Uppsala University

Natural Computation Methods for Machine Learning

1DL073

Coordinated Resource Collection

Deep Q-Networks for centralized cooperative multi-agent reinforcement learning

Anders Köhler

Emil Grund Stålvinge

Julián Ramón Marrades Furquet

June 15, 2021

Abstract

We introduce a spatial multi-agent in which cooperation is essential to success. Within a map, several agents must collect resources while coordinating in order to overcome obstacles and minimize punishments. Even though converting a multi-agent reinforcement learning problem into a into a single-agent one may backfire as the state-space grows and the number of agents increases, we show that it is possible to train a multi-agent system by means of reinforcement learning with a single centralized Deep Q-Network when the environment complexity is low. We divide our analysis in two fronts. Firstly, we train and test the Deep Q-Network quantitatively. Secondly, we examine the resulting agents' behavior and how they exploit a particular game feature. Finally, we provide some directions for future research.

1 Introduction

Reinforcement learning (RL) has been around for more than half a century as a subfield within the realm of artificial intelligence (AI). Over the years, computers have evolved and so too have RL algorithms, which can now engage much more complicated tasks than simply playing Tic-Tac-Toe. Nonetheless, regardless of the complexity of an RL algorithm, the underlying notion is always the same: rewarding an agent when it performs well and punishing it when it does not, i.e. behaviorism. It is often surprising how such a simple rule can produce AIs with skill sets that not even human experts can dream of attaining. For instance, see [17].

However, creating superhuman GO players is not the only strength of RL. Algorithms have been evolved to handle multi-agent systems (MAS). That is, environments where many competing and/or cooperating learners are present: multiplayer games, autonomous driving, company warehouse robots, etc. As one can imagine, the possibilities are immense, but so are the challenges, as we discuss in Section 2.1. In this paper, we introduce a spatial MAS in which cooperation is essential to success. Within a map, several agents must collect resources while coordinating in order to overcome obstacles and minimize punishments. Then, we employ a naive approach to train agents in several environment versions of increasing complexity to study the resulting collective behavior.

Hereinafter, this paper is organized as follows. Section 2 covers some background material which is essential for the presented work. Section 3 provides an explanation of the proposed MAS. Section 4 showcases several publications regarding cooperative multi-agent reinforcement learning (MARL) via Q-learning and/or deep learning. Section 5 accounts for the computer implementation of the environment and the experiment design. Section 6 presents and discusses the experiment results. Finally, section 7 wraps up the paper and indicates possible directions for further investigation.

2 Background

2.1 Multi-agent reinforcement learning (MARL)

In classical single-agent RL (SARL) problem, given a set of states S, a set of actions A, a transition function $\delta : S \times A \to S$, and a reward function $\mathcal{R} : S \times A \to \mathbb{R}$, with the latter two being unknown to the agent, a policy π must be found so that

$$V^{\pi}(s_t) := \sum_{i=0}^{\infty} \gamma^i r_{t+i} \tag{1}$$

is maximized, where s_t denotes the state at time t, γ is the discount factor, and r_{t+i} is the reward at time $t+i$.

A simple way to extend SARL algorithms to handle MAS is to let each agent be an independent learner, considering other agents as dynamic components of its environment (see [19]). However, this introduces non-stationarity [4, 15], since a given state-action pair for a single agent may lead to several states. Usually, in MAS, an agent should also be interested in other agents. Where are they? What are they doing? What is their plan? While classical RL algorithms attempt to figure out which actions in a given sequence led to a reward, MARL algorithms try to learn which actions of which agents led to a reward. As one might expect, this increases the level of complexity. If m denotes the number of agents, then:

- The state space grows exponentially with m.
- Let A_k be the set of actions of agent k. The set of possible actions (A) grows exponentially with m: $A := A_1 \times A_2 \times \cdots \times A_m$.
- The space of joint-action outcomes grows exponentially with m: $\delta : S \times A_1 \times A_2 \times \cdots \times A_m \to S$.
- Rewards can be shared or individual: $\mathcal{R} : S \times A_1 \times A_2 \times \cdots \times A_m \to \mathbb{R}$ or $\mathbb{R}^m$.

2.2 Deep Q-Networks (DQNs)

When the state space becomes too large to handle in table form one can instead approximate the Q-values as a function. For instance, with a neural network (NN), which in this case takes the name of Deep Q-Network (DQN), seen in Fig. 1. One should note that since the purpose of the DQN is to approximate a function, the activation function of its output layer should be linear. In this paper, we employ DQN agents, which are learners that employ DQNs for Q-value approximation.

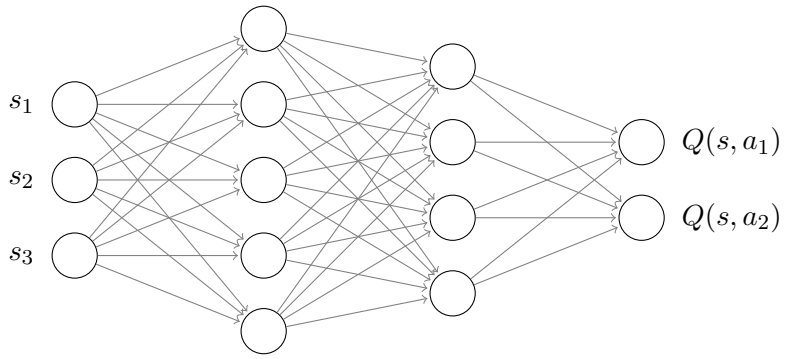


Figure 1: DQN with 3 input variables as state observations, 2 outputs which each correspond to the Q-value of one action, as well as 5 and 4 nodes in its first and second hidden layer, respectively.

2.2.1 Q-value and loss function

For a given state-action pair (s, a), its Q-value should converge to

$$Q_*(s, a) = \mathbf{E}\left[r_{t+1} + \gamma \max_{a'} Q(s', a')\right], \tag{2}$$

where $\mathbf{E}[\cdot]$ denotes expectation. In order to calculate the Q-value of the right hand side, one would normally simply consult the Q-table. However, with the DQN as a function approximator, we rely

on it to compute both $Q_*(s,a)$ and $Q(s,a)$, as explained in section 2.2.2.

To compute the loss function L of the DQN, the current Q-value of a state-action pair $Q(s,a)$ is subtracted from the desired Q-value $Q_*(s,a)$, whereupon the resulting loss is the squared value of this difference (i.e. the mean squared error). Formally, this is:

$$L(Q_*, Q, s, a) = (Q_*(s,a) - Q(s,a))^2 = \left(\mathbf{E}\left[r_{t+1} + \gamma \max_a Q(s', a') \right] - \mathbf{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \right] \right)^2 . \quad (3)$$

This operation is carried out in batches, however, entailing that the loss be averaged over them. With a batch size of N, this gives us:

$$L(Q_*, Q, s, a) = \frac{1}{N} \sum_{i=1}^{N} \left(Q_*(s,a) - Q(s,a) \right)^2 . \quad (4)$$

The result of minimizing the loss is a network configuration which outputs Q-values for each state-action pair that are as close as possible to the target Q-values given by Eq. 2. In so doing, we have a DQN that provides us with an approximated optimal policy.

2.2.2 Target update and experience replay (ER)

When training a DQN agent, two networks are needed: a policy network, i.e. the one being trained, and a target network. This is because if only one network is used for both the Q-value output $Q(s,a)$ and the Q-value target $Q_*(s,a)$, they will be strongly correlated. The outputs will move towards the targets, with the target values moving in the same direction, much like a dog chasing its own tail, due to s and s' being computed from the same weights, with only one timestep in between. To solve this we need a fixed target to move towards, but we also need to periodically update this target to ensure that we are not moving towards outdated values.

As such, the policy network is copied and "frozen" for a period of time, thereby becoming the target network. The active policy network is then trained with the target network dictating the target values $Q_*(s,a)$ for updating the weights. Any correlation that initially exists thus swiftly vanishes. If a hard update policy is employed, the policy network is "refrozen" at certain time intervals and completely replaces the old target network. A soft update policy, on the other hand, entails that the target network is *updated* at every timestep according to

$$N_* := N \cdot \texttt{target\_model\_update} + (1 - \texttt{target\_model\_update}) \cdot N_*, \quad (5)$$

where N_* is the set of weights in the target network, N is the set of weights in the policy network, and $\texttt{target\_model\_update} \in [0, 1]$ dictates the extent to which the policy network affects the update. Simply put, the updated target network is a linear combination of its predecessor's weights and those of the current policy network.

To further decrease correlation and allow for improved training, we utilize experience replay (ER). Instead of updating the Q-values based on the most recent simulation steps, each step is saved in a dedicated memory as an experience $e_t = (s_t, a_t, r_t, s_{t+1})$, whereupon experiences are randomly chosen from memory during Q-value updates. As seen in [13], where DQNs are applied to different games, fixed-Q targets (hard update policy) combined with ER offer superior performance.

In summary, the DQN training regime employed in this paper is accounted for in algorithm 1 below.

Algorithm 1 DQN training regime

```
Initialize policy network
Initialize target network (copy of policy network)
Initialize replay memory
for episode = 1, 2, ..., max_episodes do
    Initialize starting state s_0
    for step = 1, 2, ..., max_steps do
        Select action a in current state s
        Perform action, leading to s' and r
        Store s, a, r, s' in replay memory
        Sample random batch from replay memory
        Pass batch through policy network (output)
        Pass batch of next state through target network (targets)
        Calculate loss
        Perform a gradient descent iteration on the policy network
        if a certain interval of steps has passed then
            Update target network with weights from policy network
        end if
    end for
end for
```

3 The resource extraction game

3.1 Overview

In accordance with fundamental considerations of complexity theory, we seek to employ a MAS which may allow for the agents to develop complex behaviors as an emerging feature of simple world rules. Additionally, it is desirable that the agent/world interactions be meaningfully mapped to some form of real-world phenomena, allowing for contextualization and a broader range of data interpretation. Thus, we propose a resource extraction game for the purpose of training agents to find a sustainable policy of resource collection under circumstances where the process is met with some form of resistance. In this paper, we assume that the agents represent authority figures operating on behalf of some state apparatus. The state's goal is to extract as many resources as possible from a population while maintaining societal order, measured by the absence of civil unrest/riots.

Consider a two-dimensional ontology populated by three types of inhabitants: enforcers, resources and civilians. The enforcers, representing the state – which may be thought to exist conceptually "in the background" as a fourth inhabitant, or simply embodied by the collective of enforcers – operate with the one goal of extracting resources. This extraction process is initiated by means of an enforcer approaching a resource, whereupon work required to carry out the process is performed by civilians located within the region surrounding the resource. This is the resource collection aspect mentioned above. The civilians, however, do not appreciate being subjected to such master/slave dynamics, and may express their opposition through the act of rioting, if not controlled by the enforcers. This is the resistance aspect. The policy that the enforcers seek to adopt answers the question "how can extraction value be maximized while keeping riots to a minimum?"

The scenario, then, may be summarized as such:

- State-loyal enforcers move to locations of resources and initiate extraction.
- Civilians in the vicinity of the extracted resource are subjected to forced labor.
- The extraction process generates income for the state but may entail civilian riots.
- The enforcers wish to maximize income value while avoiding/suppressing riots.

Apart from the sociological value in modeling societal dynamics that have played a defining role throughout large sections of human history, such as feudal societies, the model setting is of interest as it may be equally applicable to real-world scenarios that are not necessarily characterized by such explicitly oppressive relations. For instance, in a business setting, the civilians may represent workers or departments in a company. The enforcers would then constitute some force that initiates work in, or draws resources from, a certain sector, but which also serves to relieve overburdened workers/departments, hence a generalized notion of "riot control".

3.2 Computational specifics

We employ a rectangular grid as our computational domain. Inside it, there are n civilians positioned statically at random locations. Initially, c resources are spawned at arbitrary locations of the grid. Once a resource is extracted, it will either respawn at a random location or simply disappear from the map until a reset is carried out, depending on implementation version (see section 5). There are m moving agents – enforcers – whose aim is to extract the resources. Note that it is possible for all unit types to overlap at any one position. A visualization example is given in Fig. 2.

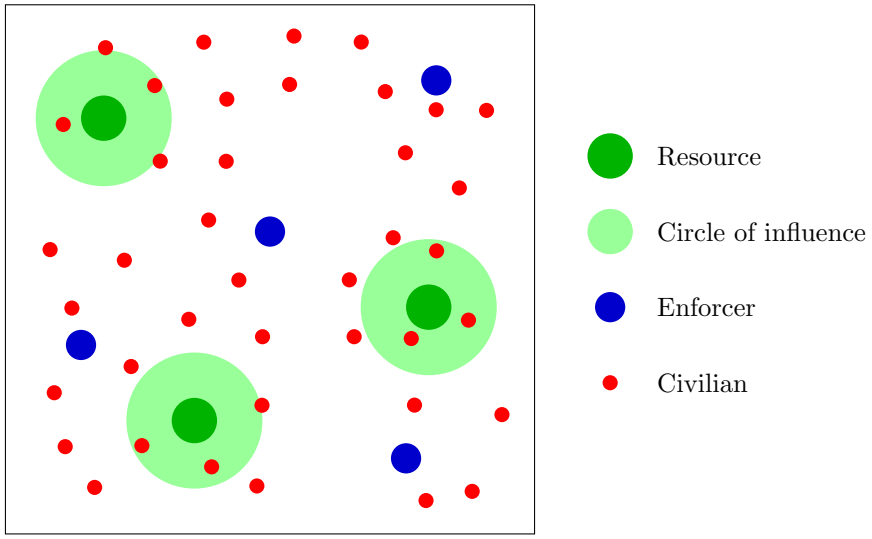


Figure 2: Concept art of a resource extraction game setup. There are $m = 4$ enforcers, $c = 3$ resources with their respective circles of influence (see below), as well as $n = 40$ civilians.

If an enforcer moves on top of a resource, it will be extracted and removed. When that happens, we depict a circle centered at the extracted resource with radius I, which we call *circle of influence*. Let N_C be the set of civilians within the circle of influence of resource C, and M_C the set of enforcers

within it. As a response to the extraction process, civilians in N_C will riot unless controlled by an enforcer in M_C. It is defined that an enforcer can control p rioting civilians within the circle of influence. The reward r for that extraction is then computed according to:

$$r(s,a) = r_d + \sum_{i=1}^{c} f(C_i) \cdot (r_e + r_o)\,, \tag{6}$$

$$f(C_i) = \begin{cases} 1, & \text{if } C_i \text{ is extracted in the current step,} \\ 0, & \text{otherwise,} \end{cases} \tag{7}$$

$$r_o = \begin{cases} \alpha\,(p \cdot M_{C_i} - N_{C_i})\,, & \text{if } p \cdot M_{C_i} \leq N_{C_i}, \\ \beta\,(p \cdot M_{C_i} - N_{C_i})\,, & \text{otherwise,} \end{cases} \qquad \beta << \alpha, \tag{8}$$

where $\alpha, \beta \in \mathbb{R}^+$, r_d denotes the default reward component at every timestep, r_e is the base reward value for an extraction, and $p \cdot M_{C_i} - N_{C_i}$ stands for the number of people who riot within the circle of influence of the i-th resource (the o in r_o denotes "order").

As Eq. 8 shows, the agents are punished for the act of performing an extraction under circumstances where insufficient control is exercised over rioting civilians, determined by the α parameter. However, if an extraction is overcontrolled, meaning that more enforcers than necessary are present in the circle of influence, this entails a positive bonus being added to the reward, determined by the β parameter. In order not to overincentivize the agents to rely on fully controlled extractions and thus move as one single cluster – thereby hardly accomplishing complex emergent behavior – it is likely necessary that β be considerably smaller than α.

If several extractions occur simultaneously, their rewards are added up and given to the agents. Note that since we want cooperation, each agent should receive the same reward. The environment is fully observable, i.e. each agent knows the position of all enforcers, resources and civilians.

4 Related work

Over recent years, there have been many contributions towards cooperative MARL with Q-learning and/or deep learning. Such publications have drifted apart from central control – which is what we employ in this paper – towards distributed duties in order to cope with the complexities of MAS.

Abdoos, Mozayani and Bazzan [1] employ multi-agent Q-learning for traffic light control in which each agent controls an intersection. Then, given a queue length on approaching roads, agents select a regime; i.e. green and red light timings. Liu, Liu and Chen [10] expand this approach, with agents accounting for neighboring intersections when selecting an action. Chu et al. [3] evolve the previous approaches into decentralized actor-critic deep MARL, which is claimed to be vastly scalable.

Egorov [5] uses a convolutional neural network (CNN) to extract features from an image-like state and output estimated Q-values for a given agent. Training is performed one agent at a time, keeping the policies of all other agents fixed. If there are types of agents, a given agent will distribute its policy to all other agents of its kind after its training period, speeding up the learning process significantly. Execution is still distributed, since each agent has its own CNN/DQN controller.

Lin et al. [9] propose a contextual MARL framework which handles large-scale fleet management. They propose two algorithms: contextual deep Q-learning and contextual multi-agent actor-critic,

which achieve efficient explicit coordination among thousands of agents and can adapt to variable action-spaces.

Lowe et al. [11] present a training procedure employing a critic per agent which can access all agent observations, actions and policies. All information is then concatenated as input and Q-values are computed by looking at the local rewards.

Foerster et al. [6] propose a single centralized critic which employs the global state, an aggregated action vector, and a joint reward. However, each agent has its own actor which computes the contribution of the agent to the reward.

Yang et al. [20] present a cooperative environment in which the agents also have individual goals. Their training regime takes place sequentially. Firstly, each agent is trained with a single-agent algorithm to achieve its personal goal. Then, a centralized critic is employed to train for the global task.

Jiang et al. [7] propose a graph-based approach in which each agent observation is encoded into a feature vector. Then, local observations of neighbors are combined into a latent features. Finally, such latent features are merged into a final vector, which is fed into a DQN.

5 Methodology

5.1 Implementation

The computational domain is implemented as a square grid of size g, with distances computed via chessboard distance $D = \max(|x_2 - x_1|, |y_2 - y_1|)$. The environment itself is built on top of Open AI Gym [2] and displayed to the user with `pygame` [16]. It is important to note that the environment instance is given a seed for the random number generator for reproducibility reasons.

In every iteration, each enforcer can either move to a neighboring square or not move at all. Neighboring squares are the immediate neighbors in the vertical and horizontal directions. Moving outside of the boundaries is not allowed. That is, trying to move outside of the domain will leave the agent's location unchanged. Actions are encoded in an action vector

$$a = [a_1, a_2, \ldots, a_m], \tag{9}$$

where a_i represents the action of the i-th agent. Instead of implementing all the features at once, the development is incremental: as the version increases, so does the complexity.

5.1.1 Versions 1.0 and 1.5

In these versions, only agents and resources are present (see Fig. 3). Since there are no civilians, the circle of influence is irrelevant. The state vector is built as

$$[C_1^x, C_1^y, C_2^x, C_2^y, \ldots, C_c^x, C_c^y, M_1^x, M_1^y, M_2^x, M_2^y, \ldots, M_m^x, M_m^y], \tag{10}$$

where C_i^x, C_i^y denote the x and y coordinates of the i-th resource, and M_i^x, M_i^y denote the x and y coordinates of the i-th enforcer. The size of the vector is $2(c+m)$. The only difference between the versions is the range of the vector entries; in v1.0, the coordinates are integers in the range $[0, g-1]$, whereas in v1.5 the values are normalized to $[0, 1]$. Once a resource is extracted, another one spawns at an arbitrary position. This feature introduces stochastic components into the environment.

In this version, the reward for a state-action pair is simplified to

$$r(s,a) = r_d + \sum_{i=1}^{c} f(C_i) \cdot r_e, \quad r_e \in \mathbb{R}^+. \tag{11}$$

An episode terminates after T steps.

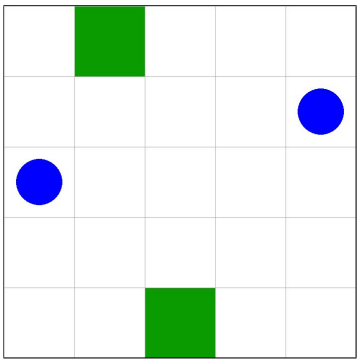


Figure 3: Example frame of versions 1.0 and 1.5 with a 5-by-5 grid. Green squares denote resources and blue circles are enforcers.

5.1.2 Versions 2.0 and 2.5

Now, circles of influence for the resources and civilians are added to the environment (see Fig. 4). Every feature stays the same as in the first set of versions except for the state vectors and reward function. For v2.0, the state vector is built as

$$[C_1^x, C_1^y, C_2^x, C_2^y, \ldots, C_c^x, C_c^y, M_1^x, M_1^y, M_2^x, M_2^y, \ldots, M_m^x, M_m^y, N_1^x, N_1^y, N_2^x, N_2^y, \ldots, N_n^x, N_n^y], \tag{12}$$

where N_i^x, N_i^y denote the x and y coordinates of the i-th civilian. The size of the vector is $2(c+m+n)$. In this version the vector entries are not normalized. For v2.5, the state vector is built as

$$[C_1^x, C_1^y, N_{C_1}, C_2^x, C_2^y, N_{C_2}, \ldots, C_c^x, C_c^y, N_{C_c}, M_1^x, M_1^y, M_2^x, M_2^y, \ldots, M_m^x, M_m^y], \tag{13}$$

where N_{C_i} denotes the amount of civilians present in the circle of influence of the i-th resource. The size of the vector is $3c + 2m$. In this version the vector entries are normalized to $[0, 1]$. The reward function is the same for v2.0 and v2.5, as described in Eq. 6, 7 & 8, and episodes last for T steps.

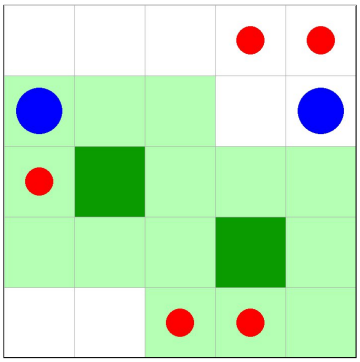


Figure 4: Example frame of versions 2.0 and 2.5 with a 5-by-5 grid. Dark green squares denote resources, light green areas are the circles of influence, red circles are civilians, and blue circles are enforcers.

5.1.3 Versions 3.0 and 3.5

In this last pair of implementations, the resource respawn feature is eliminated. An episode will now terminate once all resources in the grid have been extracted by the agents or after T steps.

As in Section 5.1.2, the state vector also differs between v3.0 and v3.5. For v3.0, the state vector remains the same as for v2.5 (see Eq. 13), including normalization. However, $[C_i^x, C_i^y, N_{C_i}]$ is set to $[-1, -1, 0]$ (becoming $[-1/(g-1), -1/(g-1), 0]$ after normalization) if the i-th resource has been extracted, i.e. it is moved out of bounds. The state vector for v3.5 is

$$[C_1^x, C_1^y, F(C_1), N_{C_1}, \ldots, C_c^x, C_c^y, F(C_c), N_{C_c}, M_1^x, M_1^y, M_2^x, M_2^y, \ldots, M_m^x, M_m^y], \tag{14}$$

$$F(C_i) = \begin{cases} 1, & \text{if } C_i \text{ has been extracted,} \\ 0, & \text{otherwise.} \end{cases} \tag{15}$$

As in v3.0, N_{C_i} is set to 0 if C_i has been extracted. Entries are normalized to $[0, 1]$ in both versions.

5.2 Experiments

We transform the problem into a SARL setting, in which a central controller handles the actions of all the agents. Given that each agent has 5 possible actions, as described in section 5.1, the total amount of possible moves in the system is 5^m. Hence, we let the output layer of the DQN have 5^m neurons and associate each node to an action vector (see Eq. 9).

Then, we train the *master* agent via a DQN as described by Mnih et al. [13, 14] with `keras-rl2` [12] and an Adam optimizer [8] (see parameters in Table 1), since the latter is appropriate for non-stationary objective functions and noisy problems. For experience replay, we make use of `SequentialMemory` with `limit`=50000 and `window_length`=1. Regarding action selection, we employ an ε-greedy policy with $\varepsilon = 0.1$. Each DQN agent operates a 2-hidden-layer DQN with 32 and 16 neurons, respectively. Both hidden layers use the ReLU as their activation function. Other parameters for the DQN agents are depicted in Table 2.

Parameter	`learning_rate`	`beta_1`	`beta_2`	`epsilon`
Value	0.001	0.9	0.999	10^{-7}

Table 1: Adam optimizer parameters.

Parameter	`gamma`	`batch_size`	`nb_steps_warmup`
Value	0.99	32	100
Parameter	`train_interval`	`memory_interval`	`target_model_update`
Value	1	1	0.01

Table 2: DQN training parameters.

Sunehag et al. [18] showed that when transforming a MARL problem into a SARL problem, traditional SARL algorithms may not find the optimal solution since the agents, given a reward r, do not know how much they contributed to it, and could become lazy over time. Nonetheless, we will show our MARL-to-SARL adaptation being successful for small-scale environments.

The conducted experiments can be categorized as quantitative and qualitative. For the first case, we train DQN agents with identical characteristics for v1.0 and v1.5 under the same environment conditions (`seed` = 1) for 5 million steps. Then, we test both agents in their respective environment versions for a set of 10000 episodes (`seed` = 2) and compare their rewards. This procedure is repeated for v2.0 against v2.5, and v3.0 against 3.5. For the qualitative case, we study the observable behavior of the enforcers. One would expect them to (1) group up when large amounts of civilians are located within the resources' circles of influence, to minimize punishment, and (2) operate alone when rewards are isolated from population. The sets of environment parameters are presented in Table 3.

Parameter	g	c	m	n	I	p	r_d	r_e	α	β	T
v1.0 & v1.5	5	2	2	-	-	-	-1	10	-	-	50
v2.0 & v2.5	5	2	2	5	1	1	-1	10	6 to 10	0	50
v3.0 & v3.5	5	2	2	5	1	1	-1	10	6	0	50

Table 3: Environment parameters. For v2.0 and v2.5, we only employ $\alpha = 6$ for quantitative experiments, with $\alpha \in \{6, 7, \ldots, 10\}$ being used for qualitative behavior analysis.

6 Results and discussion

6.1 Quantitative results

Fig. 5 shows the training process of v1.0 and v1.5 as a heat map. One can see that training reward for v1.5 grows faster (left part of the plot) and the process is much more stable. This may be due to the input ranges of the DQN, which in v1.0 span from 0 to $g - 1$, while in v1.5 they are normalized to $[0, 1]$. If one considers the definition of ReLU:

$$\mathrm{ReLU}(x) = \max(0, x), \tag{16}$$

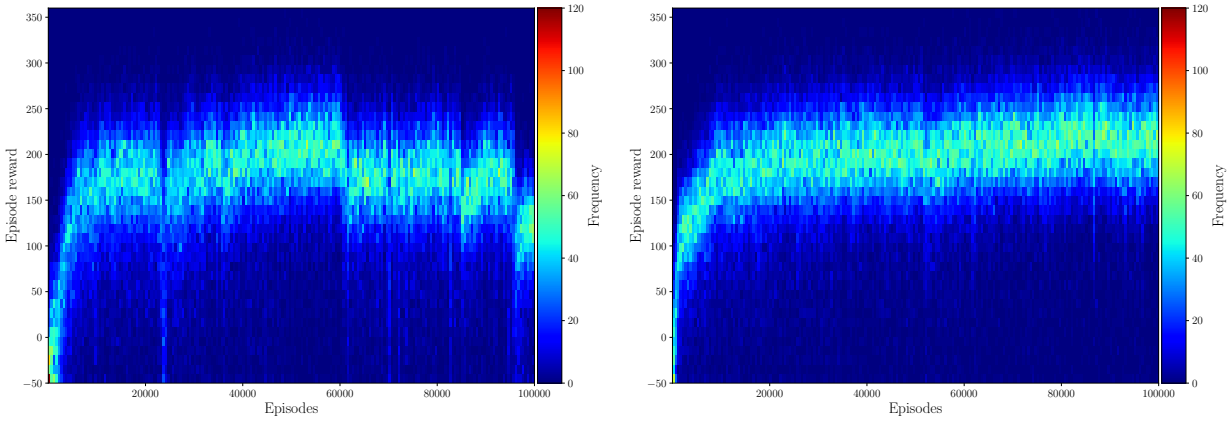
one may notice that large inputs can be the cause of large neuron activations. Such phenomena may lead the network to become really dependent on those nodes, causing a loss of generalization power (overfitting). Since the chances of the same state being presented twice to the agents are almost 0, and since we have resources respawning at arbitrary locations, generalization capabilities are essential for success. Fig. 8a plots the test performance of v1.0 vs v1.5 in a brand new set of episodes. One can see that the performances observed in the plot are of the same nature as the underlying training processes, with v1.5 being significantly better than v1.0.

Fig. 6 depicts the learning process of v2.0 and v2.5. On the one hand, it is clear that our "small" DQN fails to learn any decent policy for v2.0. This is because it is presented with the not-normalized locations of resources, enforcers, and civilians, whereupon it has to learn all the logic by itself, i.e. how many civilians are in each circle of influence, how to move towards resources, when to group up, etc. One would probably need a much larger network and significantly longer training time to make this work. On the other hand, training for v2.5 is successful, with episode rewards not reaching as high as for v1.5 due to the much more punitive nature of the environment. This positive outcome comes from the state vector simplification and input normalization. In v2.5, instead of giving the location of each civilian, we provide the DQN with the amount of civilians close to each circle of influence, since as long as they are within the circle their particular location is irrelevant.

This significantly simplifies the learning for the network, as confirmed by Fig. 6b.

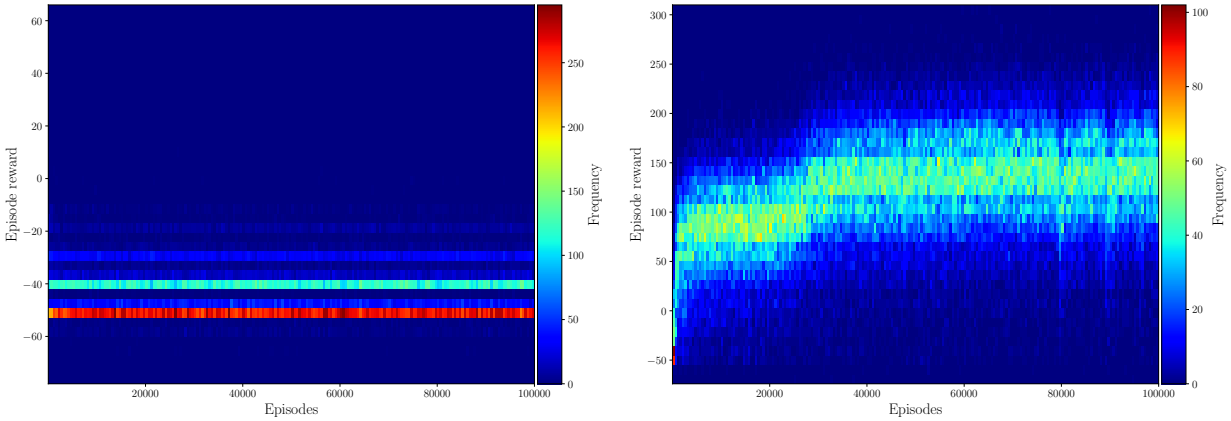
Fig. 8b shows the test outcome of both versions. As expected, v2.0 does not even reach positive episode rewards, while v2.5 successfully handles the presence of civilians. Since they represent an obstacle to resource extraction, comparing Fig. 8a & 8b, one can see that the distribution of episode rewards for v2.5 is flatter than for v1.5 and shifted down (the mean drops by approximately 100).

Fig. 7 aggregates the training of v3.0 and v3.5. Recall that in this new set of versions, resources do not respawn and an episode finishes once all rewards on the grid have been extracted or a certain number of steps have been taken. One should also remember that, in order to indicate that a resource has been extracted, v3.0 sets its coordinates to $[-1, -1]$ while v3.5 gives an extra input per resource with value 1 if the resource has been extracted and 0 otherwise. Both versions show an almost identical training process, with v3.5 being better by a barely noticeable difference. Fig. 8c & 8d show the test results for v3.0 vs v3.5. Looking at their learning process, one would expect their performance in new sets of episodes to be almost identical, as confirmed by the plots.



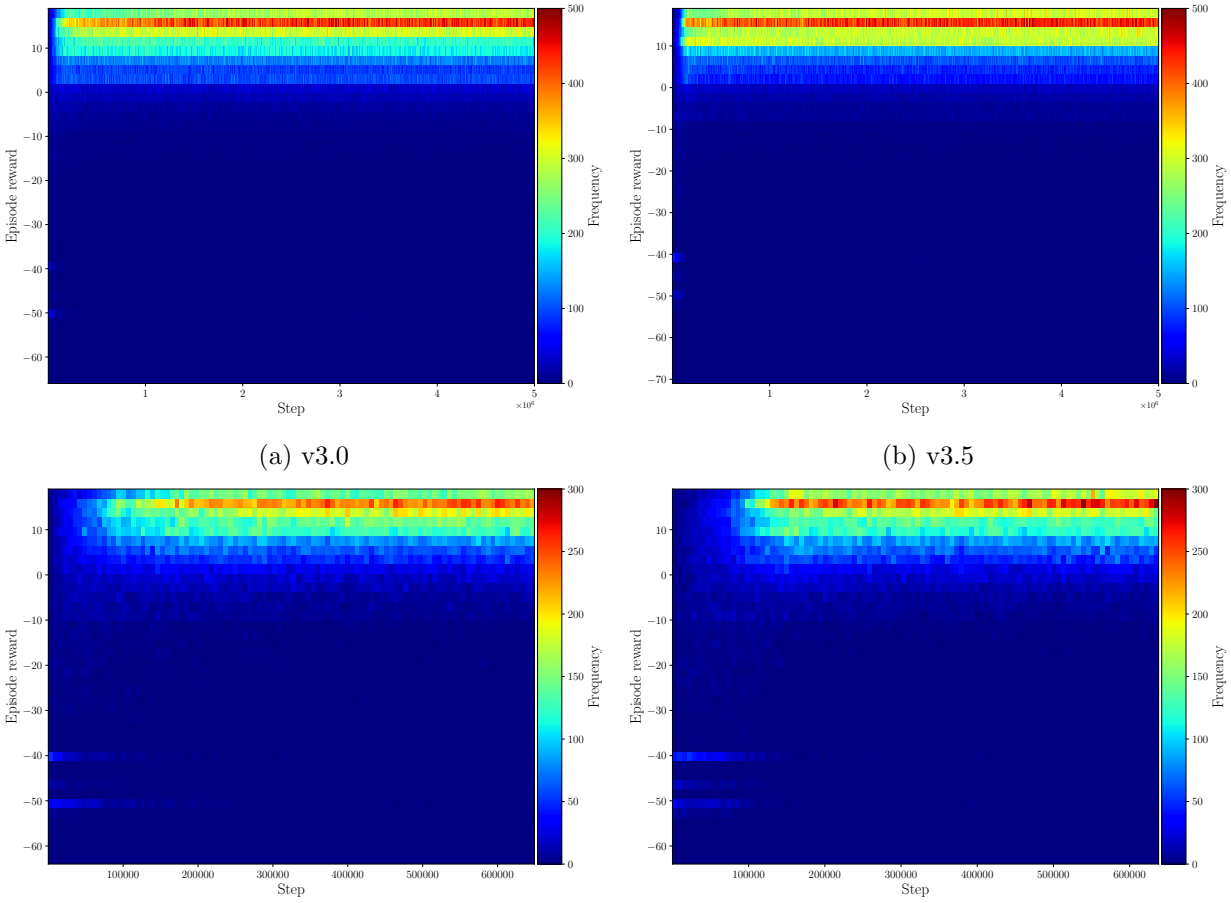
(a) v1.0 (b) v1.5

Figure 5: Training process for v1.0 and v1.5 agents.



(a) v2.0 (b) v2.5

Figure 6: Training process for v2.0 and v2.5 agents.



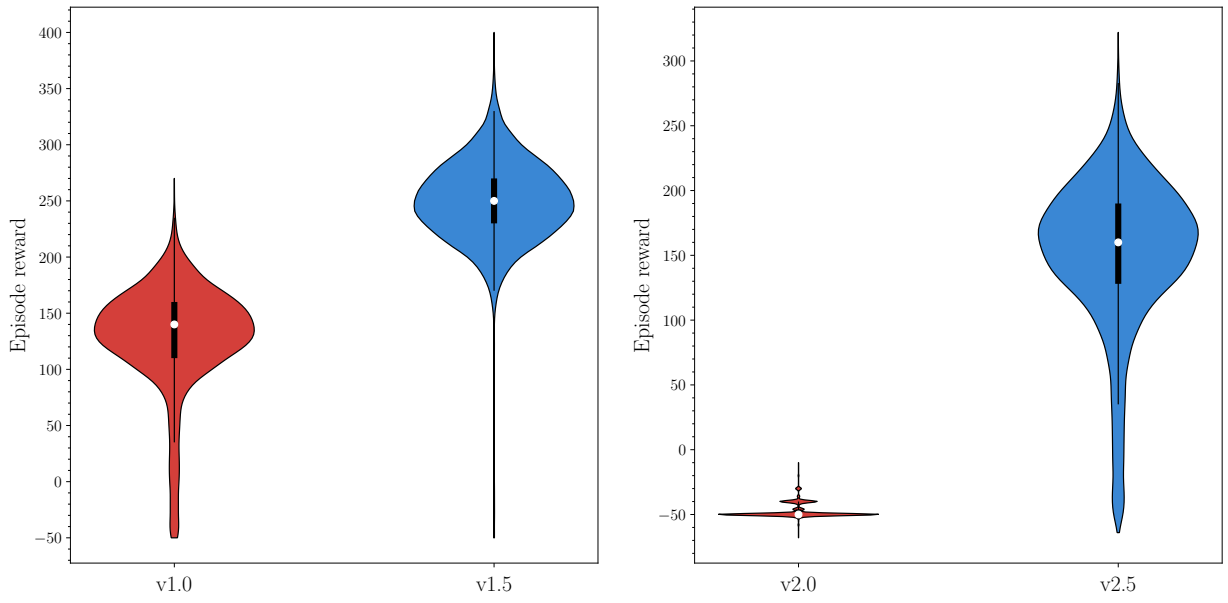
(a) v3.0

(b) v3.5

(c) v3.0 (zoom)

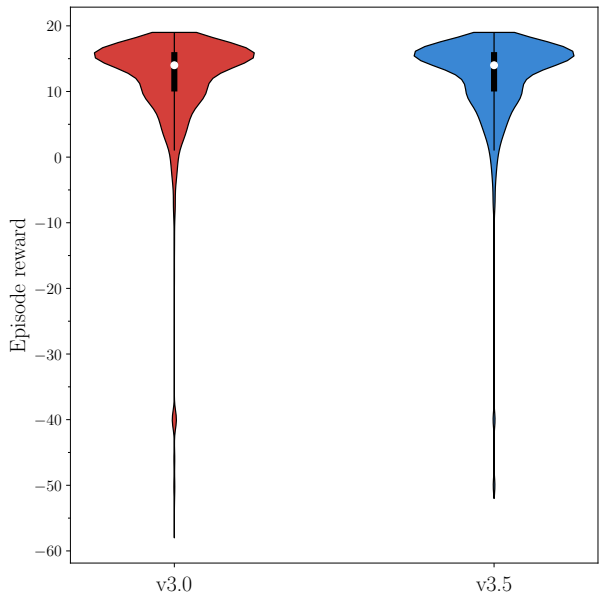
(d) v3.5 (zoom)

Figure 7: Training process for v3.0 and v3.5 agents.

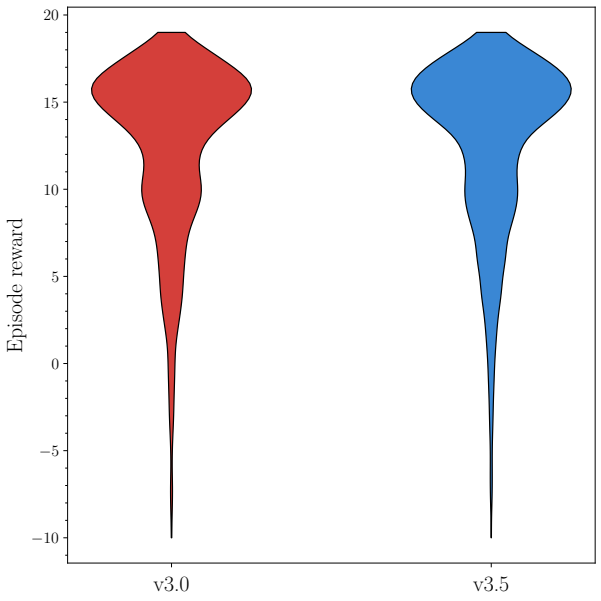


(a) v1.0 and v1.5.

(b) v2.0 and v2.5.



(c) v3.0 and v3.5.



(d) v3.0 and v3.5 (zoom). Only shows episode rewards above −10.

Figure 8: Box + violin plots with data from agent testing over 10000 episodes with different initial states.

6.2 Qualitative results

6.2.1 Behavior of v1.0 and v1.5

Our observations of the behavioral policies of the v.1.0 and v1.5 environments are that both cause the two agents to move in the most efficient cooperative manner, splitting up and pursuing one resource each. In general, an agent heads towards the closest resource unless its companion is already closer to it, in which case it aims for the other one. However, it is important to note that this is not always the case, as aspects of overfitting in the networks cause them to associate one agent with one resource in certain situations, thereby insisting that the agents pursue the resource that is the furthest away from them. This "swapping" maneuver is more apparent in the v2.5 agent, and it may be observed in Fig. 10, section 6.2.2.

As cautioned by Sunehag et al. [18], however, v1.0 did indeed prove to struggle with the "lazy agent" scenario which entails at least one agent not actively contributing to the work to be done, as shown in Fig. 9. This problem was not observed for v1.5, thereby further confirming the benefits of normalization.

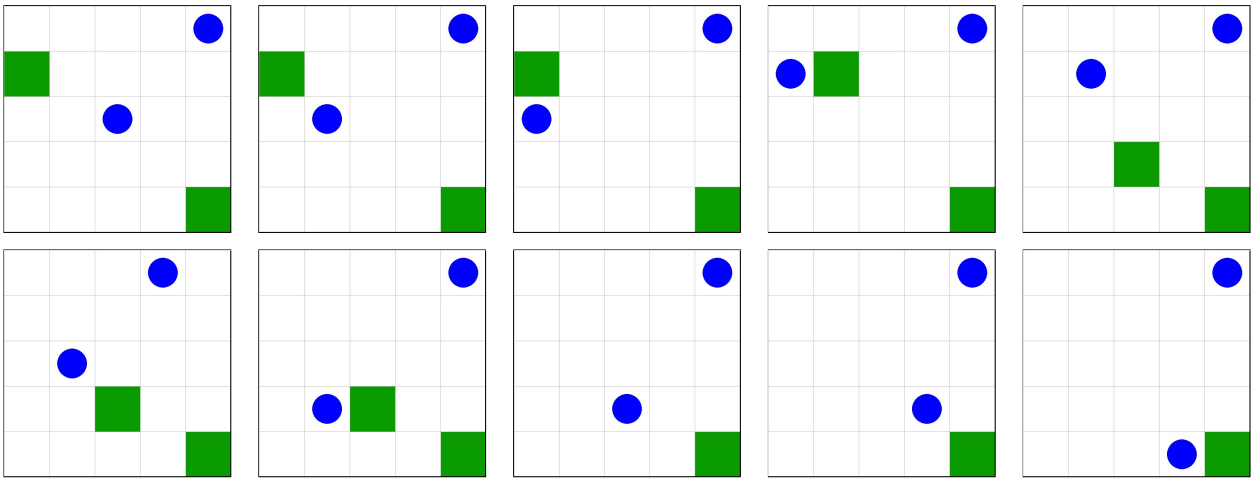


Figure 9: Step-by-step frames of a v1.0 lazy agent. Time flows from left to right and top to bottom.

6.2.2 Behavior of v2.0 and v2.5

As shown in Fig. 6a & 8b, the v2.0 agent does not perform at all. Hence, we do not analyze its behavior.

For v2.5, even with the punishment factor introduced by the presence of civilians, for $\alpha = 6$, the agents are just as incentivized to employ the same greedy uncontrolled extraction scheme as the previous two agents, meaning that one agent pursues one resource regardless of population concentration within its circle of influence. An example is shown in Fig. 10. This may be due to such a brute force approach being statistically favorable in the long run, as the extraction of a resource immediately entails the spawning of a new one. Even if that specific extraction produces a negative reward, the ones to follow most likely will not. Spending steps on grouping up to engage in riot control would then simply serve to accumulate negative default rewards for no good reason. As a result, the agents learn to exploit this environmental mechanism, much like a stock market trader with unlimited capital placing careless bets on assets.

We try to prevent this behavior by increasing α. Setting the parameter value to 7 or 8 yields the same results. However, for $\alpha \in \{9, 10\}$ the agents start to move close together all the time, even if they have the possibility to extract 2 isolated rewards. Moreover, they still perform uncontrolled extractions in order provoke respawns. For an example, see Fig. 11.

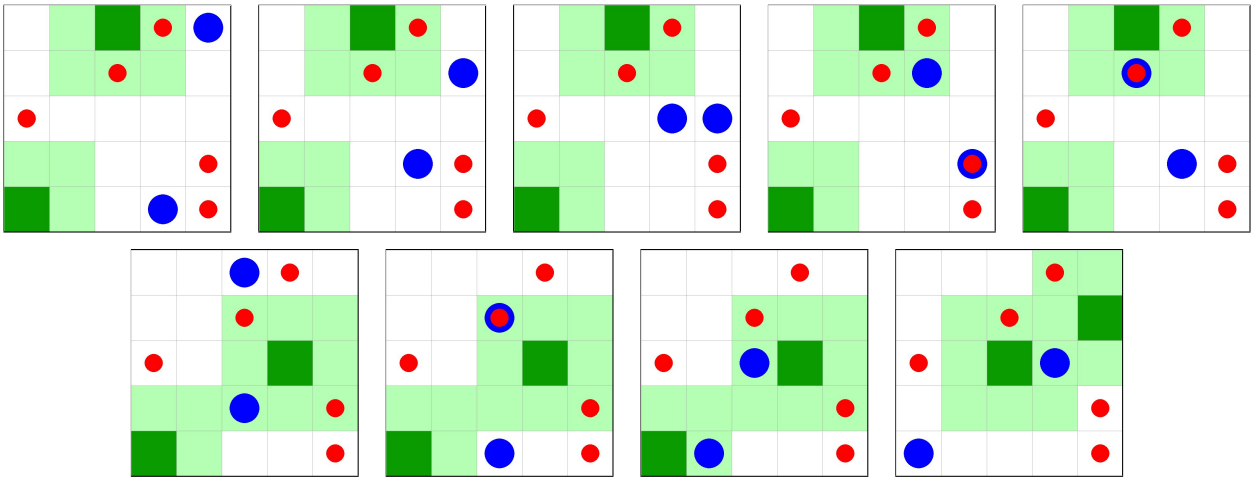


Figure 10: Step-by-step frames of a v2.5 agent splitting up. Time flows from left to right and top to bottom.

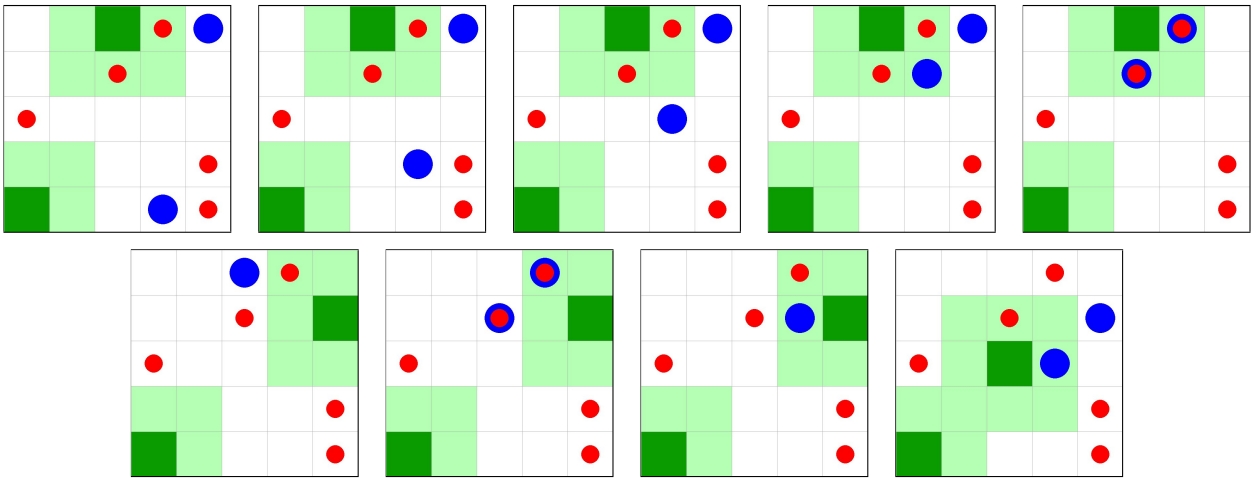


Figure 11: Step-by-step frames of a v2.5 agent grouping up. Time flows from left to right and top to bottom.

6.2.3 Behavior of v3.0 and 3.5

In v3.0 and v3.5 we attempt to counteract the unwanted behaviour of performing uncontrolled extractions by removing resource respawn and finishing the episode once all rewards on the map have been collected. Such changes result in the expected coordinated behavior: agents team up to extract an reward when it's surrounded by civilians (see Fig. 12), but the agents can split up and take one reward each when there are no uncontrolled civilians around (see Fig. 13). Note that only frames for v3.5 are present since the behavior of v3.0 is almost identical.

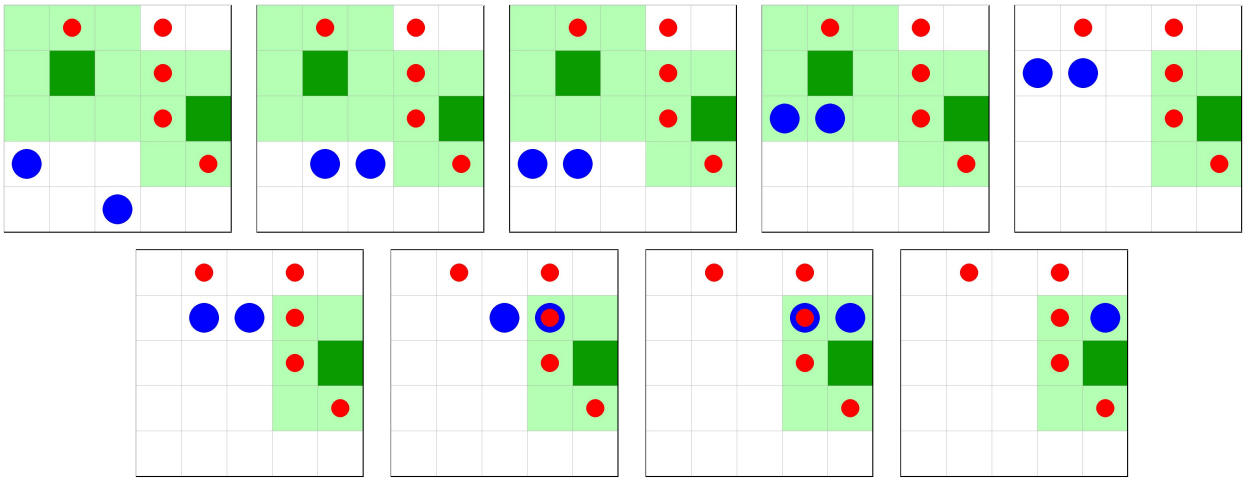


Figure 12: Step-by-step frames of a v3.5 agent grouping up. Time flows from left to right and top to bottom.

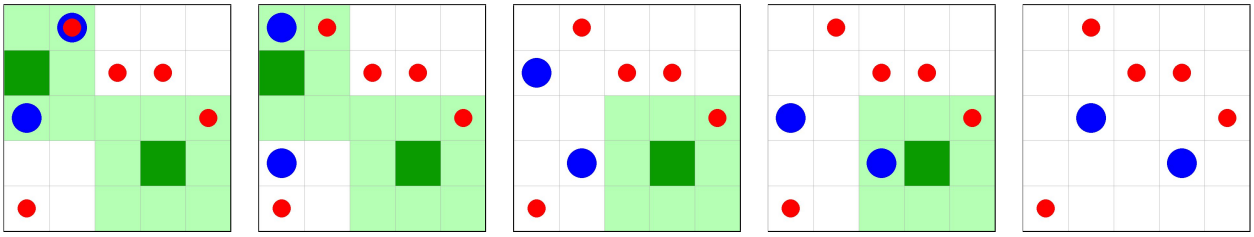


Figure 13: Step-by-step frames of a v3.5 agent splitting up. Time flows from left to right.

6.3 Beyond 5-by-5 grids

All experiments accounted for in the previous sections have been conducted on grids of size $g = 5$. It ought to be mentioned that attempts to train agents on larger grids have been made, up to $g = 10$, with a few additional resources and civilians added, up to $c = 5$ and $n = 15$, with severely unsatisfactory outcomes. Fig. 14 shows an example of an agent trained on a 6-by-6 grid getting stuck after seizing 2 out of 3 resources, which is a very common phenomenon for an environment that is ultimately only slightly more complex than the ones presented above.

We have tried increasing the size of the hidden layers in our DQN to tackle these larger environments, but these bigger networks have not served to improve performance in any environment. The drastic increase in state space simply turns out to be unmanageable for our SARL solution to what is ultimately a MARL problem. From perusing related work we have not seen any publication using this method, but rather one DQN per agent, which turns out to be a significantly more scalable solution.

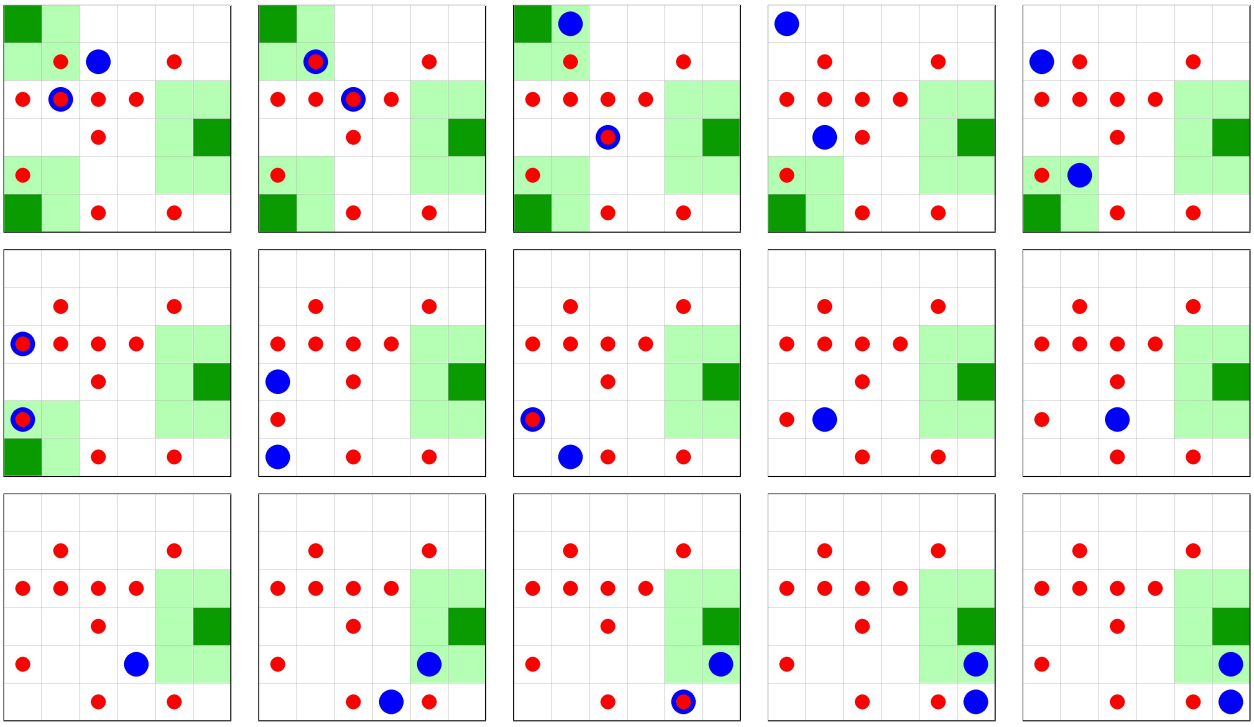


Figure 14: Step-by-step frames of a v3.5 agent getting stuck on a 6-by-6 grid. Time flows from left to right and top to bottom.

7 Conclusions

In this paper, we have proposed a spatial MAS based on resource extraction where coordination is essential for success. Even though converting a MARL problem into a into a SARL one may backfire as the state-space grows and the number of agents increases, we have shown that it is possible to train a MAS via RL with a single centralized DQN when the environment complexity is low. In particular, we have

1. showcased the importance of normalized inputs when using ReLU,
2. demonstrated how to simplify the state vector so that the DQN can successfully learn,
3. provided a detailed explanation of the agents' behavior while fighting environment exploits, and
4. shown that centralized learning and execution is not effective for larger state spaces and/or number of agents.

Looking ahead, one could apply some of the latest distributed learning/execution schemes in the literature (section 4) to the proposed game in order to analyze results for larger grids, more civilians, agents and resources. Another direction might be to transfer an agent trained in a 5-by-5 grid into a larger grid-size, trying to avoid a full re-training process. It could also be interesting to analyze the effect which parameters such as the radius of influence or the agent power have on the agents' behavior. As a final idea, continuous state and action spaces may be a good challenge.

References

[1] M. Abdoos, N. Mozayani, and A. L. C. Bazzan. Traffic light control in non-stationary environments based on multi agent Q-learning. In *2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, pages 1580–1585, 2011.

[2] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. OpenAI Gym, 2016.

[3] T. Chu, J. Wang, L. Codecà, and Z. Li. Multi-Agent Deep Reinforcement Learning for Large-Scale Traffic Signal Control. *IEEE Transactions on Intelligent Transportation Systems*, 21(3):1086–1095, 2020.

[4] K. Driessens. Lecture notes in Intelligent Systems, February 2020.

[5] M. Egorov. Multi-agent deep reinforcement learning. *CS231n: Convolutional Neural Networks for Visual Recognition*, pages 1–8, 2016.

[6] J. Foerster, G. Farquhar, T. Afouras, N. Nardelli, and S. Whiteson. Counterfactual Multi-Agent Policy Gradients. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), Apr. 2018.

[7] J. Jiang, C. Dun, T. Huang, and Z. Lu. Graph Convolutional Reinforcement Learning. In *International Conference on Learning Representations*, 2020.

[8] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. In Y. Bengio and Y. LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[9] K. Lin, R. Zhao, Z. Xu, and J. Zhou. Efficient Large-Scale Fleet Management via Multi-Agent Deep Reinforcement Learning. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, page 1774–1783, New York, NY, USA, 2018. Association for Computing Machinery.

[10] Y. Liu, L. Liu, and W.-P. Chen. Intelligent traffic light control using distributed multi-agent Q learning. In *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, pages 1–8, 2017.

[11] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, and I. Mordatch. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6382–6393, Red Hook, NY, USA, 2017. Curran Associates Inc.

[12] T. McNally. keras-rl2. `https://github.com/wau/keras-rl2`, 2021.

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[15] A. OroojlooyJadid and D. Hajinezhad. A Review of Cooperative Multi-Agent Deep Reinforcement Learning, 2021.

[16] P. Shinners. Pygame. `http://pygame.org/`, 2011.

[17] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[18] P. Sunehag, G. Lever, A. Gruslys, W. M. Czarnecki, V. Zambaldi, M. Jaderberg, M. Lanctot, N. Sonnerat, J. Z. Leibo, K. Tuyls, and T. Graepel. Value-Decomposition Networks For Cooperative Multi-Agent Learning, 2017.

[19] M. Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 330–337, 1993.

[20] J. Yang, A. Nakhaei, D. Isele, K. Fujimura, and H. Zha. CM3: Cooperative Multi-goal Multi-stage Multi-agent Reinforcement Learning. In *International Conference on Learning Representations*, 2020.