Multi-Agent Reinforcement Learning for Coordinated Block Transportation

Maikel Withagen (S1867733) [a] Steven Bosch (S1861948) [a]
Thomas Derksen (S2016346) [a]

[a] *University of Groningen, Faculty of Mathematics and Natural Sciences, Nijenborgh 9, 9747 AG, Groningen*

Abstract

Multi-agent block transportation is a problem that has been used as a way to compare different machine learning algorithms. It can be complicated because agents have to find a way to cooperatively execute a task in an unknown environment. In this paper, we will use this task to compare the performances of the Q-learning and team Q-learning algorithms. While Q-learning is an algorithm that has previously been very succesful, team Q-learning might be better at tasks where agents have to use coordination. To test the performances of the algorithms, we created a simulation in which two agents have to move a block to a goal area. **ADD SOMETHING ABOUT RESULTS**

1 Introduction

One of the returning issues in machine learning is that it is often hard to decide which learning algorithm should be used for a certain application. Over the years, a lot of valid learning algorithms have emerged, each with their own advantages and disadvantages. Some algorithms might be faster, while others might be more precise, and an algorithm that is better in one area usually performs worse in another. Matching the right task with the right algorithm is not always a straightforward task.

In this paper we will look at the problem of multi-agent box-transportation. For this task, two virtual robots will have to cooperate to transport a box to a certain goal area, while avoiding certain obstacles. This is a problem that has already gathered some attention in machine learning research. In 2006, Wang and de Silva tried to solve this problem with reinforcement learning techniques [4]. They compared the performance of a Q-learning algorithm with the performance of team Q-learning in a dynamic environment. In 2010, a team at the University of Deft also showed how you could use Q-learning, team Q-learning and the WoLF-PHC algorithm to accomplish a similar task in a static environment [1].

We will also compare the Q-learning and team Q-learning algorithm, since they seem most suitable for a task like this. For our initial task, two agents have to transport a box in a two dimensional static environment. We will compare the performance of our two reinforcement learning algorithms by looking at the time it takes for them to converge to the optimal solution (given that they find it). We have created a simulation to do so, which will provide us with the results we need.

2 Application

The box-transportation problem is significant in both the theoretical and practical domain. From a theoretical point of view, it is a relatively simple problem with clear boundaries. Although there are multiple versions of this problem, the basic idea is intuitively easy to understand. Because of its simplicity, it is not hard to extend it to more complicated, real-life situations. It is also a good task to observe the effects of different learning algorithms since, in the more simple, stripped down versions of this task, there are not a lot of parameters that influence the outcome. Finally, it is an interesting way of studying cooperation between multiple autonomous agents.

Although we use a simulation in this paper, the task has successfully been completed by physical robots. For example, a team of two Pioneer 2 mobile robots was used to physically push a box to a goal area while a third robot served as an observer[2]. In kube1996 they took a different approach, where they used a swarm of smaller custom robots to collectively move a box to a designated spot (as can be seen in figure 1). It is not hard to come up with practical applications for these kind of systems. In the future, they might for example be useful in construction, or to clear the rubble after a building has collapsed.

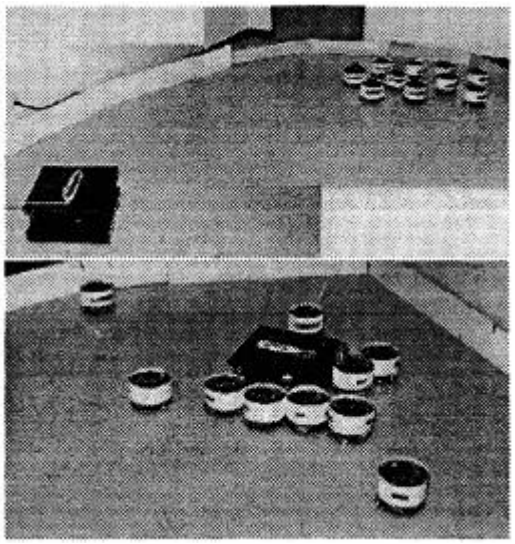


Figure 1: Robots from [3] locating a lit box.

3 Method

To compare the various learning algorithms, we have written a multi-agent simulation in Python in which agents have to move an object to a certain goal area. In this section, the various components of the simulation will be explained.

3.1 The Environment

The environment consists of an **x by y** grid in which the agents can move around. There are a number of different types of cells:

Free cells These are cells that agents can freely move to. Once an agent or a block moves to a free cell, this cell becomes occupied. They are denoted by dots in the visual representation of the simulation.

Walls These cells are occupied, so an agent can never move to them. They are initialized at the start of the simulation and will not change during the simulation. There are outer walls, which are on the edges of the grid, and some walls inside the grid to make the environment more complex. They look like hash tags in the visualization.

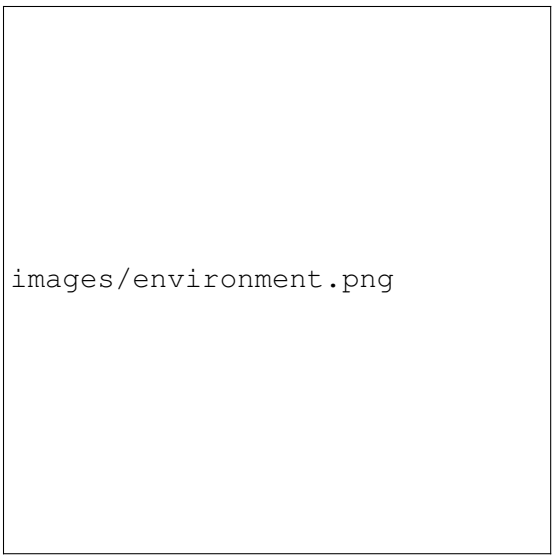
Block The block is an item that has to be transported to a certain goal area by the agents. It occupies one cell, and agents cannot move through it. It can only be moved if it is grasped by

all of the agents present in the simulation, and those agents move in the same direction. If the block reaches the goal, an epoch ends. In the simulation, it is denoted by a B.

Goal The goal cell is basically the same as a free cell, with the exception that if the block reaches the goal a reward is given to the agents and the run of the simulation ends. Agents can freely move on and over the goal, just like with free cells. The goal is represented by a B in the simulation.

Agent We'll talk about these in more detail in the next subsection. They occupy one cell, so other agents cannot move through each other. In the simulation they are represented by A's. In figure 2 you can see what the visual representation of the simulation looks like.

Figure 2: The initial state of the simulation. The *A*'s are agents, while the *B* is the block



3.2 The Agents

For our initial simulation, we start out with two agents. At the start of a run, the agents are spawned at their respective start locations which are defined in the algorithm. Each step, an agent can perform one of five actions: **Stay**, **Left**, **Right**, **Up**, **Down** or **Grab**. The actions mostly speak for themselves. The agent can stay where it is, move in one of four directions or grab the block. The *grab* action can be performed at any step, but will only do something when an agent is next to the block. Once an agent has grabbed the block it will not let go. The move actions *left, up, right* and *down* only do something when the cell the agent wants to move to is actually free. So, for an action to be performed successfully, the algorithm has to check a few things: do the agents move into a space that is occupied? Have the agents grasped the block? And if they have grasped the block, are they moving in the same direction? Each step, an agent chooses the best action based on its learning algorithm. Then, the agents perform their actions and update the values related to their learning algorithms (usually these are the values in their Q-tables). This process repeats itself until the block has reached the goal. After that, a new run starts and the agents are reset to their starting positions. There are two ways for an agent to get a reward: they can either grab the block to get a reward of 10, or they can get the block to the goal for a reward of 100. These rewards are used to calculate the Q-values, which we will talk about more in the next section.

3.3 The Learning Algorithms

For this paper, we compared two machine learning algorithms: Q-learning and team Q-learning. Both are reinforcement learning algorithms, and they are based on the same principles. The goal of both algorithms is, in our case, to make sure that the agents find the optimal policy that will result in the highest reward. It can be found through a value iteration method.

Q-learning was first introduced by Watkins in 1989 [5]. It is a model free reinforcement learning technique for multi-agent systems. In our simulation, the agents begin with a Q-table filled with zeros. Each step, they choose one of the five actions with certain probabilities, which are based on the different Q-values. The probabilities are calculated as follows:

$$P(a_k) = \frac{e^{Q(s,a_k)/\tau}}{\sum_{l=1}^{m} e^{Q(s,a_l)/\tau}} \tag{1}$$

In this case, s is the current state, m is the total number of Q-values for the current state, a_k is the k^{th} action and τ is a parameter that gradually decreases during the simulation. After an action is chosen and a new state s' is reached, the value in $Q(a_k)$ is updated as follows:

$$Q(s, a_k) = (1 - \varepsilon)Q(s, a_k) + \varepsilon(R[s, a_k, s'] + \gamma \max_{a'} Q[s', a']) \tag{2}$$

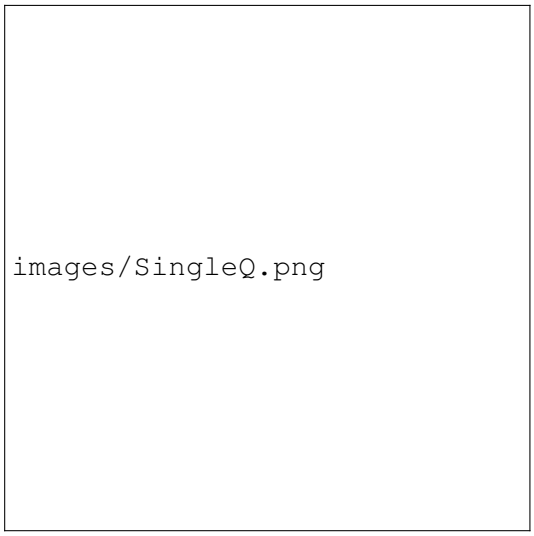
(where ε is the learning rate between 0 and 1). From this formula, it becomes apparent that in order to calculate the Q-value for a certain action in a certain state, both the outcome of that action and the outcome of the best possible action in the next state are considered. After each epoch, to make sure the algorithm converges, we update τ: $\tau_i = \tau_i - (\tau_1 - 0.1/N)$, where τ_i is the value of τ-value in epoch i and N is the total number of epochs.

After the agents have reached the goal with the block, their Q-tables are transferred to the next run. This should result in the fact that the agents learn to choose the path that gains them the highest reward. One notable thing in our simulation is that the agents have two separate Q-tables: one for when the agents have grasped the block, and one for when they have not. This is necessary because the best actions change when the agents start moving the block. Thus, they have to optimize two seperate paths.

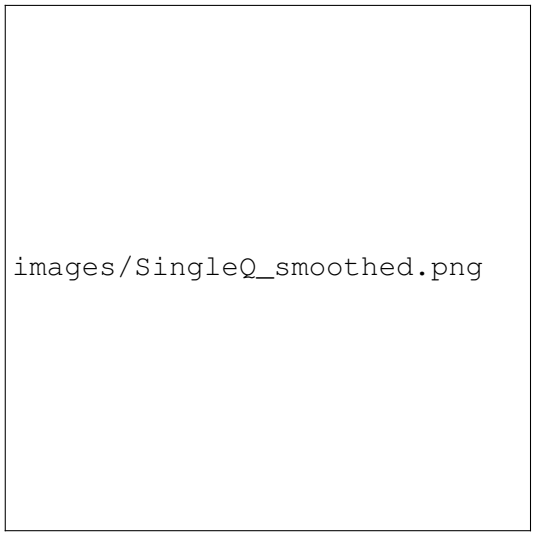
Team Q-learning is based on the same principles as Q-learning, and is actually an extension of the Nash Q-learning algorithm. One of its drawbacks is that it does not have a very good theoretical foundation. The primary difference between the two algorithms is that in team Q-learning, the agents share a single Q-table, and the Q-values are based on their joint actions. In this case, each step the best action pair is chosen based on the current position of both of the agents. So, after both agents have taken an action, the Q-value for the joint position and joint action of the agents is still updated with the same formula, but in this case, $Q(s, a_k)$ is the Q-value for the shared action a_k in the combined state s. The probability that a certain action is taken is calculated in a way that is very similar to Q-learning. The main difference is that the probabilities for shared actions are calculated, instead of probabilities for the actions of the individual agents. So, in team Q-learning the agents share a Q-table and action probabilities. With this algorithm, it is like the agents try to cooperate with each other.

4 Results

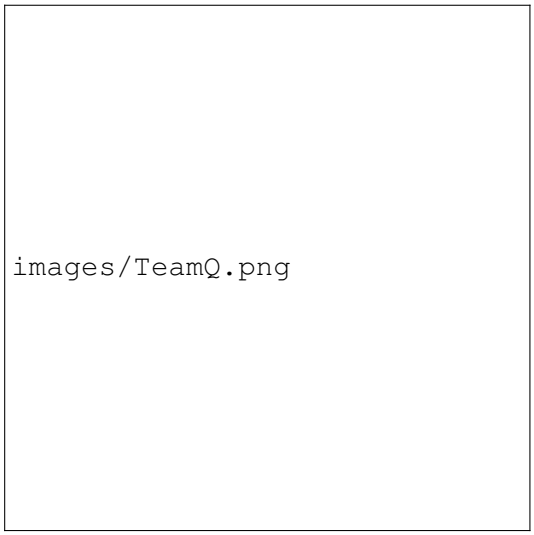
Figure 3 shows the number of steps taken per epoch by two agents using single or team agent Q-learning for a run of 3000 epochs. Each plot shows a graph of the number of steps it takes for the agents to reach and grab the block and a graph of the number of steps it takes them to move the block to the goal. We measured the difference in performance between the two Q-learning



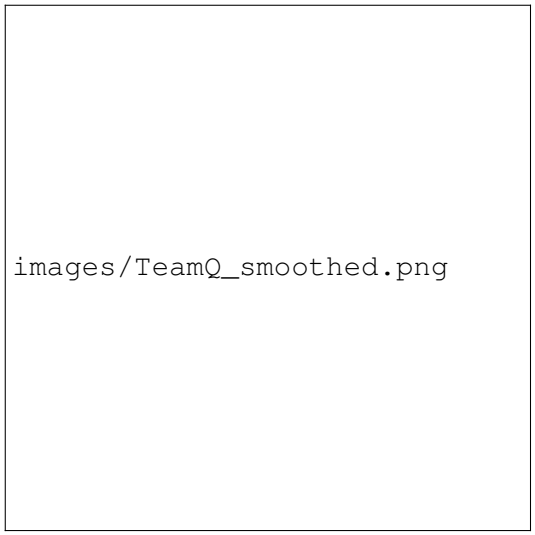
(a) Single Q



(b) Single Q (smoothed)



(c) Team Q



(d) Team Q (smoothed)

Figure 3: Number of steps over 3000 epochs

algorithms by both the speed at which the amount of steps is approximately converged, and the optimality of the learned solution.

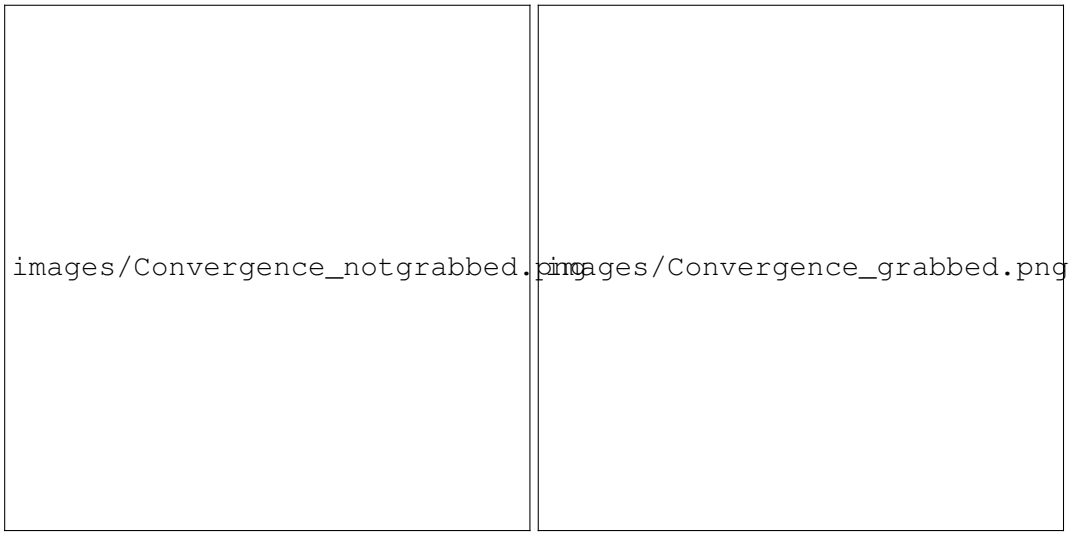
4.1 Speed of convergence

To measure the speed of convergence we used as a convergence criterion that the standard deviation of 20 consecutive epochs be smaller than a set value.[1] Figures 4a and 4b show the standard deviations for both the single and team learning algorithm. The simulation was run 30 times per learning algorithm and every run the number of epochs it took to reach the convergence criterion was counted. Figures 4c and 4d show the results of these counts per condition (grabbed or not grabbed).

To test whether the difference was significant we performed an unpaired t-test on the not grabbed, grabbed and summed results. Given a 95 percent confidence interval the t-test tested

[1] For these results $\sigma_{notgrabbed} < 7$ and $\sigma_{grabbed} < 2$.

(a) Single Q (b) Team Q



(c) Not grabbed (d) Grabbed

Figure 4: (a and b) Standard deviation over 20 epochs in a run of 3000 epochs. (c and d) Epochs before convergence.

significant for the not grabbed samples with $m_{single} \approx 2609$ and $m_{team} \approx 2512$: $t(44.223) = 2.8123$, $p < 0.05$. It tested highly significant for the grabbed samples, with $m_{single} = 1772$ and $m_{team} = 290.4$: $t(52.353) = 14.032$, $p < 0.001$. Finally, the t-test tested highly significant for the added results, with $m_{single} \approx 4381$ and $m_{team} \approx 2802$: $t(49.721) = 14.672$, $p < 0.001$.

From these results we can conclude that single Q-learning is converged slightly faster when the block is not grabbed, but considerably slower when the block is grabbed. In total this results in the team Q-learning being quite faster overall, with an average difference of approximately 1597 epochs. Note that this difference is taken over the sum of the epochs for grabbed and not grabbed samples. Since the algorithms do both the grabbed and not grabbed part during one epoch, you might also consider just the difference between the samples with the highest number of epochs, which in both cases is the not grabbed one. This difference was still measured highly significant, but is much smaller: ± 97 epochs.

(a) Not grabbed (b) Grabbed

Figure 5: Average number of time steps taken for the agents to solve the problem.

4.2 Solution optimality

To determine how good the found solutions were, we calculated the average number of steps taken in the final twenty epochs that showed a standard deviation under the threshold value, as discussed in the previous section. Figure 5 shows the average number of steps taken in the grabbed and non grabbed scenario by both of the Q-learning algorithms.

Again we performed an unpaired t-test on these results. This tested highly significant for the not grabbed samples with $m_{single} \approx 31.9$ and $m_{team} \approx 38.1$: $t(56.961) = -5.2118$, $p < 0.001$. For the grabbed samples the test did not test significantly with $m_{single} \approx 8.66$ and $m_{team} \approx 8.99$: $t(57.428) = -1.9165$, $p > 0.05$. Finally for the total number of steps the test tested highly significant with $m_{single} \approx 40.6$ and $m_{team} \approx 47.1$: $t(56.165) = -5.4113$, $p < 0.001$.

These results show that, within the number of epochs in which the algorithms converge to below the predefined threshold, on average the single agent Q-learning algorithm finds the better solution for the first stage of the simulation in which the agents have to find the block and grab it. for the second stage, in which the agents have to move the block, there was no significant difference between the two algorithms. Overall however, because of the difference in the first stage, the single agent algorithm performed significantly better: the agents finished the simulation in approximately 6.5 steps less than with the team algorithm.

5 Discussion

Looking at the results we can observe that team Q-learning converged too a solution faster than single Q-learning. This was the case in the first stage of the simulation. On the other hand, the single Q-learning solution turned out more optimal than its counterpart, with on average 6.5 less steps than the team Q-learning solution. These are however, the overall results of the simulation. It is easier and more insightful to discuss the more specific phenomena we encountered in the results.

First, let us look into the two stages of the simulation separately. Before the block has been grabbed we see little difference between both of the algorithms in the average number of

epochs it takes before convergence (2609 vs. 2512), but we do see a significant difference in the average number of steps taken in the final solution (31.9 vs. 38.1). This difference is quite considerable. 6.5 Steps on a total of ± 35.15 steps comes down to approximately 18.5%. If the state space would be of a higher order, this could grow to become a considerable drawback to using this algorithm. The reason that team Q-learning gives a significantly less optimal solution might be that **REASONS!!!!!!**

In the second stage of the simulation, we see a different phenomenon. Instead of a difference in solution (this turned out insignificant), we see a difference in convergence speed, and quite a considerable one. Where it took single Q-learning on average 1772 epochs, team Q-learning finished in 290 epochs. This comes down to a factor ± 6.1 faster convergence. This is most probably due to the fact that... **REASONS!!!!**

Second, the progression of the standard deviations (figures 4a and 4b) show interesting phenomena as well.

Finally, let us look at the curves of the epochs themselves (figure 3)

References

[1] Lucian Busoniu, Robert Babuska, and Bart de Schutter. Multi-agent reinforcement learning: An overview. In Dipti Srinivasan and Lakhmi Jain, editors, *Innovations in Multi-Agent Systems and Applications*, chapter 1, pages 183–221. Springer Berlin Heidelberg, Berlin, 2010.

[2] Brian P. Gerkey and Maja J. Mataric. Pusher-watcher: an approach to fault-tolerant tightly-coupled robot coordination. In *International Conference on Robotics and Automation*, pages 464–469, 2002.

[3] C. Ronald Kube and Hon Zhang. The use of perceptual cues in multi-robot box-pushing. In *International Conference on Robotics and Automation*, pages 2085 – 2090, 1996.

[4] Ying Wang and Clarence de Silva. Multi-robot box-pushing: Single-agent q-learning vs. team q-learning. In *International Conference on Intelligent Robots and Systems*, 2006.

[5] Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, 1989.