies' actions one by one. This can (and did) work in trivial scenarios with few stocks (like those of Q1,2).

In larger scenarios with more stocks, Q-learning usually finds a different optimal policy, but that does not necessarily mean it is a wrong one. Two explanations are the following: either the policy of Q-learning yields a very similar reward with policy iteration, or the differences may correspond to states that happen very rarely and consequently.

This means that the convergence criterion for Q-learning cannot only be complete equality of the two optimal policies, but should rather be based on policy performance. A more appropriate way to compare the two algorithms is to run a simulation for the MDP process using the 2 policies. Afterwards, one can check if the 2 algorithms yield similar average cumulative rewards over a set number of trials.
For this purpose, a simulation function was built, which gets to play a given policy for an N_{trials} number of rounds, over N_{sim} simulations (distinct episodes). The function then returns the mean cumulative reward per round across all simulations.

1. **Q-learning for small scenarios**
   First the built Q-learning algorithm was tested on the 2 small-scale scenarios of phase I ($N = 2$) with parameters:

Parameter	**Value**	**Description**
$N_{episodes}$	100	number of distinct episodes
N_{steps}	100	steps within each episode
α	0.1	learning rate
ϵ	1	ϵ-Greedy exploration
$decay$	0.9	exploration decrease per episode

   * *All of these parameters except N_{steps} remained constant across all* scenarios. The choice of a decaying exploration rate resulted in noise in the initial steps, but smooth performance further on.
   In both cases, Q-learning found the exact same optimal policy as policy iteration in trivial time ($<$ 1min), so there was no justified reason for further performance testing.

2. **Q-learning for larger scenarios**
   For the medium-scale scenarios, the Q-learning's steps per episode were initially increased to $N_{steps} = 1000$, so that the algorithm is given more learning time. For the larger scenarios this number was further increased.

   - For **N=4** and a total of 64 states, we can observe what was described above: Q-learning's optimal policy was at 80% equal to that of Policy Iteration and further testing was needed for convergence evaluation. After trying simulations of the produced policies, the Q-learning policy's mean cumulative reward per trial was found 99.76% that of the "ground truth" policy of P.I. (fig. 2).

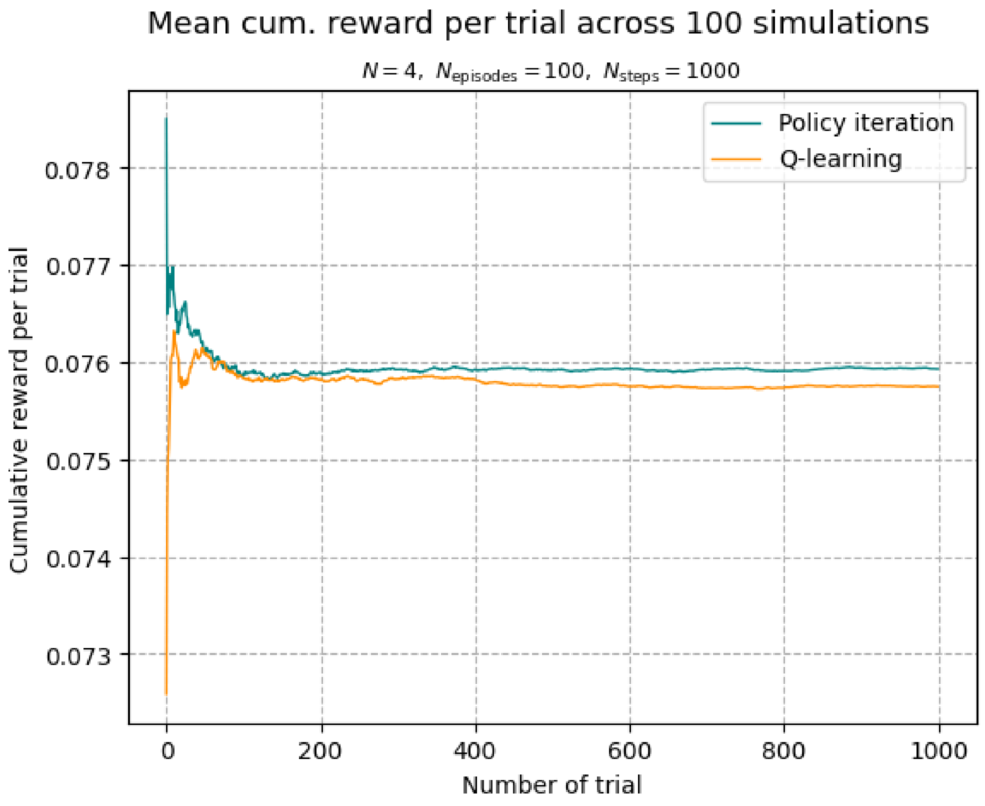

Figure 2: Q-learning's policy vs. "ground truth" policy of P.I., for 4 stocks

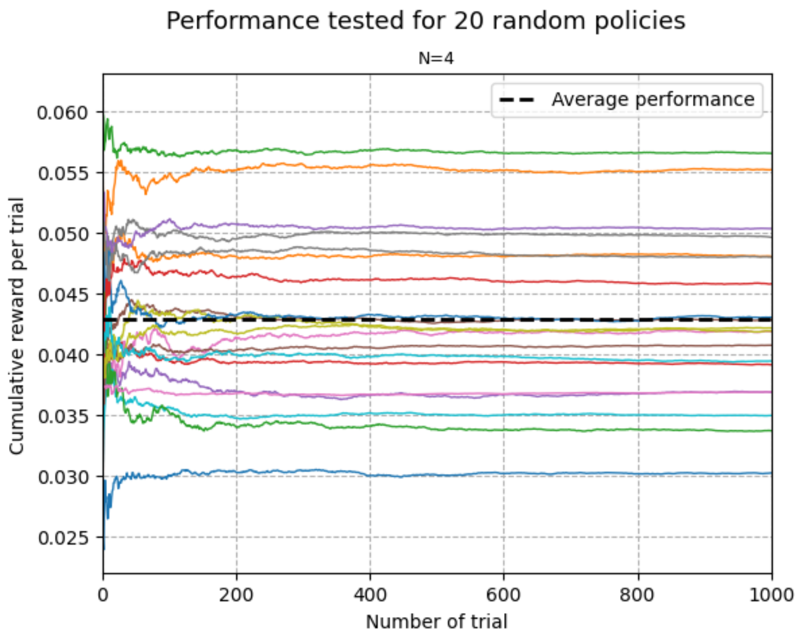

Figure 3: Performances of random policies.

After running the simulation on a number of random policies and comparing to the above plot, it was confirmed that the 2 algorithms indeed produced smart, non-trivial policies that perform optimally.

- For **N=6**, the possible states are 384 and Policy Iteration converged to the optimal policy in t = 5min. Q-learning was executed using the same numbers of episodes and steps/episode as before and terminated in $t = 25s$. The runtime difference is a clear indication that the Q-learning' optimal policy is an approximate one.

  In this case, Q-learning's optimal policy was only at 45% equal to that of P.I., but its performance approximated that of P.I. at 91.85% (fig. 4). This contrast indicates that the action discrepancies between the two policies correspond to cases that happen rarely, or they don't significantly affect policy performance.

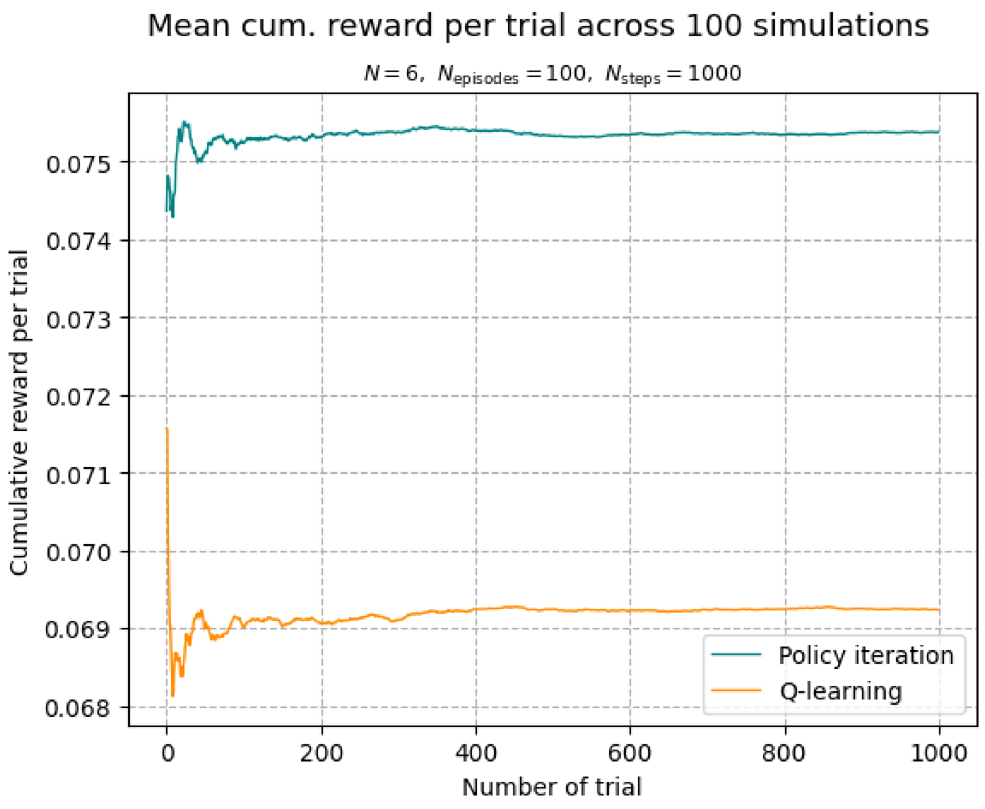

Figure 4: Q-learning's policy vs. P.I.'s policy, for 6 stocks

Pushing the number of steps per episode to N_{steps} = 10000, Q-learning terminated in t = 5min, which is equal to the convergence runtime of P.I. for this environment. Given that, it was still expected to find an approximate, although better, optimal solution, since it is bound to . Indeed, it yielded a closer approximation of P.I.'s policy, with a 58% equality and a 96.92% of its performance (fig. 5).

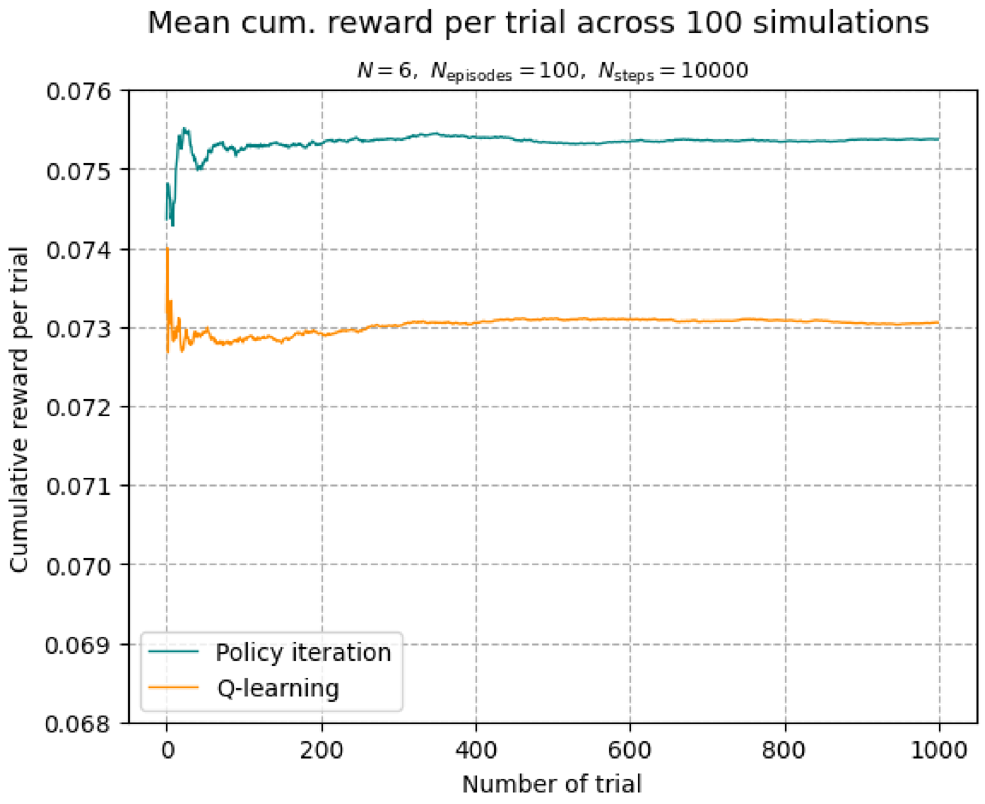

Figure 5: A closer Q-learning approximation, for 6 stocks

- For **N=7**, the possible states are 896 and Policy Iteration converged in t = 26min. For $N_{steps} = 10^4$ steps per episode, Q-learning terminated in t = 4min, finding a 37% equal policy to that of P.I. and with 89.47% its performance (fig. 6).

  To find a better policy, the Q-learning steps per episode were increased to $N_{steps} = 10^5$ and the algorithm terminated in t = 46min. It successfully found a better optimal policy, which was 65% equal to that of P.I. and with 97.87% its performance (fig. 7). This was the largest scenario that was tested for the project's purposes.

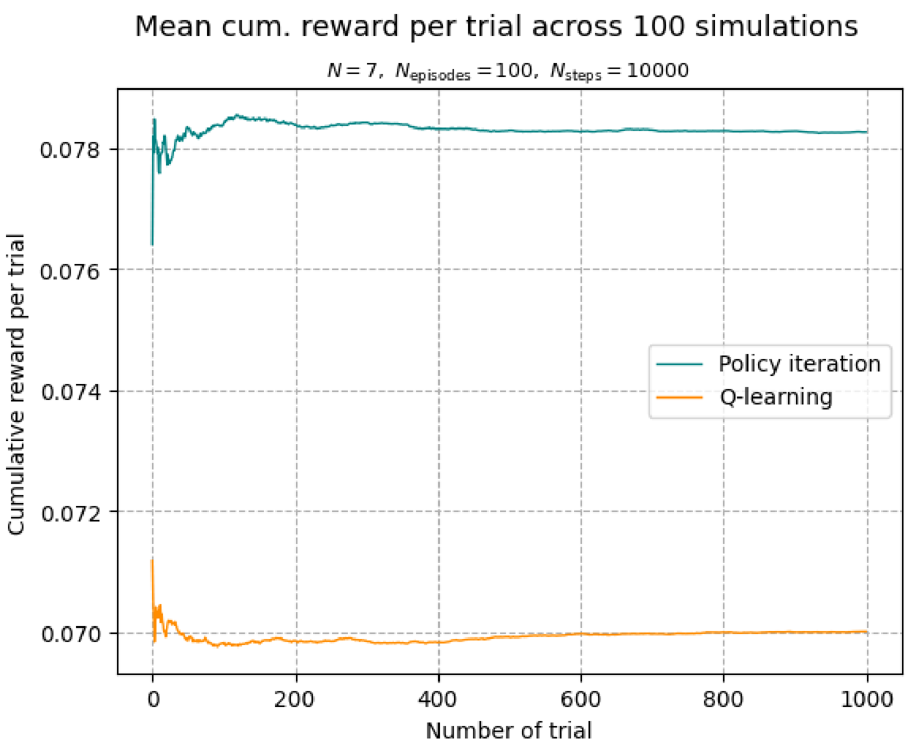

Figure 6: Q-learning's policy vs. P.I.'s policy, for 7 stocks

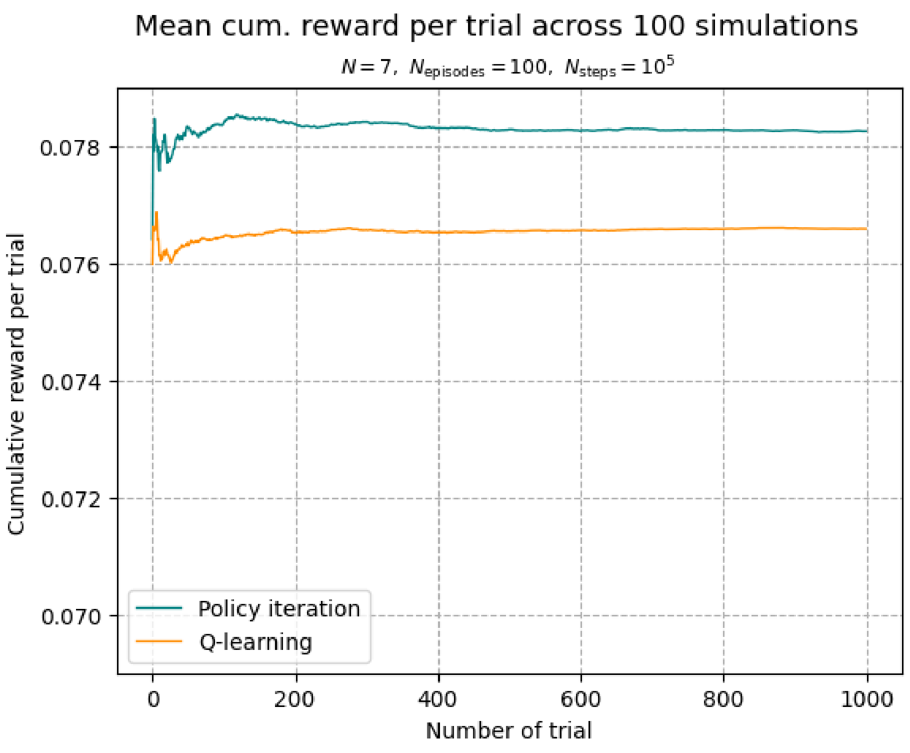

Figure 7: A closer Q-learning approximation, for 7 stocks

All results discussed above are gathered in the following tables:

N (stocks)	**2**	**4**
episodes	100	
steps	100	10^3
policies similarity	100%	80%
performances similarity	N/A%	99.76%
Q-learn. runtime (min.)	<1	<1
P.I. runtime (min.)	<1	<1

Table 1: Results for small scenarios

N (stocks)	**6**		**7**	
episodes	100			
steps	10^3	10^4	10^4	10^5
policies similarity	45%	58%	37%	65%
performances similarity	91.85%	96.92%	89.47%	97.87%
Q-learn. runtime (min.)	<1	5	4	46
P.I. runtime (min.)	5		26	

Table 2: Results for large scenarios

Once more, the policies for the large scenario were tested against random policies and outperformed them significantly (fig. 8).

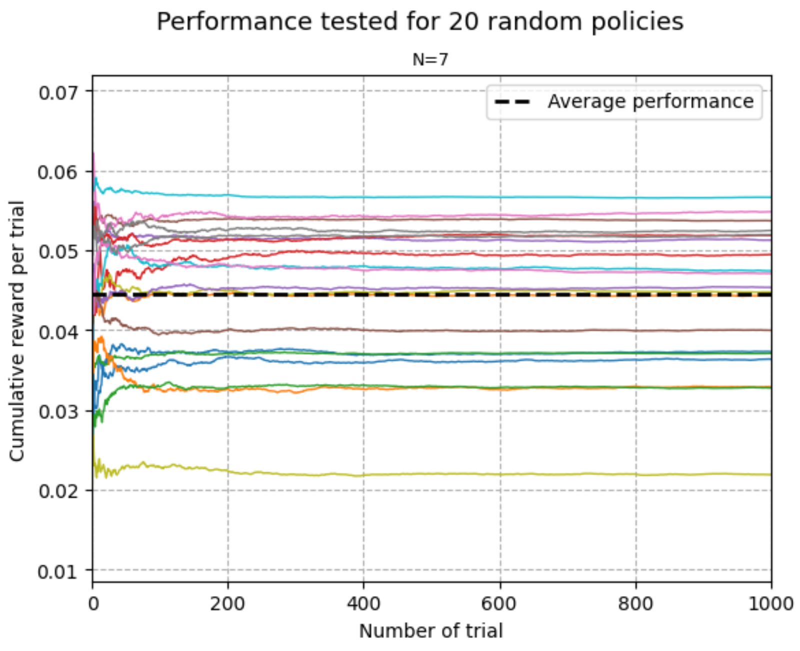

Figure 8: Random policies for $N = 7$.

3. **Deep-Q-Network**
   In this last step, in order to speed-up the convergence to an approximate optimal policy in the larger scenario, Deep Reinforcement Learning was explored and a *Deep-Q-Network* agent with the following architecture was implemented:

   - **Input Layer**: N+1 neurons (state size) that take the state representation of the environment as input.
   - **Hidden Layer 1**: 64 neurons with ReLU activation.
   - **Hidden Layer 2**: 64 neurons with ReLU activation.
   - **Output Layer**: N neurons (action size) with Q-values as the output.

   The input for the DQN is transformed from the traditional states (actions, stock state for each stock) to sequences of simulated "trading days", where each state comprises of the current stock (action) and the current rewards of all stocks. This sequence of states is generated using the transition probabilities matrix.

   Example transformation:
   $$(0,0,1) \implies (0, R_L^0, R_H^1)$$
   Experimentation with the hyperparameters proved to be necessary, as they heavily impacted both termination runtime and performance of the network.
   The exploration probability was now fixed and learning rate was reduced, which helped in stabilizing performance and reduce noise.
   A set of values that produced valuable results in reasonable time is the following:

Parameter	**Value**	**Description**
buffer size	10^4	number of experiences saved for training
batch size	32	number of experiences sampled for one update
$N_{episodes}$	100	number of distinct episodes
N_{steps}	100	steps within each episode
α	0.01	learning rate
ϵ	0.1	exploration probability

   Using these hyperparameters, the DQN converged to a 95.97% performance-wise similar policy to that of Policy Iteration, in only 38s (fig. 9). Comparing the DQN's policy with the random ones shown in fig. 8, one can conclude again that the agent found a non-trivial, dynamic policy.

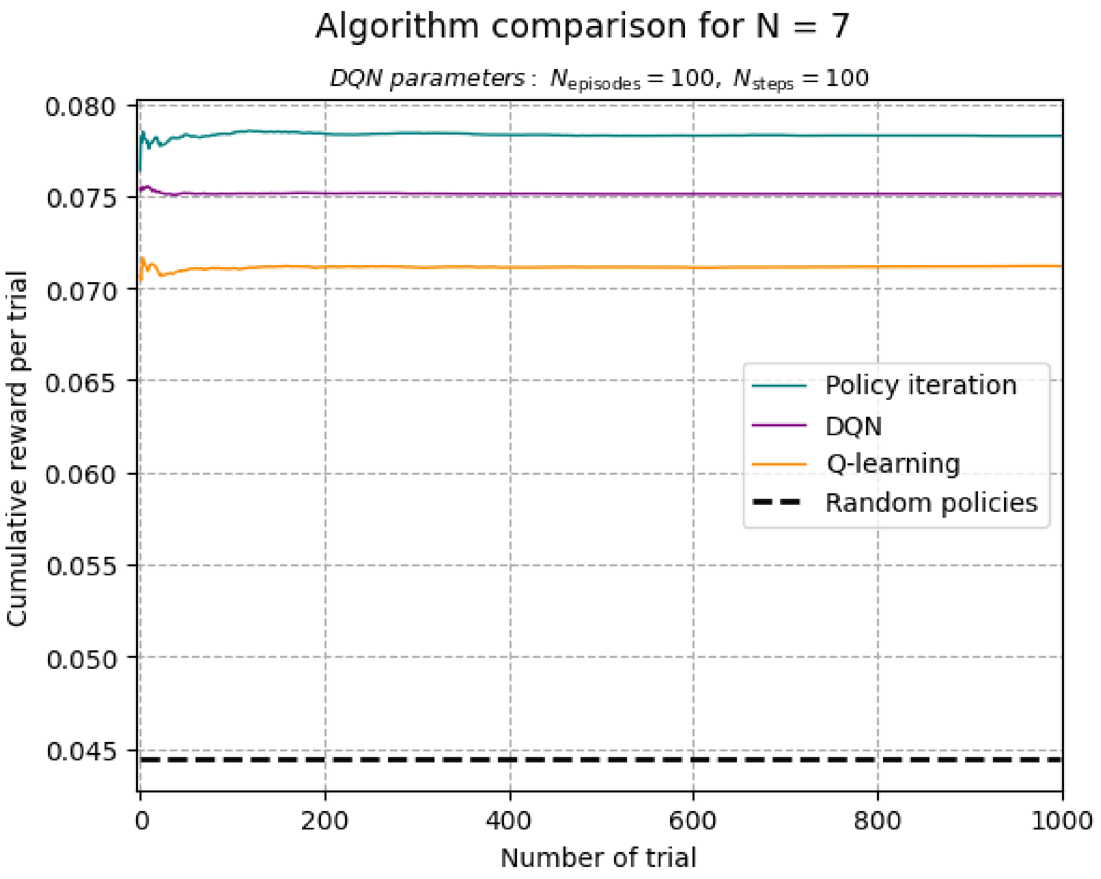

Figure 9: DQN approximation for $t = 38s$ vs. Q-learning approximation for $t = 4min$. Ground truth of P.I. and a random policy baseline are also included for comparison.

As expected, the DQN outperformed Q-learning by achieving a higher approximation of the optimal policy in less time. This superior performance is attributed to the DQN's ability to generalize from past experiences, enabling it to learn more efficiently from large state spaces compared to the tabular approach of Q-learning.