LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

INSTITUT FÜR INFORMATIK
LEHRSTUHL FÜR MOBILE UND VERTEILTE SYSTEME

MASTER THESIS

Exploring the impact of Markets on Multiagent Reinforcement Learning

Zarah Zahreddin

Entwurf vom November 20, 2021

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN
INSTITUT FÜR INFORMATIK
LEHRSTUHL FÜR MOBILE UND VERTEILTE SYSTEME

MASTER THESIS

Exploring the impact of Markets on Multiagent Reinforcement Learning

Zarah Zahreddin

Professor:	Prof. Dr. Claudia Linnhoff-Popien
Supervisor:	Kyrill Schmid Robert Müller
Submission Date:	13. December 2021

I hereby affirm that I wrote this Master Thesis on my own and I did not use any other sources and aids than those stated.

Munich, 13. December 2021

.....
Signature

Abstract

Hier kommt der Abstract hin. Hier kommt der Abstract hin.

Contents

1. Introduction

- Motivation
- Goal
- Research Question
- Structure

2. Background

Reinforcement learning (RL) is a process that requires, on one hand, interactive parts, and on the other algorithms that improve interactions. The following Section 2.1 introduces the general concept of RL and its specifications. Afterwards, two popular learning algorithms for RL problems are presented: Proximal Policy Optimization (PPO) and the training approach of a deep Q-Network (DQN).

2.1. Reinforcement Learning

Sutton and Barto wrote in "Reinforcement learning: An introduction" [SB18] that RL is based on two components that interact with each other: an environment and an agent, see Figure 2.1. Those interactions take part during a time period with discrete time steps $t \in \mathbb{N}_0$ until a goal is reached or the ending condition applies. Formally, the journey of the agent to find that goal is described as a Markov Decision Process (MDP) [SB18]. When multiple agents act in the same environment, the Markov decision process is called a stochastic game [BBDS10].

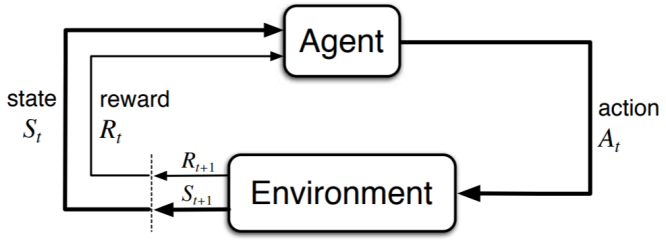


Figure 2.1.: The cycle of agent-environment interaction as shown in "Reinforcement learning: An introduction" [SB18]

One environment state S_t is part of a set S containing all possible states. In most cases, the environment state describes what the agent can see. An agent can often only see a small field of view, which turns a MDP to a partially observable MDP [SB18]. During each point in time t, the agent can interact with the environment by executing an action A_t, which changes the environment state. An example would be moving in the environment, which results in a new area that the agent can now see. In a multiagent environment, every agent chooses its action simultaneously and adds it into a joint action set, which is executed collectively during t [BBDS10].

The reward R_t is element of a set of possible rewards $R \subset \mathbb{R}$ [SB18]. Therefore, the reward can potentially be negative or very low. Depending on the environment, that

value can act as immediate feedback to the agents action. Other times, the reward is received as a result of a whole action sequence or the achievement of a certain state, for instance the goal or subgoals. The general concept of RL, as defined by Sutton and Barto [SB18], is for agents to maximize rewards. Unlike machine learning approaches, here the agent starts with no knowledge about good or bad actions and enhances the decision-making over time.

Sutton and Barto continue by defining the agents' action selection with respect to the current state as a policy π. They explain further that a policy could be as simple as a lookup table, mapping states to actions, or it could contain a complicated search process for the best decision. However, policies most of the time map action-state pairs to a selection probability, with all actions of a state adding up to 100%. During environment interactions, agents receive rewards, which can be used to update the policy accordingly. As an example, the probability of policy $\pi(a \mid s)$ decreases when receiving a negative or low reward, reducing the chances of executing the same action in that specific state again.

While rewards only rate the immediate situation, a value function, i.e. the state-value function $V^{\pi}(s_t)$ for a policy π, can be used to estimate the long-term value of a state s [SB18]:

$$v_{\pi}(s) \doteq \mathbb{E}_{\pi}\left[G_t \mid S_t = s\right] = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1|S_t=s}\right] \tag{2.1}$$

The result is the estimated discounted cumulative reward, an agent could get following that state and choosing actions based on the current policy. The discount factor is defined by Sutton and Barto as $0 \leq \gamma < 1$ and provides a constant that reduces the importance of future rewards. A high γ symbolizes a greater interest in rewards that are far away, whereas a discount of zero only takes the current reward into account. By setting γ smaller than one, it is ensured that the infinite sum results in a value.

Generally, states that offer immediate high rewards could end in a low reward streak. In the opposite case, a low reward state could subsequently yield high rewards. Therefore, value functions are of great use to achieve the maximum reward.

The last part to note about RL is that it entails the problem of balancing exploration and exploitation [SB18]. On one hand, an agent has to explore different options in order to learn and expand its knowledge. On the other hand, agents strive to maximize the reward, which can lead to greediness. An agent could start exploiting its knowledge too early, choosing actions of which it knows to result in positive rewards. However, if an agent does not explore enough, the best action sequence will stay hidden and the agents' knowledge will not improve.

2.2. Proximal Policy Optimization

In 2017 Schulman et al. introduced the concept of PPO in the article "Proximal Policy Optimization Algorithms" [SWD+17]. Policy optimization is the improvement of the action selection strategy π based on the current state s_t. This is achieved by rotating

two steps: 1. Sampling data from the policy and 2. Optimizing the objective with that data through several epochs [SWD+17].

Using those steps results in the agent gathering a small batch of experiences while choosing actions with a policy π. Afterwards, this batch is used once to enhance the current policy. Then, the experiences are discarded, and the agent uses the updated policy to gather a new batch. By repeating those two steps the Agent can learn to choose better actions. PPO is used here, to prevent drastic policy changes to stabilize the learning process.

The origin of PPO lies in a similar approach called Trust Region Policy Optimization (TRPO). TRPO also restricts policy updates by defining a trust region [SLA+15]. This is achieved by maximizing the following function [SWD+17]:

$$\underset{\theta}{maximize}\,\hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)}\hat{A}_t - \beta\, KL[\pi_{\theta_{old}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)]\right] \tag{2.2}$$

The expectation $\hat{\mathbb{E}}_t$ indicates, that an empirical average over a number t of samples is used for estimation, and the algorithm alternates between sampling and executing these calculations. The variable $\hat{A}_t$ describes an estimator of the advantage function. This function was defined in the paper "Trust Region Policy Optimization" [SLA+15] with $A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s)$. The first part calculates the state-action value, estimating the upcoming rewards for an agent, starting at state s and initially selecting action a. Afterwards, the action selection is based on the current policy π.

The second part contains the state value function $V_\pi(s)$, which works very similarly by starting at state s and using the policy. However, the difference is that the agent always chooses actions according to the policy. The advantage that is produced by the function $A_\pi(s,a)$ shows, whether a profit could be gained when deviating from the policy, by specifically choosing action a.

The fraction $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ in the Minuend of function (2.2) can be replaced by $r(\theta)$ and represents the probability ratio of an action in the current policy in comparison to the old policy [SWD+17]. Here, θ represents a policy parameter. The result of $r(\theta)$ is greater than one, if an action is very probable in the current policy. Otherwise, the outcome lies between zero and one. Schulman et al. [SWD+17] further extract the first part of function (2.2) as the surrogate objective:

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)}\hat{A}_t\right] = \hat{\mathbb{E}}_t\left[r(\theta)\hat{A}_t\right] \tag{2.3}$$

Would only this part of function (2.2) be maximized on its own, it could result in large outcomes, which would lead to drastic policy updates. In order to stay in a trust region, as the name suggests, a penalty is subtracted from the surrogate function (2.3). The penalty is the Subtrahend of equation (2.2) and contains the fixed coefficient β. Regardless of the function details and outcome of KL, the coefficient β is hard to choose, since different problems require different penalty degrees [SWD+17]. Even during a

training process it could be necessary to adapt the coefficient, due to changes.

Therefore, Schulman et al. introduced

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min \Big(r(\theta)\hat{A}_t, \quad clip(r(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \Big) \right] \tag{2.4}$$

which is very similar to equation (2.2) but does not require coefficients. The first min entry contains L^{CPI} (2.3). The second part contains a *clip* function, which narrows the space of policy mutation with the small hyperparameter ϵ. After applying the clip function, $r(\theta)$ lies between $[1-\epsilon, 1+\epsilon]$. Calculating the minimum of the clipped and unclipped probability ratio produces the lower bound of the unclipped $r(\theta)$, preventing the policy to change drastically.

Finally, the following equation is introduced

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right] \tag{2.5}$$

with c_1 and c_2 as coefficients. The authors point out that the loss function $L_t^{VF} = (V_\theta(s_t) - V_t^{targ})^2$ combines the policy surrogate and the value function error term and is necessary once a neural network shares parameters between policy and value function. Additionally, an entropy bonus S is added to ensure exploration.

Furthermore, Schulman et al. point out that the policy is executed for T time steps, with T being a smaller value than the overall episode duration. Until now, the advantage function calculates values that run over an infinite loop, see the value function (2.1) for example. Hence, the advantage function needs to be adjusted as well. It is necessary that the future estimations do not exceed that time step limit. In this context the following advantage function is used [SWD+17]:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1} \tag{2.6}$$

$$\text{where} \qquad \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{2.7}$$

Schulman et al. also showed an example of the PPO algorithm, see Fig. 2.2. The example uses an actor-critic approach, which means that a critic is responsible to approximate the value function of the policy, and the actor in turn improves the policy based on the approximation results of the critic [KT03]. Here, N detonates actors collecting data in T time steps in each Iteration. Meanwhile, the critic computes the estimations of the advantage values. Afterwards, the policy is replaced with a new one, in which the function $L_t^{CLIP+VF+S}(\theta)$ (2.5) is optimized during K epochs. For the optimization process, a small random batch of the previous time steps is used.

2.3. Deep Q-Network

Another learning approach that is often compared with PPO is the training algorithm of a deep Q-Network with Q-learning and experience replay. Instead of improving a policy, here, agents improve by maximizing a value function. Hence, This algorithm relies on

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

Figure 2.2.: Exemplary PPO Algorithm, as shown in "Proximal Policy Optimization Algorithms" [SWD+17]

the action value function, that is formally defined as follows [MBM+16]:

$$Q^{\pi}(s, a) = \mathbb{E}\left[\, R_t \mid s_t = s, a\right] \tag{2.8}$$

R_t represents the discounted cumulative reward $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$. The estimated outcome is calculated by starting at a state s, executing a specific action a and reaching the next states by using a policy π. Mnih et al. [MKS+15] state, that the optimal action-value can be approximated with a deep convolutional neural network and the following function:

$$Q^*(s, a) = \max_{\pi} \mathbb{E}\left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots | s_t = s, a_t = a, \pi\right] \tag{2.9}$$

The difference between function (2.8) and (2.9) is, that in the second one, a policy is chosen, which optimizes the outcome. Mnih et al. continue by stating, that in a scenario where the sequence s' of all actions a' are known, the optimal $Q^*(s', a')$ of the next state can be calculated. Then, this Bellman equation could be applied [MBM+16]:

$$Q^*(s, a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q^*(s', a') \mid s, a\right] \tag{2.10}$$

Many RL algorithms estimate this function through iterative update, by calculating $Q_{i+1}(s, a) = \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s', a') \mid s, a\right]$, $Q_{i+2}, \ldots$ [MKS+13]. Eventually the optimal Q value is reached with $i \to \infty$. Those calculations proved to be very impractical, since they require a lot of computational work, which is why Mnih et al. introduced the Q-network at this point. As a result, the parameters of the Q function are extended with θ as network weights ($Q(s, a; \theta)$).

However, the researchers argued that using a neural network in combination with the Q function proofed to be unstable. According to the authors, this is caused by correlating observations that are used to calculate the function. Additionally, small updates to the action value may lead to drastic changes of the policy. Such problems change the connection between Q values and their successive target values $r + \gamma \max_{a'} Q(s', a')$. To

overcome these issues, Mnih et al. introduced two new concepts: 1. An experience replay that enables random sampling of observations and 2. An iterative update process of the action values approaching the target values. The target values are only updated periodically in their implementation.

In Figure 2.3 a deep Q-learning approach with an experience replay is shown. The experience replay contains the acquired agent knowledge of each time step in form of a quadruple: (old state, action, reward, new state). The experience values are then stored into the replay memory D across multiple episodes. The states are parameters of Φ_t in the example, since they are preprocessed, to match the network input conditions.

In addition to the action value function Q, the target action-value $\hat{Q}$ is initialized to enable iterative updates. In order to fill the memory, the agent first selects actions and acts in the environment. The action selection here is based on the ϵ-greedy policy, meaning that with a probability of ϵ a random action is chosen [MKS+15]. Otherwise, the best option according to the Q-value is selected.

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $\left(\phi_t, a_t, r_t, \phi_{t+1}\right)$ in D
    Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Figure 2.3.: DQN with Experience Replay, as shown in "Human-level control through deep reinforcement learning" [MKS+15]

Executing the selected action results in a memory entry in the form of the earlier described quadruple. Afterwards, a minibatch of the replay memory is randomly sampled to calculate the difference of the target values with the old weights and the action values with the current weights. The parameter y_i holds information about the target values, if the episode is not about to end. Otherwise, only the reward is assigned to y_i. A gradient descent step is performed on the function, which means that the local minima of the

function is searched by tweaking the parameter θ.

Finally, every certain amount of steps C the target network is set to the current Q-Network. The suggested process offers several advantages [MKS+15]. For instance, the replay memory leads to a smaller deviation or fluctuation in the parameters. The random samples of minibatches can proof to be efficient, since an experience might be used multiple times to update the network weights. Furthermore, through the randomness in the samples the correlation of steps is interrupted, leading to a decrease of variance in between updates. And lastly, updating the target network periodically improves the stability of the learning process.

3. Related Work

Realistic RL scenarios often involve multiple agents solving problems together, for example robots working in warehouses and factories. Such multiagent environments come with many difficulties. On one hand, in a scenario where agents work independently, it is very probable that they get in each other's way. They aim to achieve the highest score or finish their individual tasks, preventing the overall goal to be achieved.

In cooperative environments, on the other hand, agents share the reward and therefore can not tell who contributed useful actions. The difficulties of agents working independently is discussed in Chapter 3.2 whereas the cooperation challenge is the focus point of the next Chapter.

3.1. Credit Assignment Problem

Sutton and Barto [SB18] define a RL environment as cooperative, when agents execute their actions collectively each time step but receive one overall reward in return. In this case, individual learning is difficult or even impossible. Collective actions may contain bad choices that would be rewarded or, in case of a penalty, good actions that would be punished. Deciding which agent deserves more or less of the common reward is referred to as the credit assignment problem (CAP) [Min61].

The CAP originated in a one-agent environment that only returned reward once the goal is reached or the terminating condition applied. A popular example of this is a chess game. In 1961, Minsky [Min61] elaborated on this by explaining that a player wins or loses the game, but cannot retrace which decision got him there. Later on, Sutton decomposed the CAP into subproblems, namely the structural and temporal CAP [Sut84]. He suggests, that the temporal problem lies in assigning credit to each chess move by determining when the position improves or worsens, rewarding or penalizing that certain action. On the contrary, the structural CAP is assigning credit to the internal decision that leads to each particular action.

Transferring the single-agent CAP into a multiagent environment, Agogino and Tumer [AT04] imply that the problem shifts from being of temporal to structural type. They explain that while a single agent faces the temporal CAP due to many steps taken within an extended time period, in the multiagent case it becomes a structural CAP because of multiple executed actions in a single-time-step. Since the actions are executed all at once, the challenge is now evaluating the decision that lies underneath.

Over the years many solutions and theories emerged in order to solve CAPs in multiagent environments [RB09], [ZLS+20], [AT04]. A popular example for a simple approach to solve the problem is the difference reward (DR) [NKL18], [YT14], [AT04]. The idea

is to calculate the overall reward with the joint multiagent actions as always. Then, another reward is calculated for one agent if that agent would not have submitted its action. The difference between those two values indicates how profitable the action of the analyzing agent is. A high DR indicates a lucrative action, since excluding it leads to a small Subtrahend value. With this method, each agent has the opportunity to learn how their actions contribute to the global reward, enabling individual learning.

Formally, in order to find the DR $D_i(z)$ for an agent i and the joint action set z, the following function applies [AT04]:

$$D_i(z) = G(z) - G(z_{-i}) \tag{3.1}$$

The first part of the equation $G(z)$ represents the overall reward of the action set of one time step. The second part $G(z_{-i})$ demonstrates the result of the same actions and time step, excluding only the action of agent i. Another approach to find the DR is to select a default action in $G(z_{-i})$ for the analyzing agent i, instead of dropping its action completely [VG96]. Calculating the equation (3.1) results in $D_i(z)$ of one agent i.

An example for this approach would contain two agents, i and j, that act in an environment. In a time step t agent i executes a good action and j a bad action. For this scenario, good actions produce 0.6 as reward and bad choices result in -0.5 as punishment. In this example, the cooperation reward is the sum of the rewards, which would be 0.1. Therefore, agent j would be rewarded for selecting a bad action through the positive reward of 0.1 and could end up learning this strategy. However, using the DR function here a reward of $D_j(z) = 0.1 - 0.6 = -0.5$ is calculated for agent j and $D_i(z) = 0.1 - (-0.5) = 0.6$ for agent i. Thus, the credit is assigned to each agent depending on their contribution. Since there are only two actors, their DRs match the actual environment rewards of the actions.

As a downside, this approach is often inefficient or even infeasible [NKL18]. Nguyen et al. [NKL18] imply, that large domains could make this calculation impossible. Sometimes, environment states need to be stored and replayed for every agent in order to calculate $G(z_{-i})$. Agogino and Tumer [AT04] claim that this simple function is not always applicable, since it can get difficult to exclude agents' actions. Nevertheless, this formula is often coupled with the CAP in research papers and builds a base for many advanced solutions of this topic.

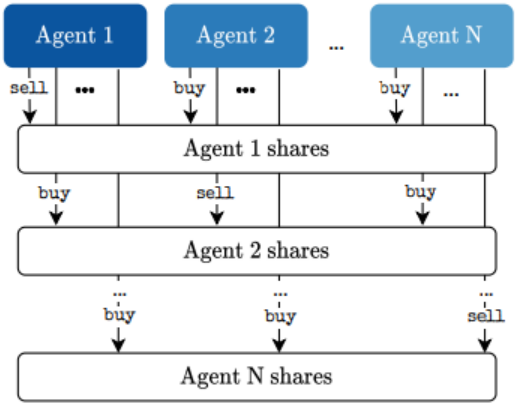
3.2. Markets

As described earlier, agents that share an environment and act independently can often hinder each other from reaching the common or individual goal. Sutton and Barto defined a game to be competitive, when agents receive varying reward signals [SB18]. In most cases, agents follow a mixed-motive, meaning that their individual rewards could sometimes align and sometimes be in conflict. An environment is purely competitive, when the increase in reward of one agent leads to reward decrease of the others [SBM+21].

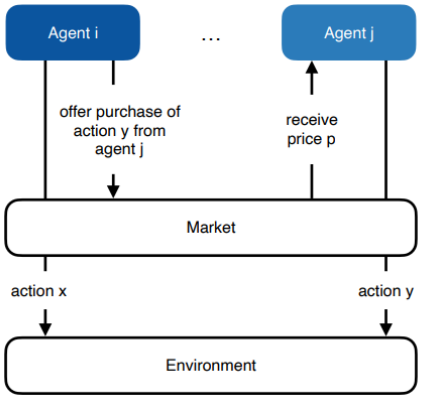
Schmid et al. introduced in "Stochastic Market Games" [SBM+21] concepts that

add incentives when agents act cooperatively in mixed-motive settings, to improve the overall rewards for all participants. The idea of a Stochastic Market Game (SMG) is to enable dominant cooperative strategies through a global and impartial trading market. According to the researchers, a stochastic game becomes a SMG if two conditions are met. First, the environment actions of agents are extended with market actions. Second, the reward function adjusts the calculated rewards based on agreements met in the market executions. Furthermore, Schmid et al. defined two types of markets: unconditional and conditional markets.

They compare the concept of unconditional markets to companies and shareholders, since shareholders do not need to fulfill any conditions to receive the dividends. In unconditional SMGs both companies and shareholders are agents that buy and sell shares as market actions. Figure 3.1a shows such a shareholder market (SM). During each time step, every agent has the possibility to put their share on the market or to announce a buying offer directed to another agent.



(a) Shareholder market (taken from "Stochastic Market Games" [SBM+21])



(b) Action market

Figure 3.1.: Illustrated Markets as defined in "Stochastic Market Games" [SBM+21] during one time step

If the buying offer coincide with a share that is up for sale in the same step, a market transaction is registered. Now, the shareholder participates in the reward of the current step of the transaction agent by a fixed dividend d. Schmid et al. mention that an optional price p can be defined as an amount a seller receives from the buyer upon each share purchase. However, they claim that agents with high rewards are very likely to gift their shares in order to align the goals of the other agents with their own. Shareholders profit from the success of the selling party through the dividends.

An example here would be agent i who wants to sell a share and agent j that wants to buy a share specifically from agent i with both being in time step t. In this scenario, the offers coincide and a share of i is now claimed by j. If i now receives a reward of one and the dividend is 20%, agent j would get a reward of 0.2 in addition to its own

reward. However, in a scenario where agent i decided not to sell, agent j would not be able to claim the share and would not profit from the reward of agent i. After each step, the shares are resolved and the matching starts again.

On the contrary, the authors define conditional markets similar to purchase contracts, where buyers pay a fixed price p to sellers when they in turn meet the buyers' demand. A proposed conditional SMG is the so-called action market (AM). In this case, actions are extended with a buying offer, containing one expected action from one specific agent, see figure 3.1b.

Again, in an example with two agents, i and j, i could need a resource that is currently occupied by j. Hence, agent i would profit from j giving up the resource and therefore offers to buy from j the action "release". Formally, agent i executes action $\overrightarrow{a}_{i,t}$ here, containing an environment action $a_{i,t}^{\text{env}}$ and the offer to agent j $a_{i,j,t}^{\text{offer}}$ [SBM+21]. The offer proposal shows that agent i is willing to buy from j at time step t a specific action $a_{j,t}^{\text{env}}$. If agent j fulfills the conditions and releases the resource in the same step t, it would receive a fixed price from agent i in form of reward. However, if that is not the case, the market conditions do not apply and j is not paid by agent i.

In both market settings, a purchase is established if the specified agent happens to execute an action that matches with a buyer. It is important to emphasize, that the matching is performed during a time step, leaving it to chance, whether purchases take place. For instance, in an AM, agents do not know in advance if and what action another agent could be buying from them. Despite this uncertainty, the researchers showed, that both market implementations yielded promising results. An increase of the overall rewards of participating agents in mixed-motive games was seen.

4. Approach

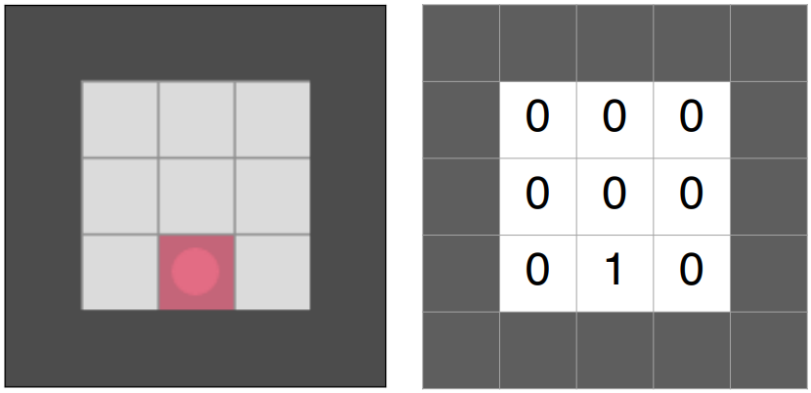
This Chapter introduces the RL components of this research, namely the coloring environment and its agents. Since, the aim here is to execute learning with markets in various multiagent scenarios, the implementation of the three agent compositions: cooperation, mixed-motive and competitive is shown in Section 4.1. The compositions are tightly coupled with reward distribution, which is why Section 4.2 covers the detailed implementation of reward calculations. Furthermore, the process of learning and how the two algorithms PPO and DQN were implemented is the topic of Section 4.3. Lastly, the two market implementations and their influence on the rewards are discussed in Section 4.4.

4.1. Coloring Environment

A RL environment is a versatile and unbiased instance, that can in this case visualize changes and agent behavior. In figure 4.1a, the environment used in this work is presented. It originated from an openAI project called "Minimalistic Gridworld Environment" [CBWP18], which is designed for one agent whose main goal is to solve labyrinth puzzles. However, for the purpose of this research, the environment is changed heavily, becoming the "Coloring Environment". Multiple agents can act in the new instance to try and achieve a common goal - to color all walkable cells.

In the visualization 4.1a the outer dark cells show walls and the white cells are floors. The agent is represented with the red dot and the cell on which it is located is marked with the same color. By moving on the available space the agent can color more cells. Every environment cell holds information about the object type it represents, being either walls, floors or agents. Furthermore, each object class contains information about its current color, whether it is accessible for an agent and, in case of a floor tile, if it is colored. Figure 4.1b shows a simplified environment observation an agent processes each time step.

Floor cells manage the coloration state in binary form, as displayed in 4.1b, with a one signalizing that the cell is colored. The environment reacts to agent movements by coloring the cells they visit. The environment is successfully solved, once all fields are colored. Otherwise, agents loose by using up a limited amount of steps. If a cell is already in coloration state one and an agent walks over it again the bit is switched, and the cell is reset to zero, removing its color. Besides moving up, down, left and right an agent can also execute the action wait, to stay in place.



(a) Human visualization of the coloring environment. A dot represents one agent. Cells change their color when agents move on them.

(b) Simplified agent observation of the current environment state. The number one represents a colored cell.

Figure 4.1.: Representations of the coloring environment

4.1.1. Compositions

When multiple agents are placed in the coloring environment together, there are several ways how they will behave towards each other. Depending on the setting, even the environment distinguishes how certain actions affect the state. Per default agents will try to work together, to reach the environment goal. Alternatively, they could work independently or even compete with each other.

Each agent has a different random color. Cells adopt the color of the agent that walks over it. The primary focus in cooperative agent compositions however, is only the binary state. The agents always receive the same reward, regardless of the colors on the grid. An extreme example here would be one agent that colors the whole environment on its own. Naturally, this would result in a high reward and for cooperation this means that all other agents would get the same amount.

In a DR setting however, agents are able to estimate their contribution, in order to improve their actions. This implementation executes the default action wait to find $G(z_{-i})$, see equation (3.1). By choosing wait as the default action, agents can learn what the environment outcome and general coloration percentage would be if they had not participated in the current step. It is important to note here, that DR settings are always an extension of the cooperation mode and are never used together with other compositions and markets.

In mixed-motive settings the colors are of importance. Agents only gain rewards based on their individual contributions. Thus, the rewards are generated by looking up each percentage a color is present and assigning that value to the same colored agent as reward. For example if a red agent colored 60% of the grid red the reward for that agent would be 0.60.

In a fully competitive mixed-motive scenario the reward calculations stay the same,

only disabling the bit switching for opponent colors. Therefore, agents can directly capture already colored cells when they walk over them. However, if the cell contains the same color as the agent that moved on it, it is reset instead. Therefore, taking over the cells of the opponents is beneficial, since it increases the presence of the own color, leading to a higher reward.

In comparison, the basic mixed-motive composition shows no advantages resetting colored opponent cells. Instead, cell resets are always punished with a small negative reward, regardless of the composition. Hence, it is not likely that the agents work against each other in mixed-motive compositions, yielding a neutral or independent behavior between agents.

4.1.2. Observation

The observations agents receive from the environment are always generated from their individual point of view, with them in the center. The observation only contains a restricted area around them, making the environment a partially observable MDP. In large environments this feature increases the difficulty. An example for an observation of an agent is shown in 4.2. This observation depicts the internal state of the environment visualization of image 4.1a.

```
[
  [ [210] [210] [210] [210] [210] [210] [210] ]
  [ [210] [210] [210] [210] [210] [210] [210] ]
  [ [210] [301] [301] [301] [210] [210] [210] ]
  [ [210] [301] [301] [412] [210] [210] [210] ]
  [ [210] [301] [301] [301] [210] [210] [210] ]
  [ [210] [210] [210] [210] [210] [210] [210] ]
  [ [210] [210] [210] [210] [210] [210] [210] ]
]
```

Figure 4.2.: The internal agent observation

Per default the agent has a view size of a seven by seven grid, represented in a three-dimensional array, similar to a picture with RGB information. Here, all highlighted entries are part of the grid that is shown in Figure 4.1a and the red array shows the agent. The first dimension of the observation array contains the whole internal observation of an agent. The second array dimension represents the environment grid. Since the view size of the agent exceeds the dimension of the 5x5 visual grid, the additional rows and columns are filled with placeholders, in this case walls. All highlighted row entries can be mapped to the columns of 4.1a, starting from the top left of the visualization. The last dimension contains cell information, which is always composed of three elements.

The first cell element defines the object type with one being just an empty cell, two

shows a wall, three a floor tile and four an agent. The second element stores the coloration status, showing whether a cell is colored, with zero signalizing an uncolored cell. Since agents can not walk onto walls, represented as [2 1 0], that object type always has a coloration state of one. The third cell encoding describes the color of a cell. To better distinguish the types in the visual representation, walls are zero for the color black and floors are initially white with encoding one. Each agent is assigned a number from two upwards which in turn stands for a randomly generated RGB color. The Floor color encoding is overwritten with the agents color code when the cell is captured. For example, should the agent move to the left, the cell of the previous position is now a colored Floor cell [3 1 2]. The new position of the agent is row three and column four with cell encoding [4 1 2].

4.2. Reward Calculations

The allocation of rewards is closely related to the composition of the agents, which can be specified by the user in training or visualization runs. In addition, the environment shape can be set, a number of agents placed and more. A basic example command for a training run is shown in listing 4.1.

Code Listing 4.1: Exemplary command to execute training with three agents in a coloring environment using PPO as algorithm

```
1 $ python -m scripts.train
2     --algo ppo
3     --model ppo-training
4     --env Empty-Grid-v0
5     --grid-size 7
6     --agents 3
7     --max-steps 20
8     --setting mixed-motive
```

The `--algo` parameter can be either “ppo” or “dqn” to choose a learning algorithm. This argument is the only required setting for training. All other configurations, including those not listed in 4.1, have default values and are shown in Appendix A. With `--model` the destination path is defined, in which all logs, recordings and status updates are stored. Line 4 and 5 configures the environment. In alternative to the empty grid option of `--env`, as shown in figure 4.1a, four homogeneous rooms can be generated with “FourRooms-Grid-v0” to increase the difficulty. The rooms are always of the same size and each room is accessible to all adjoining neighbors by one wall opening, which is random and changes in each episode. The overall size of the grid is set in Line 5. However, all grids in every layout option have outer walls that narrow the area in which agents can move. Hence, in a grid of size seven the agents can only move in a five by five field, due to the surrounding walls.

The amount of agents that act in the environment is set through the argument `--agents` and the maximum quantity of steps they can execute is defined with `--max-steps`. To gain the highest reward, the agents need to color the whole field before they run out of steps. Lastly, the argument `--setting` specifies the composition of the agents. If no setting is set the agents work cooperatively. In the example of 4.1 the setting "mixed-motive" is chosen. The last two options here are "mixed-motive-competitive" and "difference-reward".

Regardless of the composition, agents initially generate separate rewards in each step based on their individual environment change. For instance, agents that color a field produce a positive reward of 0.1, whereas agents that reset a field contribute a penalty of negative 0.1. Agents that just wait generate a reward of zero. The only exception is the setting "mixed-motive-competitive", since agents can capture opponent cells. If that is the case they get a positive reward of 0.1 otherwise the rules stay the same.

Rewards are always written into a list, which is initially returned by the environment, see algorithm 1, line 1. The position in the list indicates the accountable agent, i.e. a reward list of $[0.1, 0, \cdots]$ shows that agent zero is responsible for a reward of 0.1 and so forth. In algorithm 1 the update process of the initial environment rewards during each step is summarized. This step function takes the training arguments of Appendix A into account, which leads to the four conditions below.

Algorithm 1: Reward calculation each step

```
1  observations, rewards, done, info = environment.step(actions)
2
3  if difference reward setting then
4  |   rewards = calculate difference reward for each agent
5  else if cooperative setting then
6  |   rewards = calculate one cooperative reward
7  end
8
9  if market specified then
10 |   rewards = execute market actions and return transaction rewards
11 end
12
13 if done then
14 |   rewards = calculate final rewards
15 end
16
17 return observations, rewards, done, info
```

The first condition checks for a "difference-reward" setting. In this case, the agents work in cooperation but try to solve the CAP by calculating the DR, see function (3.1). To achieve that, the current reward array is summed up, to summarize the overall reward. As a result the variable $G(z)$ of the DR equation is now set to the sum. To find the subtrahend of the equation, the agents are iterated, and their individual calculations

take place. Here, the environment rewards are added up again, but this time the reward of the current agent is set to zero. This sum is the value of $G(z_{-i})$ for agent i. The function (3.1) is now applied for each agent and the initial reward list is updated with the individual DRs.

The last step of the DR reward update is checking, whether any value exceeds an upper or lower bound. If that is the case then the value is set to the corresponding limit. Otherwise, the reward stays as is. The upper and lower bounds are necessary, due to more participating agents possibly leading to a really big or very small sum. For example, very large sums without bounds could in turn decrease the importance of the final reward for reaching the environment goal. The other extreme are high negative sums demotivating agents to move. The bounds in this implementation are set to 0.1 and -0.1.

If the setting is set to the standard cooperation, the reward needs to be changed to a new homogeneous value for each agent, since they share the outcome. Again, the sum of the initial environment reward is calculated and reassigned to each agent position in the rewards array. The values must be clipped again in the same procedure as for the DR values. Here however, all rewards of the array are changed to the same clipped value, should the bounds be exceeded. Settings that contain "mixed-motive" skip all previous reward updates, since in this case each agent keeps their individual value.

As third condition the market argument is checked for a SM or AM. In this case the market transactions changes the rewards. Details of the market process are discussed in Section 4.4. One thing to note here is, that agents can execute market transactions in each step. For example, they can spend their current reward on actions or shares that are for sale or receive the purchase price from buyers, which in turn modifies the rewards.

The last condition depends on the done flag which signalizes the end of an episode. The environment sets done to true, once either the grid is fully colored or the maximum step amount is reached. In this case, the final reward calculations are applied, see algorithm 2. The current rewards are passed as an argument to this calculation, since the list is modified again.

Algorithm 2: Final reward calculation

```
1  if mixed in setting then
2  |  for each agent do
3  |  |  rewards[agent] += agent color percentage on the grid
4  |  end
5  else
6  |  // cooperative setting
7  |  rewards += overall coloration percentage of the Grid
8  |  if difference reward setting then
9  |  |  rewards = calculate difference rewards
10 |  end
11 end
12
13 if market specified then
14 |  rewards = final market adjustments executed on rewards
15 end
16
17 return rewards
```

During the final reward calculations, the different agent compositions are checked again. In a "mixed-motive" or "mixed-motive-competitive" setting, each agents' grid coloration percentage, based on their color presence, is added to the individual reward. Otherwise, the composition is based on cooperation and the general grid coloration, regardless of the colors, is looked up and added to each reward value.

Additionally, the presence of a DR setting needs to be checked here. For the final DR calculations the environment supplies information in the `info` variable of algorithm 1 Line 1. Namely, what the general coloration percentage of the environment would be for each agent, if this agent had executed action wait. Those percentages are subtracted from the cooperative coloration percentage to generate the DRs. Finally, the last market calculations are taken into account, see Chapter 4.4 for details.

4.3. Learning Process

In order to compare different settings and agent compositions easily, each agent manages its own learning improvement, observation and action selection. Therefore, all calculations and estimations are executed independently, for instance policy updates and value estimations. They also set up their own neural networks and optimizers and update them only with their own values. However, the environment still connects the agent experiences, by reacting to all agent actions simultaneously in each step and including visible agents in the observation.

Depending on the learning algorithm the corresponding class is instantiated by the training script, as shown in Figure 4.3. The PPO and DQN classes both extend a base class that provides some abstract methods and a multiprocessing operation to execute

actions on several environments at once. The base class returns data, allowing the training script to create recordings and log files to enable evaluation.

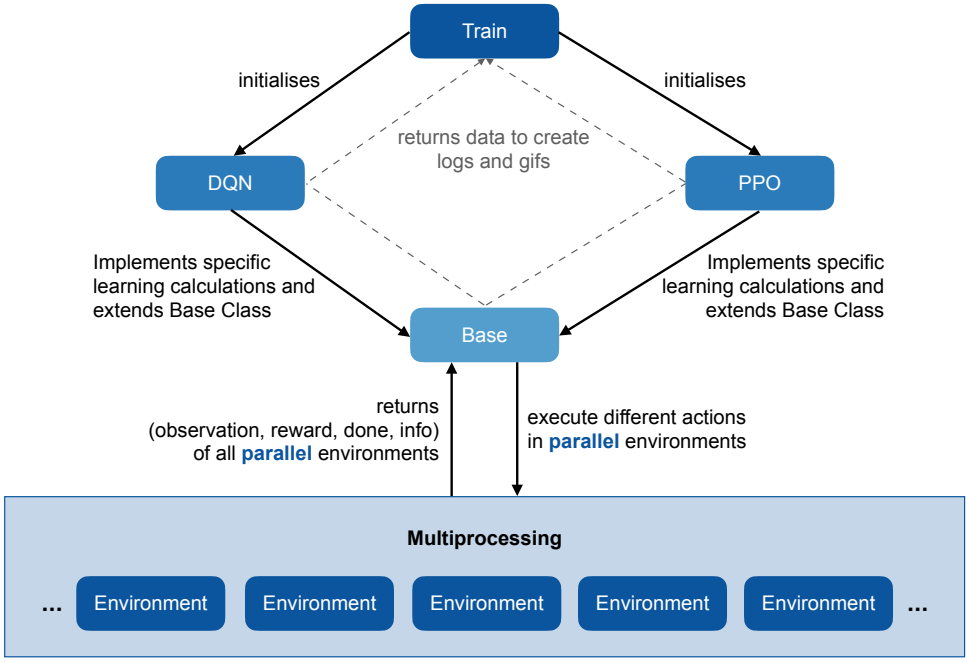


Figure 4.3.: The training structure

First, the training of agents begins by generating n environments based on the `--procs` setting of the training command, see Appendix A for the parameter list. Each environment has the same configurations, for example `--grid-size`, `--agents` and `--agent-view-size`. Second, the amount of `--frames` is taken from the parameters, defining a general training loop, which ends once this number is reached or suceeded. During the loop, the defined training algorithm `--algo` is executed.

Since both algorithms have similar procedures, they share the base class. In it a second iteration is triggered, that iterates over the amount set for `--frames-per-procs`. During each step here, the agents execute actions in all parallel environments. Therefore, each step in the `--frames-per-procs` iteration produces a frame in every environment. The action selection of the learning algorithms generate different decisions based on the state of each environment.

During the inner iteration, data like rewards, observations, actions and more are stored in base class variables that are accessible by the learning algorithms. When this iteration is done, the base variables are reset and log values of all episodes that reached the done state are returned to be logged, as shown in Figure 4.3. The training loop keeps track of a frames counter, which sums up all frames that were produces by the parallel environments in the inner iteration. The training loop ends when the frames counter is greater or equal to `--frames`. Otherwise, more experience batches are gathered.

Both learning algorithms include their own action selection methods. The PPO implementation relies on an actor-critic neural network, with an action space containing a probability distribution. In case of the DQN implementation the target network assigns Q values to actions, and agents choose one based on the maximum value with an epsilon greedy probability. In both variants the action selection results in one action for each agent and for each environment.

Unlike the PPO algorithm, in the DQN approach a quadruple of information is saved each frame into a replay memory. The four parts of the quadruple consist of the executed actions during that time step, the returned rewards and both the previous and new observation of the parallel environments. Until the frame amount of `--initial-target-update` is reached, DQN agents only gather the quadruples but do not use them yet.

After exceeding the `--initial-target-update`, the DQN learning starts. Each frame a batch of size `--batch-size` is selected by randomly picking entries from the replay memory. Then, this batch is used to apply Q-learning updates to the experience samples, enhancing the training network. Every `--target-update` amount of frames this training network is copied into the target network to enhance the action selection while keeping the algorithm stable.

The action selection itself is also improved during the training, by decreasing the ϵ gradually through $\epsilon = \epsilon_{end} + (\epsilon_{start} - \epsilon_{end}) * e^{-\frac{frames}{decay}}$. This ensures exploration in the early phase. A high ϵ leads to actions that are picked at random. In the later course as the amount of frames increase, the ϵ gets smaller. In this case, the chance to select actions based on their Q values rises, which exploits the gathered experiences. Through `--epsilon-start` and `--epsilon-end` min and max values are set, and `--epsilon-decay` defines the speed of reduction.

In the DQN implementation learning happens during the base class batch creation, whereas in the PPO algorithm the learning process is triggered after the creation of each base class experience batch. Basically, every time the inner loop finished, the gathered values are reshaped and saved into a PPO experience buffer. Additionally, the advantage values are calculated here and added to the buffer.

With that buffer the PPO model is now optimized. A small number of `--epochs` are iterated and during each iteration random batch entries are selected. With those entries the entropies, values and losses are calculated. Afterwards, the calculation results are used to update the policy and network, as suggested in the code of 2.2.

4.4. Market Settings

To include a market into the training process, the `--market` parameter can be set accordingly. The user has a choice to include an AM through the string “am” or a SM with “sm”. In either case, the environment needs to adjust the action space, since agents now have the option to conduct market transactions.

Per default, the environment action space is discrete and only contains one element with five options: moving up, down, left, right or wait. Adding a market expands that

discrete space into a multi discrete space. Hence, both markets require actions in form of arrays that contain three elements. However, they use different information in the action array slots. This and further distinctions and detailed procedures of each market are explained separately in the following.

4.4.1. Shareholder Market

A coloring environment that includes a SM constrains the first position of the action array to one of the five environment actions. The next position contains an agent index, towards which a buying offer will be made. Although, if this number is higher than the amount of agents in the game, the action intends no buying transaction. The last array position contains either a zero or a one, with one signalizing that the agent wants to sell its share. An abstract representation of a shareholder action array is: `[environment_action, agent_index, sell_share]`.

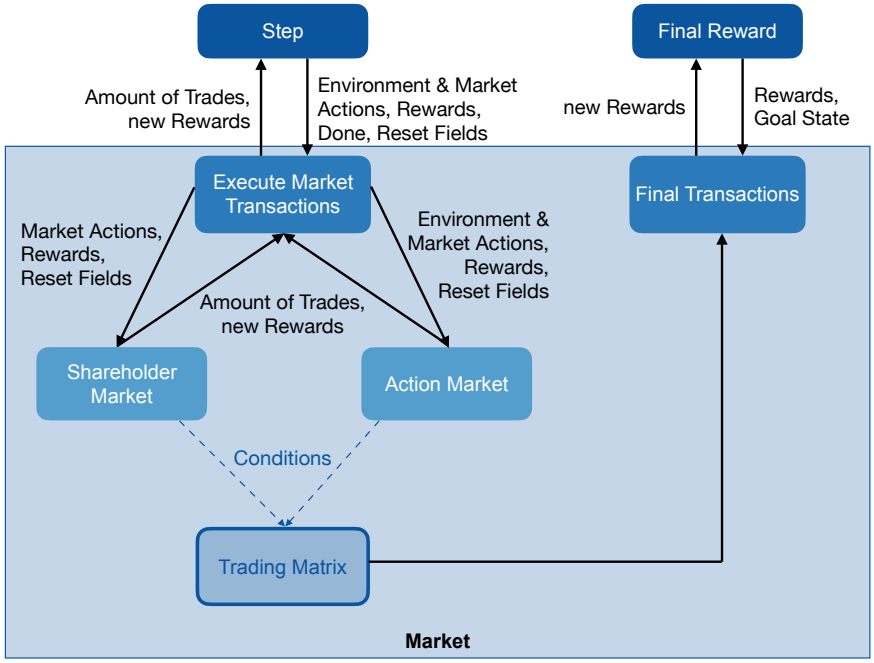


Figure 4.4.: The market elements

In Figure 4.4 market elements are visualized. On the left, the process that takes place during each step is shown, see algorithm 1 line 10. Here, the market always receives an action array that is already divided in two parts, one part only containing the first array position and the other part includes the buying and selling information in this case.

In the course of the market calculations a trading matrix is altered. This matrix is quadratic with dimensions equal to the amount of agents. In a shareholder trading matrix, the diagonal contains ones, since every agent starts with the full ownership over

their own shares. All other matrix slots are filled with zeros. An example for an initial trading matrix is shown below.

$$trading\_matrix = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

The first thing to check in a market step execution is the market type. The type can be set with the `--market` parameter and the market receives this setting during initialization. If the market type includes the string "sm" the SM matches and the corresponding function is called. Inside the shareholder function two additional matrices are created in each step, a buying matrix and a selling matrix. Both matrices are always initially filled with zeros, are quadratic, and their dimensions is equal to the amount of agents, similar to the trading matrix. Depending on the market action the buying matrix is altered to contain a one in the row of the buyer and the column of the agent that the offer is directed to. Each agent can only buy one share at a time, therefore here the rows contain at maximum one entry. The same applies to the selling matrix, which contain ones on the diagonal for the agent that wants to sell according to the market actions.

After setting up and filling out the two matrices they are iterated, extracting the rows entries of each matrix and defining the corresponding indices as buyer and seller. A transaction takes place if the following conditions are met:

- the buyer is not equal to the seller
- all entries of the buyer row and the seller row match
- the sellers shares are greater than the `--trading-fee`

If all conditions are true, the trading matrix is updated here, by changing the share of the selling agent, adding the subtracted amount to the buyer. The amount can be set with `--trading-fee`, which is 0.1 per default. The last condition ensures, that agents still receive some of their own rewards and do not trade everything off.

An example for a transaction could be two agents acting in an environment. If agent two buys a share from agent one, the trading matrix is updated. The second row stores the shares of agent two, which increases by 0.1 on the first position. This signalizes that agent two is owner of some shares of agent one and still has 100% of its own shares. In response, the shares in the first row and column of agent one decreases to 0.9.

$$trading\_matrix = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \xrightarrow[\text{from Agent 1}]{\text{Agent 2 buys}} trading\_matrix = \begin{pmatrix} 0.9 & 0 \\ 0.1 & 1 \end{pmatrix}$$

Additionally, the rewards of the current step calculation, would be updated here, if a price for the shares are set. In this implementation however, the shares have no price since agents are willing to give them away for free. Otherwise, the price would be subtracted from the buying agents' reward and that value would be added to the reward of the selling agent.

In any case, the SM triggers a reward calculation according to the trading matrix in every step. This means, that for the example above, agent two will receive 10% of the rewards from agent one in every step, until the episode ends. However, this is only the case in the steps in which agent one is not in dept.

Further details of the reward calculations in markets will be discussed in Section 4.4.3. Lastly, the transaction count is documented for evaluation purposes. At this point the market execution for the current step is done and the number of executed trades and the updated rewards are returned to algorithm 1.

4.4.2. Action Market

The agent action shape of an AM environment is similar to the shareholder action array. Again, an action has three slots, with the first being the environment execution and the second being the index of an agent a buying offer is directed to. The difference to a shareholder action is the last array position. Instead of setting a bit here to signalize the willingness of selling shares, the agent chooses an environment action that is expected from the agent of position two. Hence, an abstract representation of an action in the AM is the following: `[environment_action, agent_index, expected_action]`.

The market elements and general process of visualization 4.4 also apply to an AM setting. Here however, the trading matrix is initially filled with zeros. To establish a transaction the market actions are examined, and the following conditions checked:

- the buyer differs from the receiving agent
- the environment action of the receiving agent matches the expected action

When the two conditions apply a market transaction takes place. The `--trading-fee` parameter decides the price the buyer pays the receiving agent. Both the rewards and trading matrix are altered here, by subtracting the price from the buyer and adding it to the receiver. An example of the trading matrix update in this market setup is shown below. Here again agent two is purchasing from agent one.

$$trading\_matrix = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \xrightarrow[\text{from Agent 1}]{\text{Agent 2 buys}} trading\_matrix = \begin{pmatrix} 0 & 0.1 \\ 0 & -0.1 \end{pmatrix}$$

The trading matrix stores the market balance of both agents in each row. For agent two this means that the negative value was spent. The first row shows that agent one still has a neutral balance and gained the `--trading-fee` of 0.1 from agent two. To conclude, the market returns the number of transactions that took place in this step and the new rewards.

4.4.3. Reward Calculations

During each step agents can update the trading matrix by acting on the market. With each update the rewards are also changed. In Figure 4.5 a detailed example of the reward update in a market is shown. It is worth mentioning that it is not possible to execute

both markets simultaneously, but rather one or none must be set for a training process. This illustration shows both calculations in one image for convenience. Equal to the previous examples, the red agent two buys a share or action from the blue agent one. For both market scenarios the `--trading-fee` is set to the default value and both agent rewards start at 0.1. On the top half of the image the internal market calculation of a SM is shown, and the bottom half illustrates the calculation of an AM.

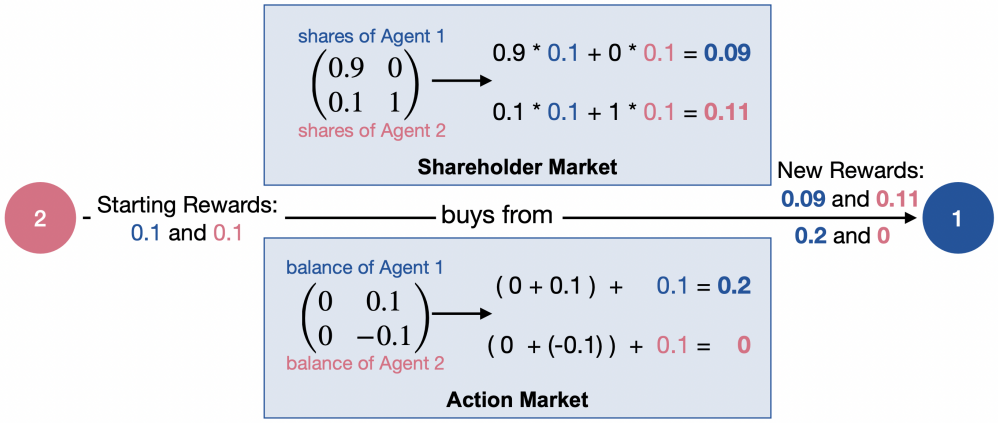


Figure 4.5.: Exemplary reward calculations of both market types

In a SM the trading matrix represents the distribution of agent shares. Each matrix column adds up to one, representing a 100% share of an agent. As mentioned earlier the diagonal of the matrix is initially set to one since agents start with complete ownership over their shares. For this example the trading matrices are configured accordingly. The first matrix row implies that agent one is owner of 90% of its own shares and is not owner of any shares from agent two. Whereas the second row shows that agent two claims 10% of the shares of agent one and has full ownership over its own.

To generate the new rewards of the agents, the market multiplies all current rewards with each matrix rows. The resulting products of each row are then summed up to represent the new reward of the corresponding agent. For the example, agent one gets a new reward value of 0.09 and agent two claims 0.01 from agent one and adds that to its full value of 0.1, resulting in a new reward of 0.11.

In contrast to an AM in a SM the rewards are just reassigned based on the current shares of the trading matrix. An exception occurs, when agents have negative rewards. In this case their share will be skipped during the redistribution, since shares are used to participate in profits.

Another difference to AMs is that the shown SM reward redistributions are executed in each step, and it is irrelevant whether market actions were executed. The only exception however is the last step, when the done flag is set to true. In this case the final rewards, see algorithm 2, need to be calculated first, before the shares are taken into account.

AMs, in most cases, update the rewards directly and only once, when a transaction is executed. The trading fee is immediately subtracted from one agents' reward and added

to the counterpart during the trade. If an agent can not afford the fee the process is completed anyway and the agent goes into debt.

Thus, this market type makes no use of the trading matrix. Nonetheless, the matrix is always updated, since a specific scenario requires the calculations to be executed at the end, see Section 4.4.4. In this case, the agents market balance, stored in the matrix rows, is summed up and added to the current reward. This procedure is illustrated in the bottom half of figure 4.4.3. Here, the fee of 0.1 is subtracted from agent two and added to the reward of agent one, leading to new rewards of 0.2 for agent one and zero for agent two.

4.4.4. Additional Conditions

The `--market` string for both types can be extended to add more conditions, namely with "no-reset", "no-debt" and "goal". The "no-reset" string enables the check whether the buyer has recently reset a cell. If that is the case, the corresponding buyers are ignored on the market for the current step. Hence, their market actions will not be applied. However, in a SM the penalized agent can still sell its shares.

With the "no-debt" Flag, transactions only take place if buyers can afford to pay the price. In this implementation with AMs and the default fee, this is solely the case if agents have colored a cell in that step. Waiting or misbehaving agents are excluded as buyers, since their rewards result in 0 or -0.1, which does not meet the fee of 0.1. For SMs this depends on the presence of a share price. Per default the price is zero, similar to the approach of Schmid et al. [SBM+21], making this condition irrelevant for the SM setting.

The last addition, "goal", lets the market process run as usual, only removing the reward changes during the steps. Here, all transactions are just documented into the trading matrix during an episode. Eventually, the transactions are executed once the final rewards of algorithm 2 is calculated. As shown on the right side of Figure 4.4, the market obtains those rewards and a Boolean describing whether the environment goal was reached.

The rewards are updated with the trading matrix content when either of the two conditions is satisfied:

- "goal" addition is present and environment goal was reached
- no "goal" addition and market type is a SM

Otherwise the rewards are return as they are and will not be processed further.

For such goal oriented markets, regardless of the type, the final environment state needs to equal the overall aim. Thus, the whole grid has to be colored, to execute the final market transactions.

If the first condition applies and an AM is present the rewards are updated by using the trading matrix. For a SM either condition must be met in order to generate the final reward. The calculations for both cases are equal to the example of Chapter 4.4.3.

After the final market updates to the rewards the new values are returned, as shown in Figure 4.4. The last thing to point out is that the additional market conditions can be

used in combination, making “sm-goal-no-reset-no-debt” for example a valid `--market` setting.

5. Results

In this Chapter the results of various training executions with different parameters are compared. All possible combinations of markets, agent compositions and learning algorithms in an easy environment setup are shown in Section 5.2. The best combinations are then extracted and applied in more challenging environments, which are compared in Section 5.3 and 5.4. Furthermore, Appendix B shows all generated training plots of the presented results.

5.1. Setup

was wurde untersucht

The results of this research compare multiagent trainings with varying settings, namely acting in different compositions and markets. In this case the amount of agents stays fixed and is greater than one. The agents use either of the two learning approaches PPO or DQN to train. Hence, the overall possible comparisons include a total amount of 102 executions. This number results from the calculation of multiplying the two learning algorithms with the three possible agent compositions (cooperative, competitive and mixed-motive) and additionally two optional markets that can contain three modular additions.

The market options are for example the following SM instances:

- “sm”
- “sm-goal”
- “sm-no-debt”
- “sm-no-reset”
- “sm-no-reset-no-debt”
- “sm-goal-no-debt”
- “sm-goal-no-reset”
- “sm-goal-no-reset-no-debt”

The 8 options above are also applied on the AM and lastly the option of no market needs to be considered, leading to 17 market scenarios. Calculating the total amount of executions now with those 17 market possibilities results in the 102 executions.

was wurde nicht untersucht

Yet, not all market arrangements are needed. For example, the mix of “no-reset” and “no-debt” is not of use in this implementation. An agent that has reset a field has a reward of -0.1 and therefore already is in debt, which means that “no-debt” includes “no-reset”. This subtracts four compositions from the 17 market scenarios. Additionally, shares are free of charge, making the options “sm-no-debt” and “sm-goal-no-debt” irrelevant. Agents can always afford to buy shares in this case. The SM is therefore left with four combinations and the overall market scenario count is now 11:

- “am”
- “am-goal”
- “am-no-debt”
- “am-no-reset”
- “am-goal-no-debt”
- “am-goal-no-reset”
- “sm”
- “sm-goal”
- “sm-no-reset”
- “sm-goal-no-reset”

wie wurde untersucht / zahlen

In total the analyzation now only includes 66 training results.

In order to compare the market approach with a credit assignment solution in the cooperation composition, the training with a DR setting is also included. This in turn adds another execution to each learning algorithm. Furthermore, to ensure that the environment is generally solvable, one agent first trains in the environment setup with each learning algorithm using similar hyperparameter as in the multiagent case. The execution count is therefore 70 in total.

For an easy environment, those 70 executions are mostly run with the default parameters that can be looked up in Appendix A. The agents solve an empty five by five grid here, in which they can only walk inside a three by three area, due to the surrounding walls, see Figure 5.1. The maximum amount of steps the agents are allowed to take is set to 25, if not manually specified otherwise. This count is generated by squaring the grid size. For the one agent execution the step size is reduced to 10 and the amounts the agent managed to color the whole grid is shown in table 5.1.

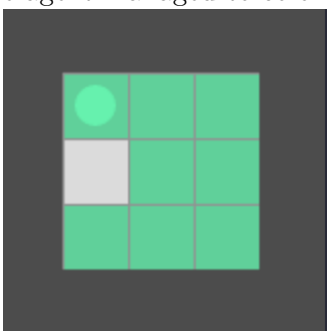


Figure 5.1.: Visualization of a small environment with one agent

Setting	Fully colored
1 ppo	3383
1 dqn	650

Table 5.1.: Number of times the agent fully colored the environment during training with each learning algorithm.

Here, the amount of times the grid on the left (Figure 5.1) was fully colored by one agent using each learning algorithm during approximately 80.000 `--frames` is shown. The PPO agent colored the whole grid a total amount of 3383 times and the DQN agent 650 times.

The average grid coloration percentage during the training is shown in plot 5.2. The

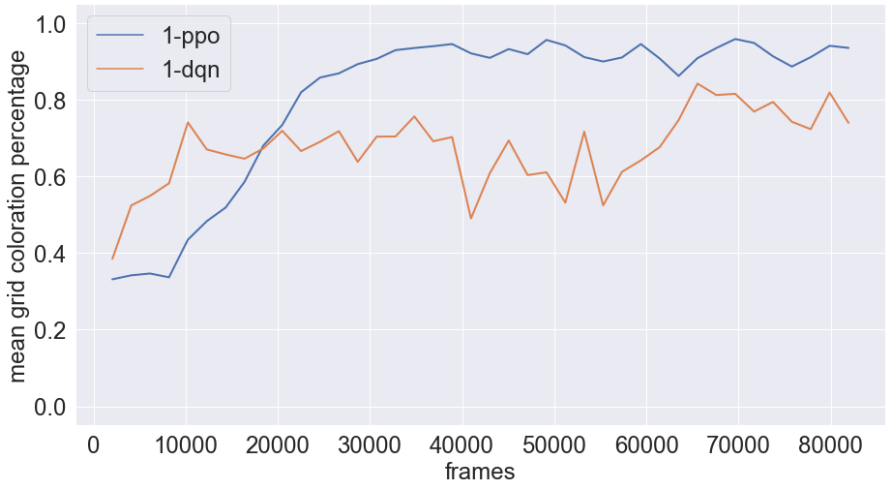


Figure 5.2.: The mean coloration percentage in a 5x5 grid and one acting agent

plot lines start at around 2048 frames, since this marks the first time an entry of the data produced by the parallel environments is logged. This specific number originates from the number 128 of the parameter `--frames-per-proc` with `proc` referencing the parallel environments. This in turns means, that 128 steps in 16 environments are executed, which produces 2048 frames in total. After each 2048 executed steps all completed episodes are summarized and documented. To do so, average values of important data are calculated.

Here for instance, the average grid coloration percentage from all completed episodes of all environments are used to generate the mean values displayed. For the DQN line this means, that the first entry at 2048 frames contains a mean grid coloration of 40% in the finished episodes. The only exception is the amount of achieving a fully colored grid, which is just counted in the episodes of the 2048 frames. In the table 5.1 the total sum of the counter is displayed.

Furthermore, the plot exceeds the 80.000 default `--frames`, since the last data entry includes the last summary of 2048 frames. The table shows, that both training runs solve the environment and the plot illustrates that in both cases generally a high percentage of the grid is colored. The DQN execution yields a better performance in the early stage, whereas the PPO agent gradually improves over time. In both cases an average coloration of over 70% is eventually reached. This concludes that the grid is solvable with the parameters set.

5.2. Easy Environment

Now the multiagent scenario is compared. Here a set of two agents are trained to color the field of the same default dimensions, see Figure 5.3. However, every agent executes an action during a step and with 10 steps and two agents in theory 20 cells could be

visited. Hence, the `--max-steps` count is reduced to eight. This should leave enough space for agents to make mistakes and still have a chance to solve the grid.

In order to get an overview of the overall 70 training results, the remaining 68 multiagent executions are divided into three different `--setting` values. The first data division only covers the cooperation compositions, including "difference-reward". Table 5.2 illustrates the scoreboard of the top three executions in this specification for each learning algorithm.

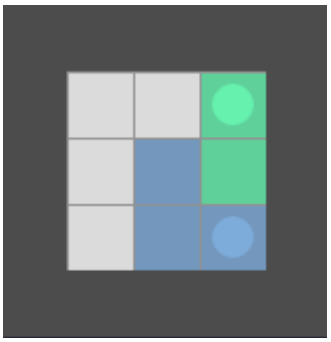


Figure 5.3.: Visualization of a small environment with two agents

	Top PPO Cooperation Settings	Fully colored
1	cooperation difference-reward	1683
2	cooperation	1109
3	cooperation sm-goal-no-reset	533

	Top DQN Cooperation Settings	Fully colored
1	cooperation difference-reward	5949
2	cooperation am-goal-no-debt	3197
3	cooperation am-goal	2880

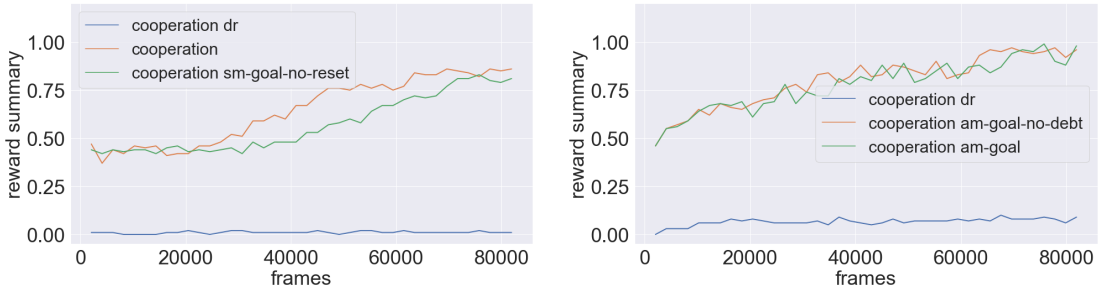
Table 5.2.: Number of times two agents working in cooperation fully colored the environment during training.

The measured attribute here is again the overall amount of fully colored achievements. In both cases the agents with the DR setting scored best, 1683 times with the PPO algorithm and 5949 times by using DQN. The PPO scores continue with the default cooperation scenario on second and the SM with the "goal-no-reset" condition on third place.

On the contrary, the DQN results show AM settings on the remaining places, with the additions "goal-no-debt" on second place and "goal" on the third place. It is visible, that in both scoreboards the second and third executions are far behind the corresponding DR setting in terms of fully coloration counts.

In the two plots of Figure 5.4 the reward summary of the top scores, see table 5.2, are displayed. The term reward summary is used here, since the rewards of cooperating agents are equal and sometimes contain only slight changes through markets. However, in other agent compositions the rewards are rather specific to each agents' contribution. In any case the logged training data contains the mean reward of every agent separately.

In order to summarize cooperation rewards, the average value of those separate agent rewards is calculated for each data entry and the results are then plotted here. For reward summaries of all other compositions each data entry is summarized with the sum of the separate agent rewards. The maximum y-axis label is set to 1.2, since agents get a reward of one if they color the whole field and additionally could get a reward of 0.1 for the final step. Through the reward summary calculations rounding errors may occur,



(a) Reward summary of PPO agents

(b) Reward summary of DQN agents

Figure 5.4.: Reward Summaries of the top three cooperation compositions using PPO (left) and DQN (right)

which could in turn exceed the maximum reward of 1.1. Furthermore, markets could also contribute to bigger rewards.

Even thought, the executions with the DR configuration scored highest in terms of reaching the goal, the reward lines in this case stay around 0.1. The reason for that is, that agents get the difference of two rewards, leading to very small values. All rewards, except the DRs, show a continuous increase and at least reach a summary reward of 0.8.

The next training executions to look at are "mixed-motive" settings. Here again a scoreboard, listing the top three results of each learning algorithm, is shown in table 5.3.

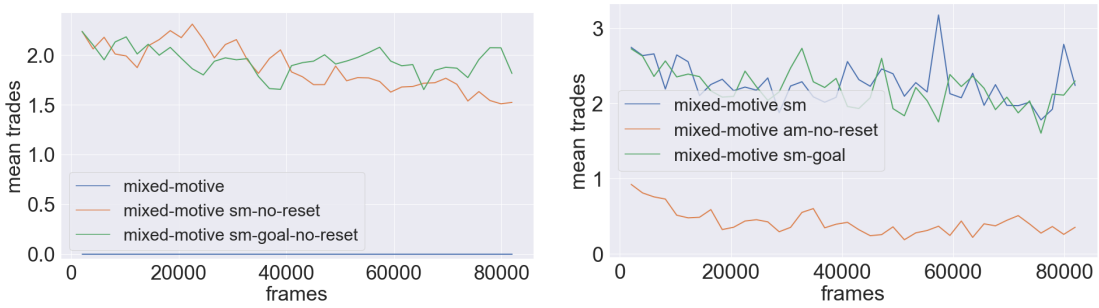
	Top PPO Mixed-Motive Settings	Fully colored
1	mixed-motive	1734
2	mixed-motive sm-no-reset	1422
3	mixed-motive sm-goal-no-reset	1377

	Top DQN Mixed-Motive Settings	Fully colored
1	mixed-motive sm	5417
2	mixed-motive am-no-reset	5379
3	mixed-motive sm-goal	5302

Table 5.3.: Number of times two agents working in a mixed-motive setting fully colored the environment during training.

The PPO scoreboard shows, that the plain "mixed-motive" setting is on the top with a score of 1734. Subsequently, SM configurations follow, with the addition of "no-reset" on second and "goal-no-reset" on third place. The DQN scores also show two SM settings, the default market occupies the first place here and the SM with the "goal" addition is on the last place, both however colored the grid a total of over 5300 times. The AM with the "no-reset" condition is on the second place in the DQN statistics.

In Figure 5.5 two plots are shown, visualizing the trading behavior of the agents during training. The plots display the mean amount of trades in the parallel environments. In plot 5.5a the blue line stays at zero, since this execution does not contain a market



(a) Mean trades of PPO agents

(b) Mean trades of DQN agents

Figure 5.5.: Mean trades of the top three mixed-motive compositions using PPO (left) and DQN (right)

extension. The other two lines are slightly decreasing and move between 2.2 and 1.5. Both lines here represent a SM interaction with the values showing that on average two market transactions occurred. This in turn means, that about two shares were bought in the completed episodes of all parallel environments.

In the right plot 5.5b an AM is represented with the orange line. Here the mean trades are fewer compared to the other two SM lines. This is often the case for the two Markets, since action purchases subtract a bigger value at once, whereas selling shares reduces the payout over time. Comparing the SM executions between PPO and DQN shows, that the restriction of "no-reset" lowers the average trades to around two, whereas without this specific condition the trade count is between two and three.

The last dataset division only includes executions of competitive agent compositions. Table 5.4 shows the scoreboards of this setup.

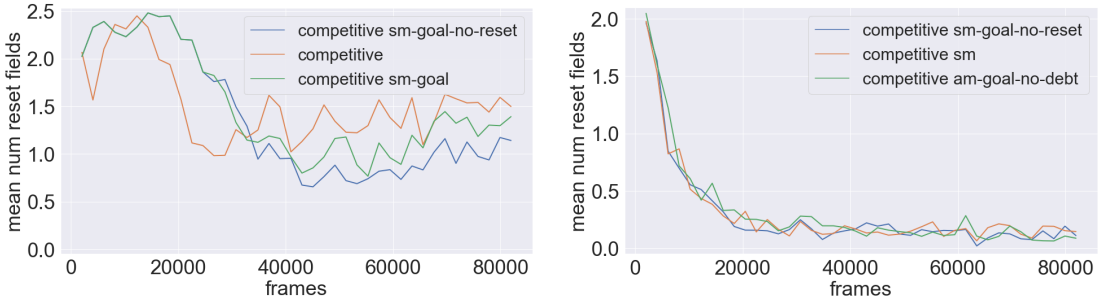
	Top PPO Competitive Settings	Fully colored
1	competitive sm-goal-no-reset	3614
2	competitive	3461
3	competitive sm-goal	3225

	Top DQN Competitive Settings	Fully colored
1	competitive sm-goal-no-reset	7877
2	competitive sm	7842
3	competitive am-goal-no-debt	7560

Table 5.4.: Number of times two agents working in a competitive setting fully colored the environment during training.

For both learning algorithms the most fully colored grids result from executions specifying a SM with the condition "goal-no-reset". In the DQN board follows the execution using the standard SM and the last place here is occupied by an AM with the specification "goal-no-debt". For PPO however the second place is the normal competitive

mode without any additions and on third place is a SM execution with the "goal" restriction. Overall, the score differences within the tables are not big, at most a difference of around 300 goal states is observed. Furthermore, The scores of the first places in these two boards are the best values achieved so far, with 3614 fully colored grids in the PPO table and 7877 in the DQN table.



(a) Mean number of reset fields of PPO agents **(b)** Mean number of reset fields of DQN agents

Figure 5.6.: Mean number of reset fields of the top three competitive compositions using PPO (left) and DQN (right)

The plots to those scoreboards are displayed in Figure 5.6. In this case the average amount of reset fields are visualized. All lines start at a value two, which means, that during the first 2048 frames an average of two cells are reset during the completed episodes. In both cases the graphs eventually drop, however for PPO execution this takes around 20.000 frames and in the later half a small increase in all executions can be observed. Meanwhile, the three DQN trainings rapidly reduce the average resets to around 0.3. This value is never reached by PPO executions, here the lowest score is approximately 0.6.

5.3. Difficult Environment

To see how the results change, when it becomes more challenging to reach the goal, all top executions showed before are now repeated in a bigger environment. The grid size is increased to seven which provides an area of 25 cells for the agents to color. Also, since the grid offers more room the amount of agents is increased to three.

To give the agents more time to learn the `--frames` are set to 200.000 and the value of `--frames-per-proc` is changed to 256. With the increase to 256 the training data entries are now 4096 frames apart instead of the 2048 we had before. The number 4096 comes from multiplying the number of environments with the `--frames-per-proc`. To successfully solve the grid with one agent some DQN specific parameters needed to be tuned. As a result the adjusted values of `--replay-size`, `--epsilon-decay` and `--target-update` are 700.000, 20.000 and 10.000.

In Figure 5.7 the one agent learning scenario during training in a difficult environment setup is shown. Here, the agent view size is also noticeable, with the lighter floor and

wall colors three tiles around the agent. In this case the agent can not see the two top rows of the grid. This of course also contributes to the difficulty of this setup.

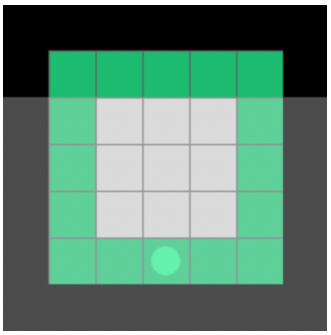


Figure 5.7.: Visualization of a 7x7 Environment with one agent

Setting	Fully colored
1 ppo	2924
1 dqn	394

Table 5.5.: Number of times the agent fully colored the environment during training with each learning algorithm.

The results of the training executions with one agent are shown in table 5.5. In this case the maximum step amount is again the default value of 49 here, since the grid size of seven is squared. Similar to the easier setup, the PPO training yield more fully colored grids compared to the DQN execution. While the PPO agent reaches the goal 2924 times the DQN agent only achieves 394 goal states. Comparing those numbers with their corresponding easy equivalent it is also obvious that for both learning algorithms the scores have decreased.

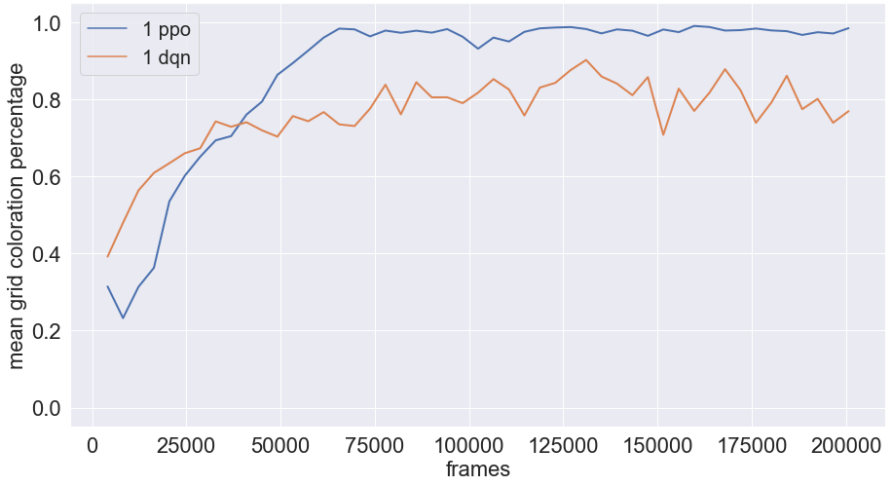


Figure 5.8.: The mean coloration percentage of a 7x7 grid and one acting agent

Looking at the mean coloration percentage plot 5.8 however, both executions show an increase to a relatively high percentage. The DQN agent training reaches an average of 80% grid coloration and deviates only slightly around it. In comparison, the training

with PPO has a steep incline from 20 to 95% and after 60.000 frames this high amount is maintained until the end of training. This proves that both algorithms learned strategies to color the environment with the set parameters.

For the training in this setting with three agents, the maximum step amount is set to 20. In Figure 5.9 a training frame is visualized. In this time step the observation of the top agent does not include the two agents on the bottom. The two agents in turn also can not see the top agent but are aware of their neighbor.

The results of the cooperation executions with three agents are listed in table 5.6.

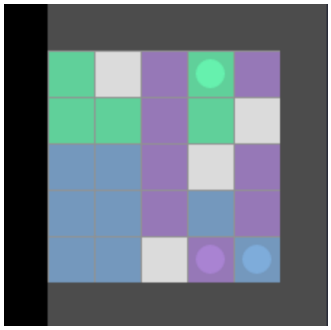


Figure 5.9.: Visualization of a 7x7 environment with three agents

	Top PPO Cooperation Settings	Fully colored
1	cooperation difference-reward	166
2	cooperation	10
3	cooperation sm-goal-no-reset	0

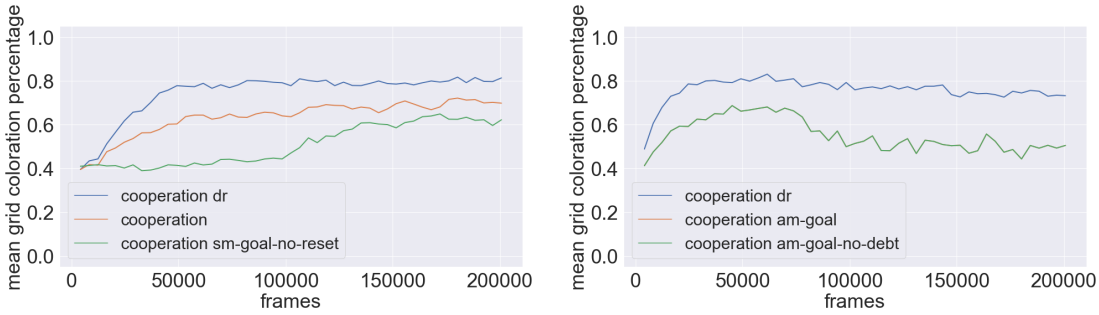
	Top DQN Cooperation Settings	Fully colored
1	cooperation difference-reward	115
2	cooperation am-goal-no-debt	0
3	cooperation am-goal	0

Table 5.6.: Number of times three agents working in cooperation fully colored the environment during training.

This table shows, that all cooperative setting with specified markets yielded no goal achievements. In the DQN scoreboard only the DR approach resulted in 115 complete colorations. The PPO table also lists the DR training result as first place with a total score of 166. On second place comes the standard cooperation training here, but only with 10 fully colored counts. Overall, the scores have significantly decreased in comparison to the small environment results.

In the plots of Figure 5.10 it is visible that both DR trainings result in a similar average coloration percentage of around 80%. In the DQN plot only two lines are visible, due to the “goal” addition of the market execution. Since the goal is never reached the final market payout with the trading matrix could not be applied. Hence, both trainings yielded the same decisions and overlap in this graph.

Continuing with the mixed-motive executions, table 5.7 shows the goal achievements.



(a) Mean coloration percentage of PPO agents **(b)** Mean coloration percentage of DQN agents

Figure 5.10.: Mean grid coloration percentages of the top three cooperation compositions using PPO (left) and DQN (right)

	Top PPO Mixed-Motive Settings	Fully colored
1	mixed-motive	247
2	mixed-motive sm-goal-no-reset	133
3	mixed-motive sm-no-reset	66

	Top DQN Mixed-Motive Settings	Fully colored
1	mixed-motive sm-goal	116
2	mixed-motive sm	42
3	mixed-motive am-no-reset	31

Table 5.7.: Number of times three agents working in a mixed-motive setting fully colored the environment during training.

The PPO board still has the default mixed-motive composition on first place, with 247 fully colored counts. Comparing this board with the results of the smaller environment it is clear that the next two position have swapped places. Also in the DQN board the order has changed. The previous last placed “sm-goal” setting is now on the first place with 116 colorations and the other two executions moved down.

Figure 5.11 displays the two plots that show how the fully coloration numbers are generated. On the left-hand side the PPO based trainings show an increase of reaching the goal state towards the end, with a maximum of around 14. This means, that in a data point, which extracts over 4096 frames and 16 parallel environments 14 episodes ended with the grid fully colored. Looking at the DQN plots it is evident, that the highest point of the learning curve lies at approximately 60.000 frames and then level off.

(a) Fully colored grid achievements of PPO agents **(b)** Fully colored grid achievements of DQN agents

Figure 5.11.: Environment goal achievements of the top three mixed-motive compositions using PPO (left) and DQN (right)

	Top PPO Competitive Settings	Fully colored
1	competitive	277
2	competitive sm-goal	165
3	competitive sm-goal-no-reset	111

	Top DQN Competitive Settings	Fully colored
1	competitive sm-goal-no-reset	367
2	competitive sm	240
3	competitive am-goal-no-debt	198

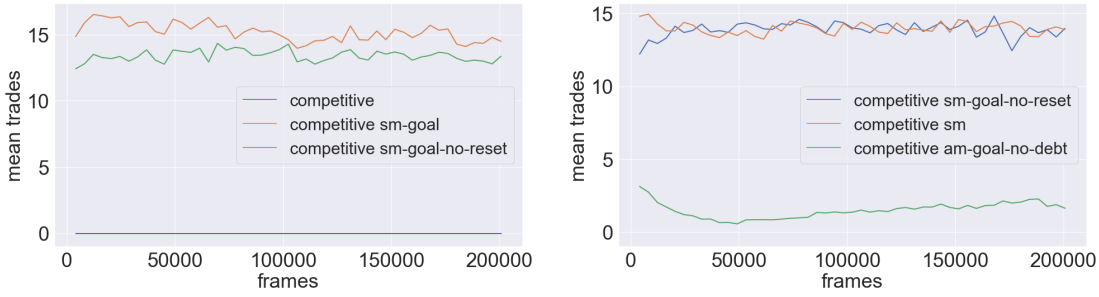
Table 5.8.: Number of times three agents working in a competitive setting fully colored the environment during training.

Finally, the last table 5.8 shows the top scores of competitive compositions in a seven by seven coloration environment. While up to this point the first places of PPO executions in the difficult setting resulted in higher scores than their DQN counterparts, in this table it is not the case. Here the Top competitive DQN training resulted in 367 completely colored fields, whereas the highest PPO score is only 277. Furthermore, the order of the DQN board stayed the same, while the PPO scores changed, with the previous first placed "sm-goal-no-reset" now being on the last place.

The last thing to point out here, are the mean trade counts in those trainings. Plot 5.12a and 5.12b display how many market transactions on average where executed. The two plots show similarities of building the strategy to sell around 15 shares on average. For the AM in the DQN execution the mean amount is only at approximately 3 trades.

5.4. Room Divided Environment

To further stress test the learning abilities of agents the environment is now also divided into rooms. In order to set up a reasonable room division the environment size needed to be increased to a 9x9 grid. A visualization of this setting is shown in Figure 5.13. or



(a) Mean trades of PPO agents

(b) Mean trades of DQN agents

Figure 5.12.: Mean trades of the top three competitive compositions using PPO (left) and DQN (right)

the training with one agent the maximum step count is again the default option which results in 81 here. Other than changing the `--env` parameter to "FourRooms-Grid-v0" and the grid size to 9 the same parameters of the previous seven by seven settings were used.

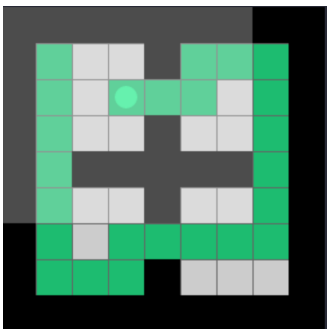


Figure 5.13.: Visualization of an environment with rooms and one agent

Figure 5.14.: The mean coloration percentage in a 9x9 Rooms Environment and one acting agent

Figure 5.14 replaces the table, since both trainings never achieved to fully color the grid. However, the plot shows potential, since an increase from 20% coloration to around 70% can be observed here.

For the multiagent training with three agents the step amount is set to 30 since the grid only offers 40 cells to color. A visualization of this setup is shown in Figure 5.15.

Furthermore, just the first places of the scoreboards of Chapter 5.3 were trained in this environment, due to long execution times. As a result these six settings are analyzed here, for PPO:

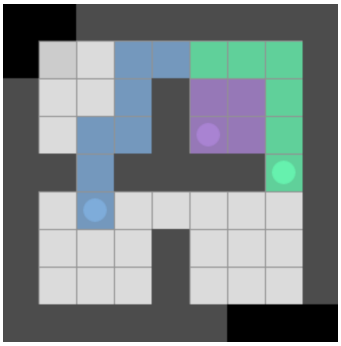


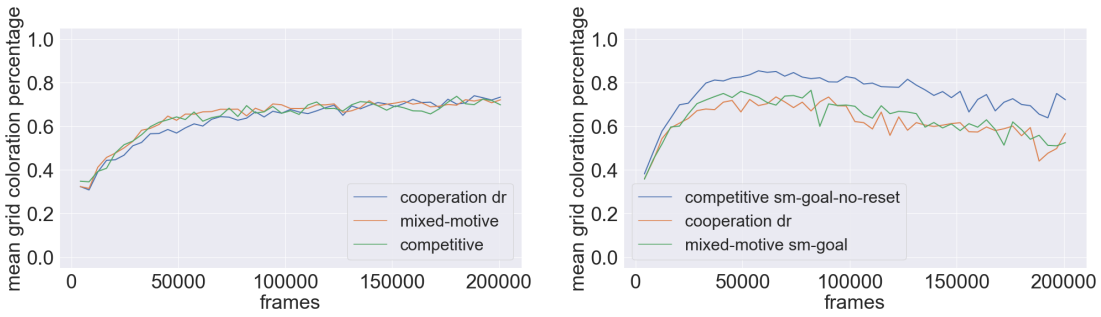
Figure 5.15.: Visualization of an environment with rooms and three agents

- cooperation dr
- mixed-motive
- competitive

and for DQN:

- cooperation dr
- mixed-motive sm-goal
- competitive sm-goal-no-reset.

Unfortunately, all but one of the training executions never fully colored the environment. Only the top DQN competitive training managed to reach the goal state a total of 10 times. In Figure 5.16a it is shown that the PPO agents learn to color 70% of the grid over time. In the DQN plot however, they reach the peak of 80% early on and drop the coloration percentage slowly afterwards, with the lowest point in the DR setting to 40%.



(a) Mean coloration percentage of PPO agents

(b) Mean coloration percentage of DQN agents

Figure 5.16.: Mean grid coloration percentages in a 9x9 Rooms Environment using PPO (left) and DQN (right)

6. Discussion

- Are the findings as expected?
- Why are the things as they were observed?
- New experiments that provide further insights
- Make your results more comprehensible

6.1. Easy Environment Results

6.2. Difficult and Rooms Environment Results

7. Conclusion

(Briefly summarize your work, its implications and outline future work)
- What have you done?
- How did you do it?
- What were the results?
- What does that imply?
- Future work

Each of the three compositions presented in Chapter 4.1 lead to learning problems or game losses. Cooperation may reward misbehavior, namely field resetting, leading to the CAP of Chapter 3.1. In mixed-motive or fully competitive settings the overall goal may be never reached due to greediness or disorder. This research further compares the effects of markets not only on competitive settings as suggested by Schmid et al. [SBM+21], but rather on all three configurations.

Appendix A.

Training Parameters

```
required arguments:
  --algo ALGO           Algorithm to use for training. Choose between 'ppo'
                        and 'dqn'.

optional arguments:
  -h, --help            show this help message and exit
  --seed SEED           Generate the same set of pseudo random constellations,
                        colors, positions, etc. every time the algorithm is
                        executed. (default: 1)
  --agents AGENTS       Amount of agents. (default: 2)
  --model MODEL         Path of the model inside the storage folder, if none
                        is given then a random name is generated. (default:
                        None)
  --capture CAPTURE     Boolean to enable capturing of the environment. The
                        outcome are in form of gifs. (default: True)
  --env ENV             Environment ID, choose between Empty-Grid-v0 for an
                        empty environment and FourRooms-Grid-v0 for an
                        environment divided into equal sized rooms. (default:
                        Empty-Grid-v0)
  --agent-view-size AGENT_VIEW_SIZE
                        Grid size the agent can see. Agent Observation is
                        based on that field of view. For example, 7x7 grid
                        size means agent can see three tiles in each
                        direction. (default: 7)
  --grid-size GRID_SIZE
                        Size of the environment grid. (default: 5)
  --max-steps MAX_STEPS
                        Maximum amount of steps an agent has to reach a goal.
                        If none is given then this max count is set to: grid
                        size * grid size. (default: None)
  --setting SETTING     Setting can be either: '' for cooperation, 'mixed-
                        motive' for a mixed motive environment, 'mixed-motive-
                        competitive' for a competitive composition or
                        'difference-reward' for a setting that calculates
                        difference rewards. Cooperation means all agents get
                        the same reward. If set to mixed-motive or mixed-
                        motive-competitve the reward is not shared and each
                        agent is responsible for its own success. In
```

```
                        competitive mode, agents can take over opponent
                        coloration without resetting the cells, otherwise
                        cells are always reset when colored and walked over.
                        The last option 'difference-reward' is a cooperation
                        setting but calculates the reward for each agent by
                        subtracting a new reward from the total reward. The
                        new reward just excludes the action of this one agent.
                        A high difference reward means, that the action of
                        that agent was good. (default: '' for cooperation)
  --market MARKET       There are three options: 'sm', 'am' and '' for none.
                        SM = Shareholder Market where agents can sell or buy
                        shares on the market. AM = Action Market where agents
                        can buy specific actions from others. (default = '')
  --trading-fee TRADING_FEE
                        If a market transaction is executed, this value
                        determines the price, i.e. in an action market this
                        defines the price the buyer pays. In a shareholder
                        market this value defines the share value. (default:
                        0.1)
  --frames FRAMES       Number of frames of training. (default: 80.000)
  --frames-per-proc FRAMES_PER_PROC
                        Number of frames per process. In case of PPO this is
                        the number of steps, before the model is optimized.
                        (default: 128)
  --procs PROCS         Number of processes/environments running parallel.
                        (default: 16)
  --recurrence RECURRENCE
                        Number of time-steps the gradient is back propagated.
                        If it is greater than one, a LSTM is added to the
                        model to have memory. (default: 1)
  --batch-size BATCH_SIZE
                        Batch size that is used for sampling. (default: 64)
  --gamma GAMMA         Discount factor with 0 <= gamma < 1, specify how
                        important future estimated rewards are. High value
                        means high importance. (default: 0.99)
  --log-interval LOG_INTERVAL
                        Number of frames between two logs. (default: 1)
  --save-interval SAVE_INTERVAL
                        Number of times the --frames-per-proc amount of frames
                        needs to be reached, to log the current training
                        values, i.e. rewards, into a csv file. (default: 10, 0
                        means no saving)
  --capture-interval CAPTURE_INTERVAL
                        Number of times --frames-per-proc amount of frames
                        needs to be reached, to capture the last --capture-
                        frames amount of steps into a gif. Warning: --capture
                        needs to be set to True as well. (default: 10, 0 means
                        no capturing)
  --capture-frames CAPTURE_FRAMES
```

```
                      Number of frames that are captured. (default: 50, 0
                      means no capturing)
--lr LR               Learning rate. (default: 0.001)
--optim-eps OPTIM_EPS
                      Epsilon value for the Adam optimizer. (default: 1e-8)
--epochs EPOCHS       [PPO] Number of epochs for PPO optimization. (default:
                      4)
--gae-lambda GAE_LAMBDA
                      [PPO] Lambda coefficient in GAE formula, used for
                      calculation of the advantage values. (default: 0.95, 1
                      means no gae)
--entropy-coef ENTROPY_COEF
                      [PPO] Entropy term coefficient. (default: 0.01)
--value-loss-coef VALUE_LOSS_COEF
                      [PPO] Value loss term coefficient. (default: 0.5)
--max-grad-norm MAX_GRAD_NORM
                      [PPO] Maximum norm of gradient. (default: 0.5)
--clip-eps CLIP_EPS   [PPO] Clipping epsilon for PPO. (default: 0.2)
--epsilon-start EPSILON_START
                      [DQN] Starting value of epsilon, used for action
                      selection. (default: 1.0 -> high exploration)
--epsilon-end EPSILON_END
                      [DQN] Ending value of epsilon, used for action
                      selection. (default: 0.01 -> high exploitation)
--epsilon-decay EPSILON_DECAY
                      [DQN] Controls the rate of the epsilon decay in order
                      to shift from exploration to exploitation. The higher
                      the value the slower epsilon decays. (default: 5.000)
--replay-size REPLAY_SIZE
                      [DQN] Size of the replay memory. (default: 40.000)
--initial-target-update INITIAL_TARGET_UPDATE
                      [DQN] Frames until the target network is updated,
                      Needs to be smaller than --target-update! (default:
                      1.000)
--target-update TARGET_UPDATE
                      [DQN] Frames between updating the target network,
                      Needs to be smaller or equal to --frames-per-proc and
                      bigger than --initial-target-update! (default: 15.000)
```

Appendix B.

Detailed Results

B.0.1. Easy Environment

The following plots show all details of the best training results in a small 5x5 grid. The default parameters of Appendix A are used for all executions. Only the agent amount, setting and market value change. In the first Figure B.1, one agent is acting in the environment, all other trainings are executed with two agents. For details see Chapter 5. An example to run a training process is shown below.

```
$ python -m scripts.train
    --algo ppo
    --model ppo-easy
    --agents 2
```

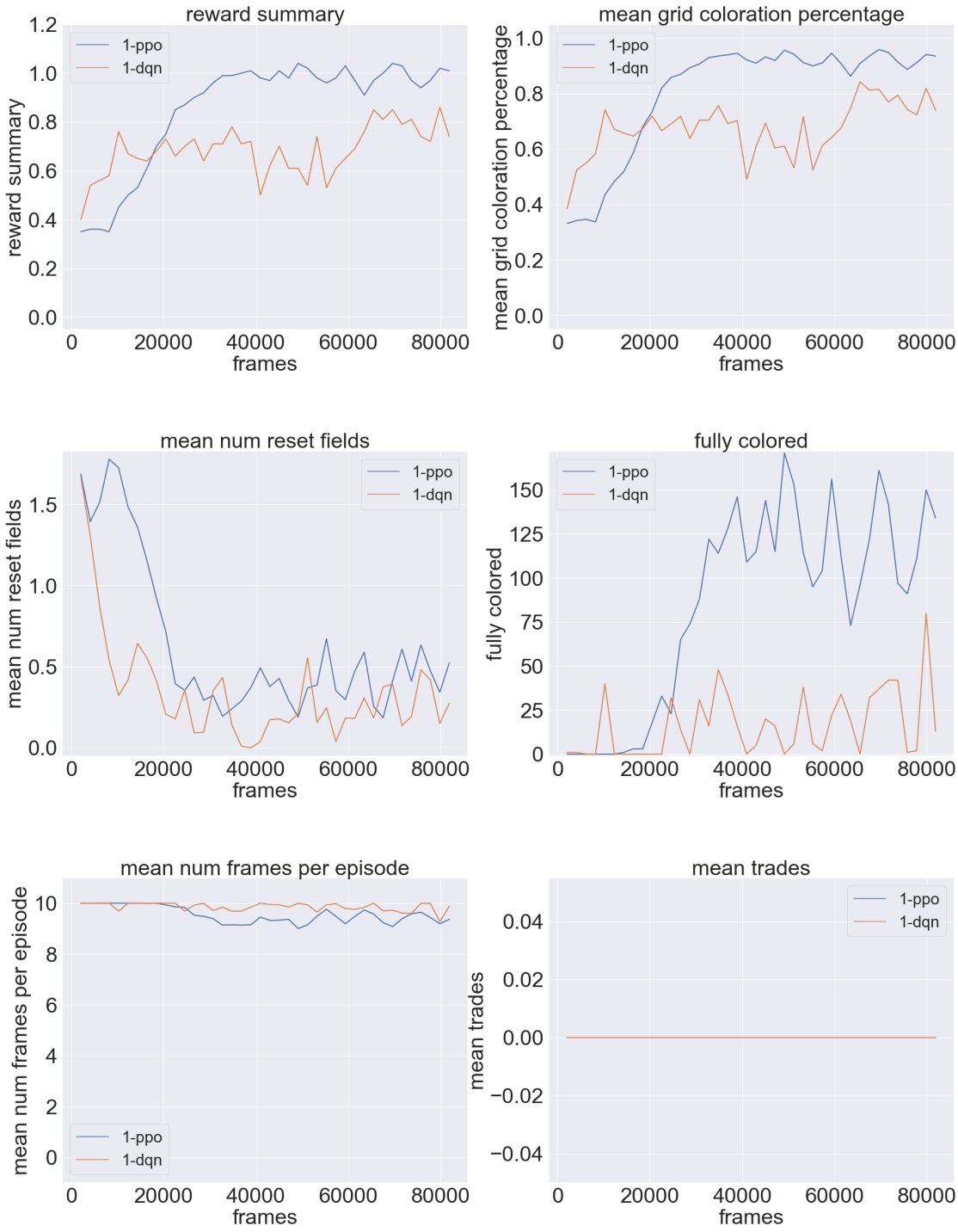


Figure B.1.: Details of the training with one agent using PPO and DQN

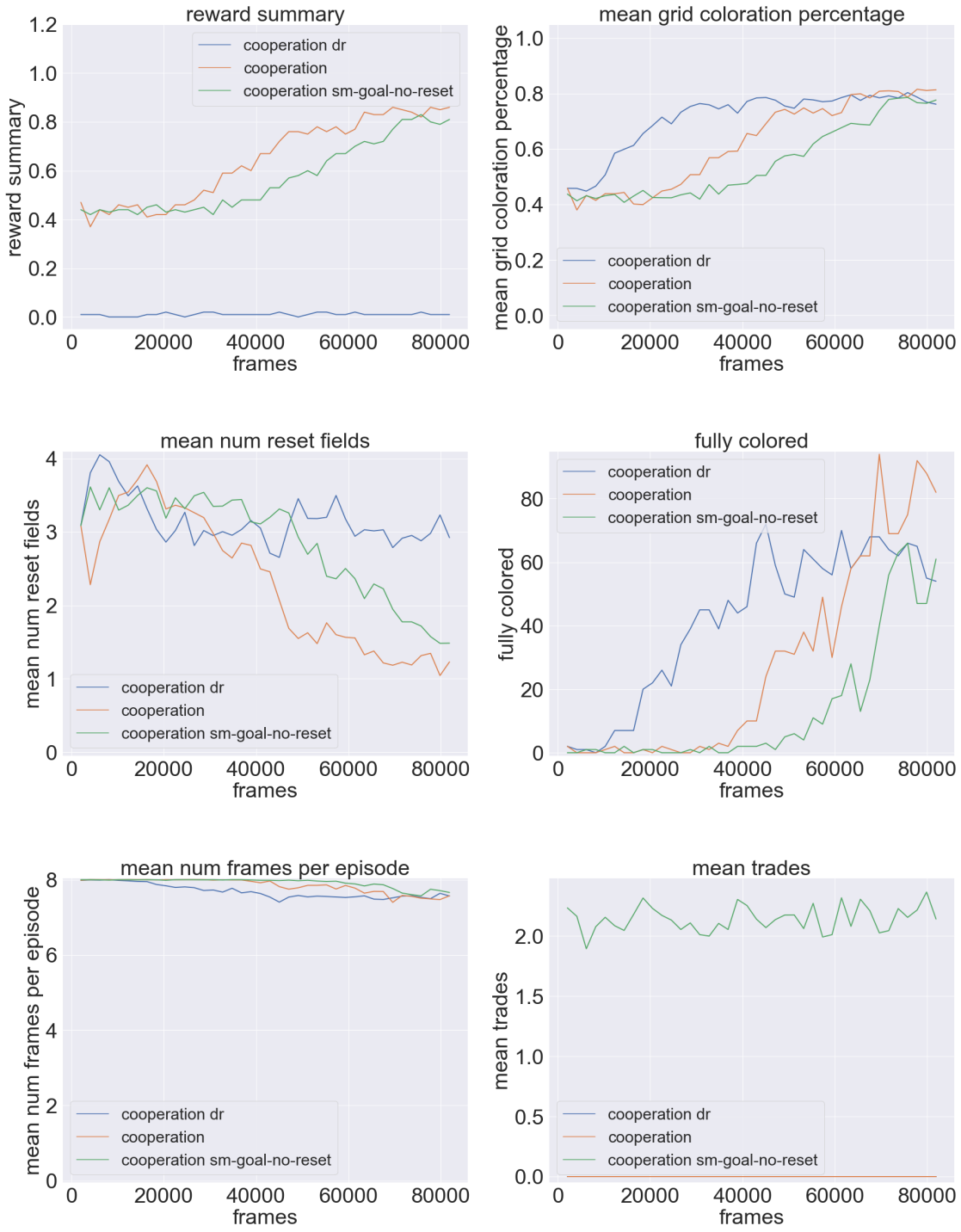


Figure B.2.: Top cooperation score details of two PPO agents

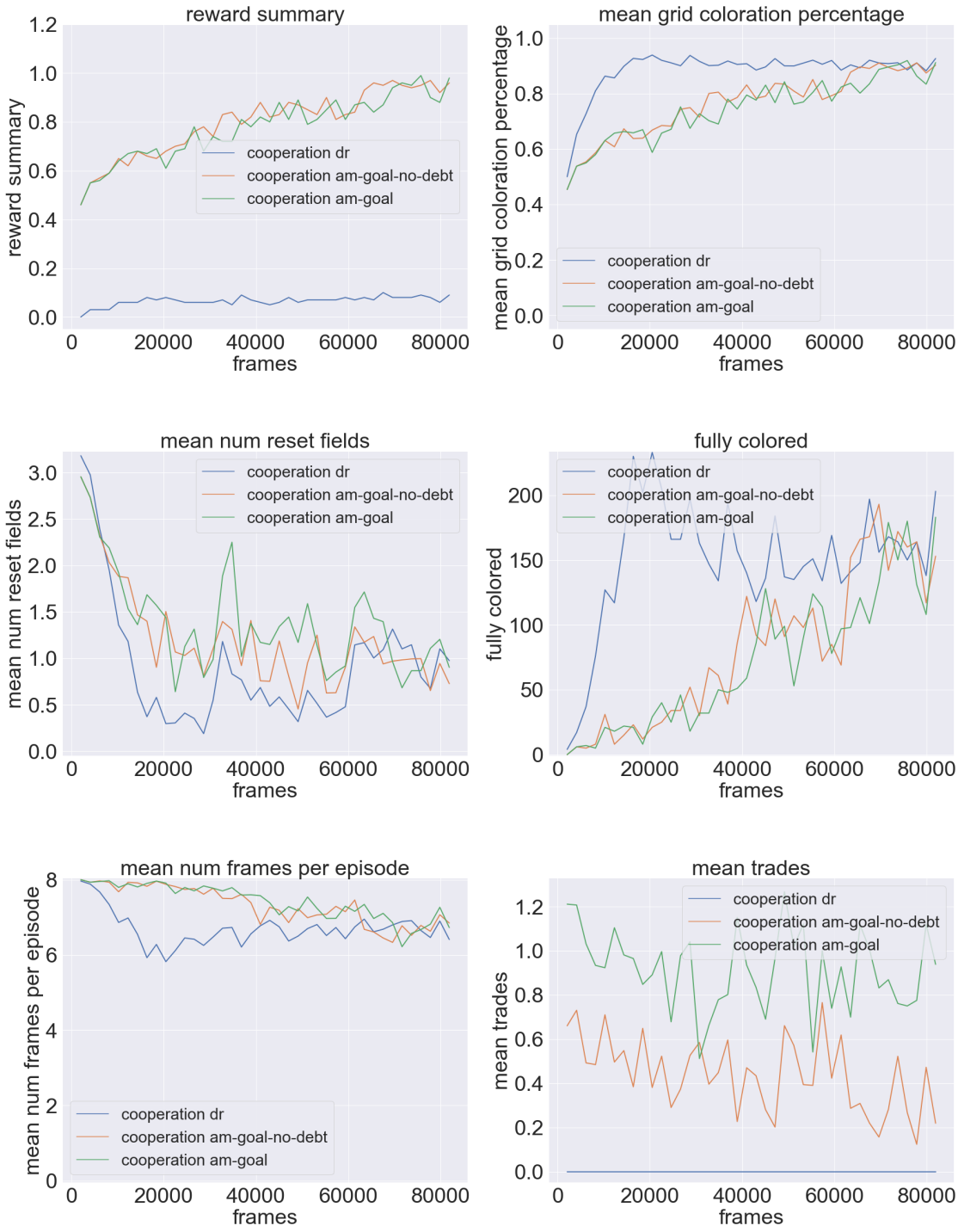


Figure B.3.: Top cooperation score details of two DQN agents

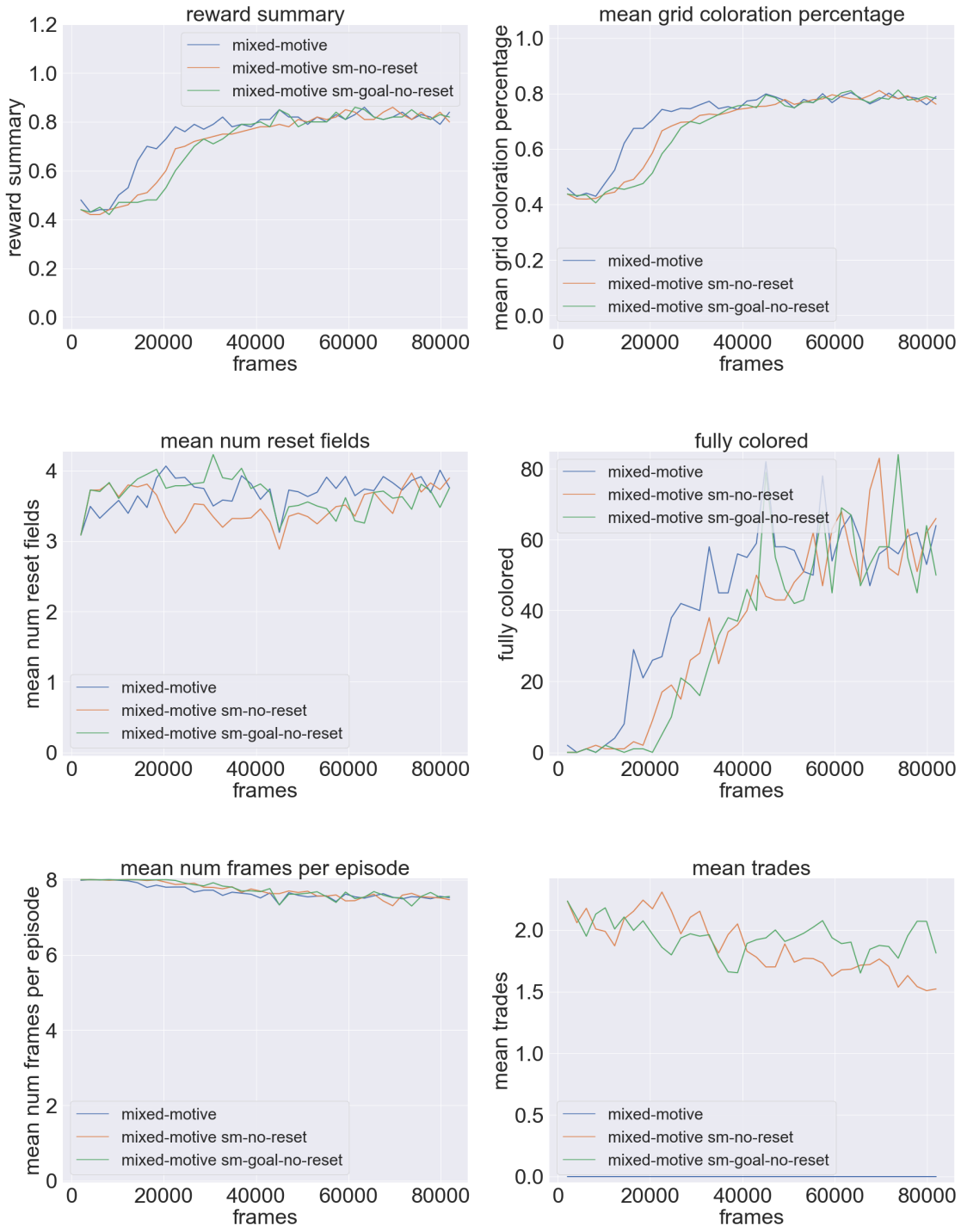


Figure B.4.: Top mixed-motive score details of two PPO agents

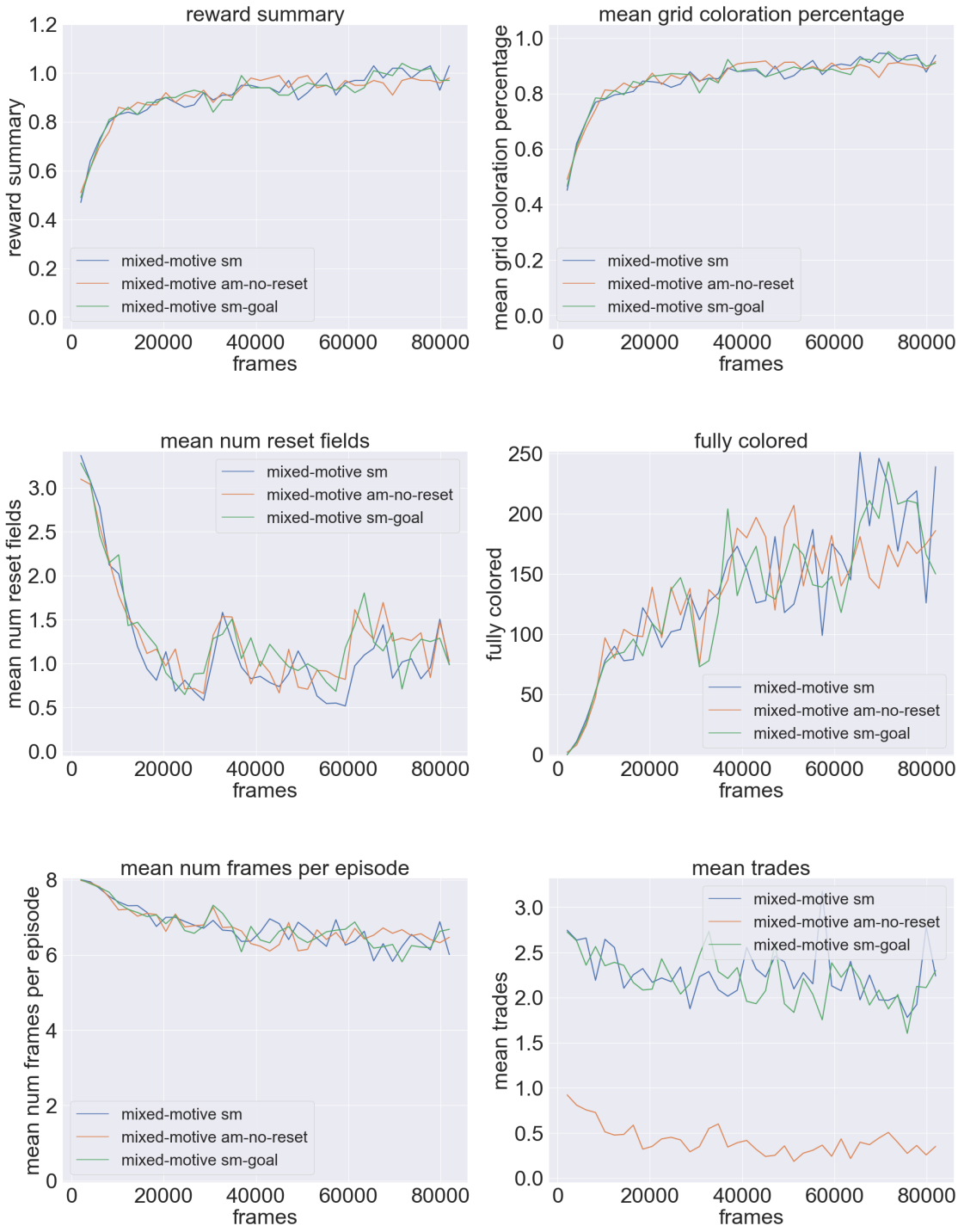


Figure B.5.: Top mixed-motive score details of two DQN agents

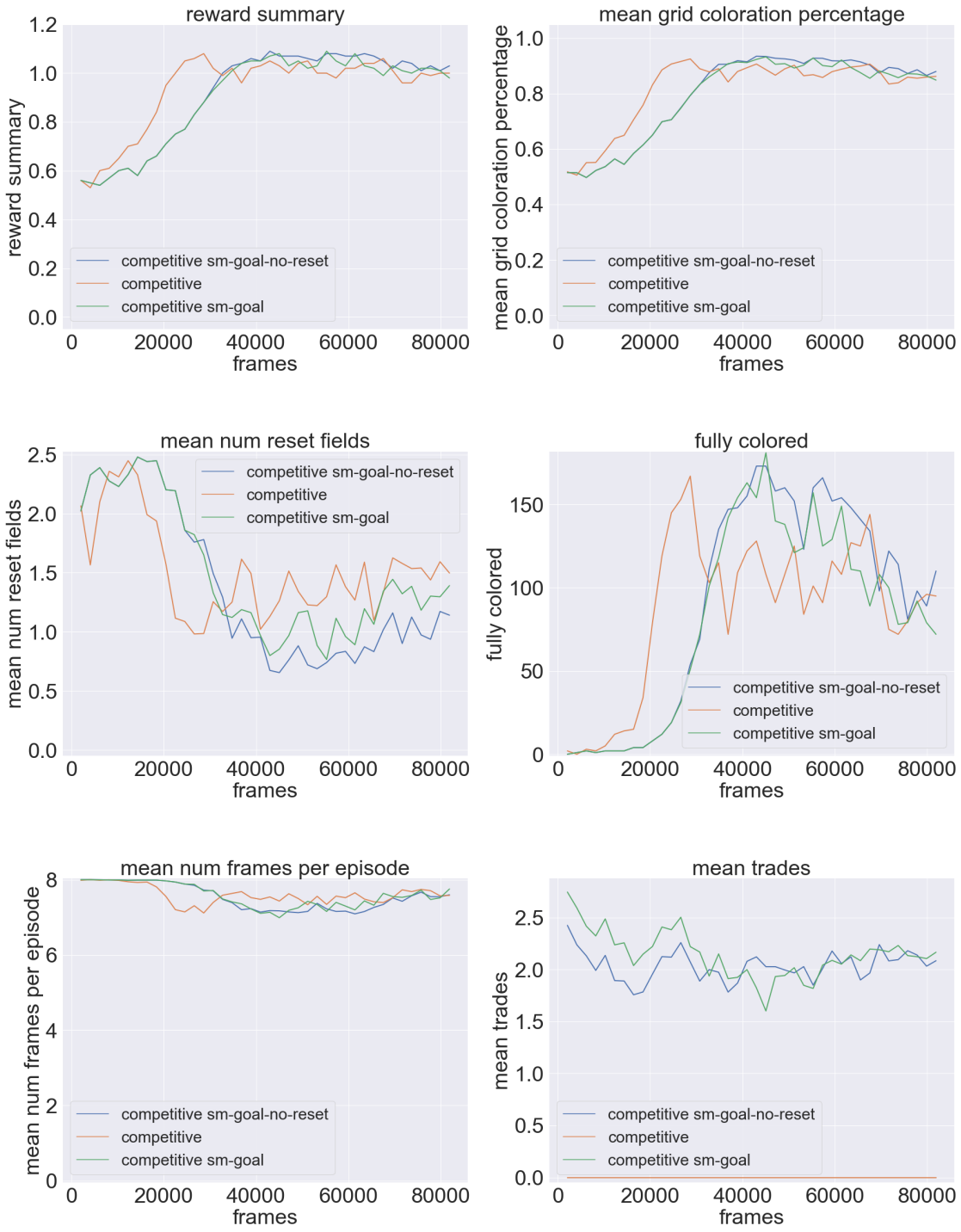


Figure B.6.: Top competitive score details of two PPO agents

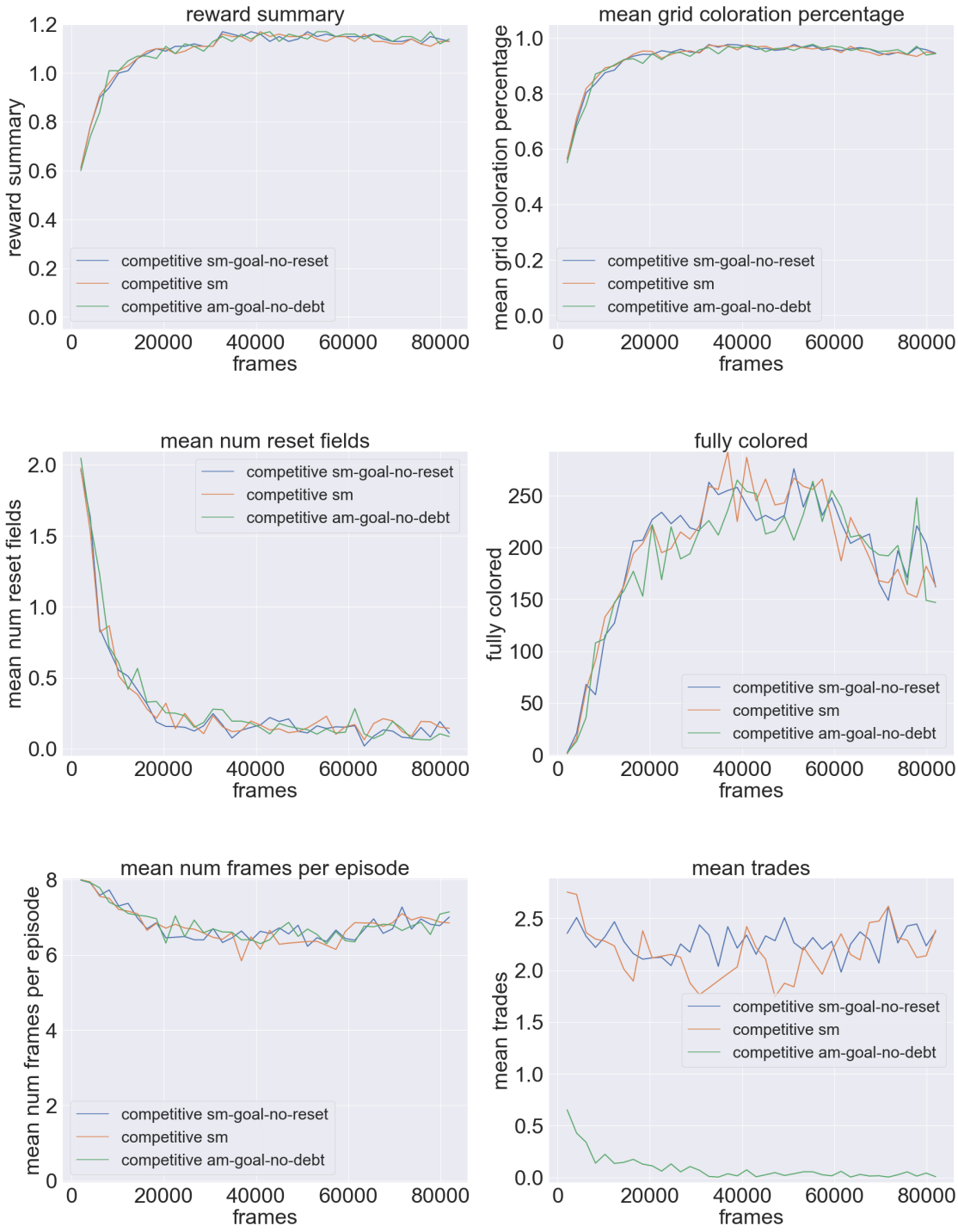


Figure B.7.: Top competitive score details of two DQN agents

B.0.2. Difficult Environment

An example to run a training process in a seven by seven environment is shown below.

```
$ python -m scripts.train
    --algo dqn
    --model dqn-difficult
    --agents 3
    --target-update 10000
    --replay-size 700000
    --epsilon-decay 20000
    --grid-size 7
    --max-steps 20
    --frames-per-proc 256
    --frames 200000
```

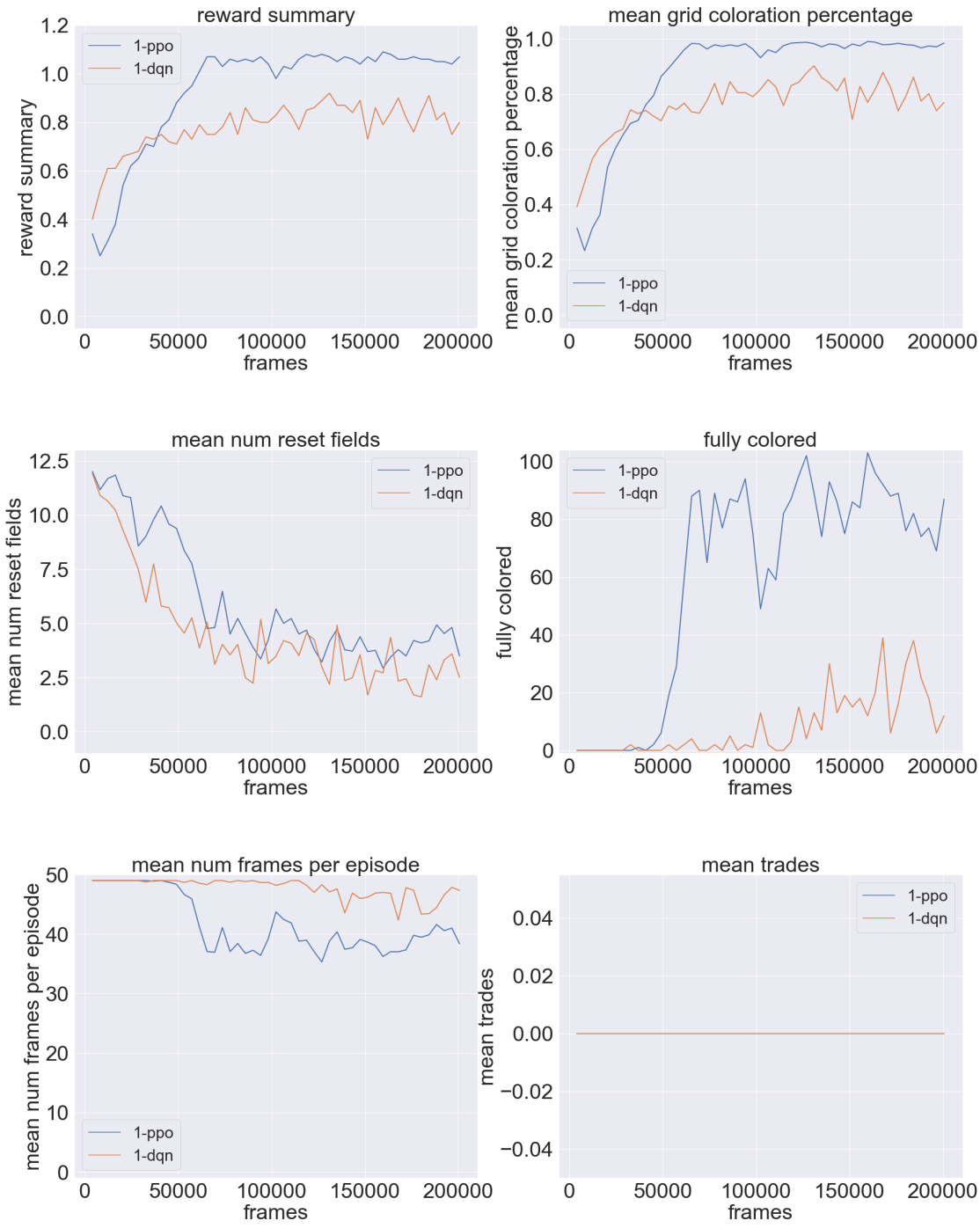


Figure B.8.: Details of the training in a 7x7 Environment with one agent using PPO and DQN

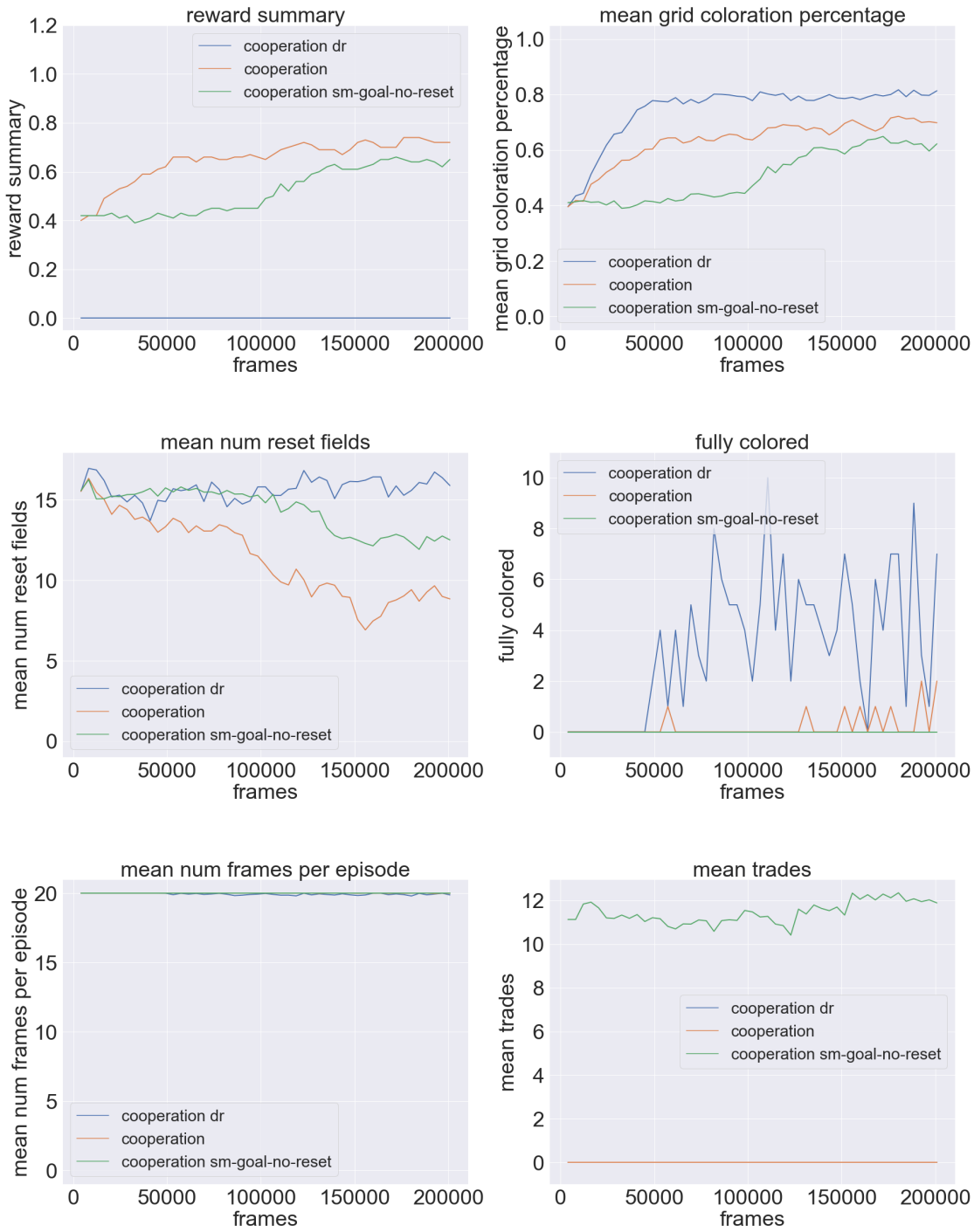


Figure B.9.: Top cooperation score details of three PPO agents in a 7x7 Environment

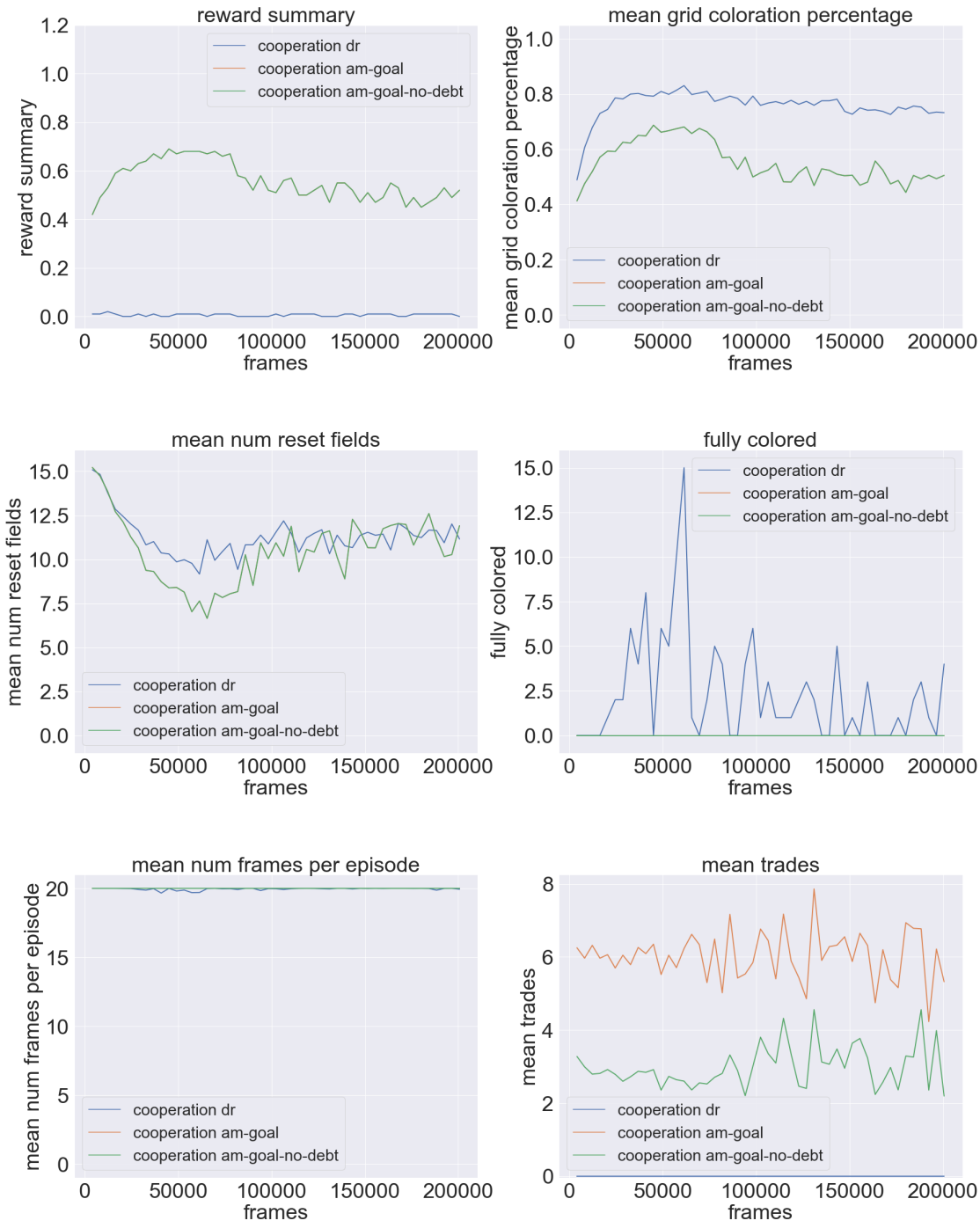


Figure B.10.: Top cooperation score details of three DQN agents in a 7x7 Environment

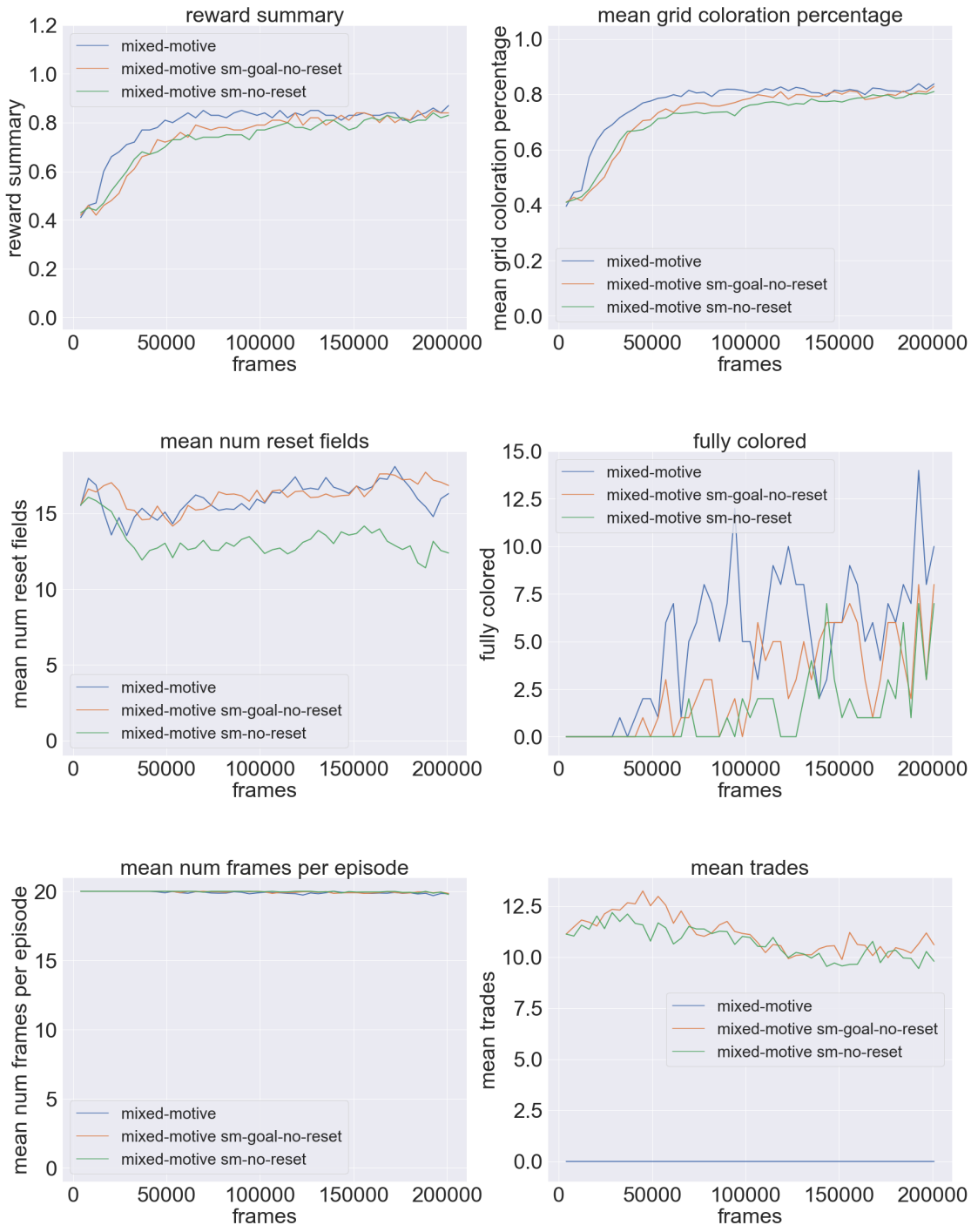


Figure B.11.: Top mixed-motive score details of three PPO agents in a 7x7 Environment

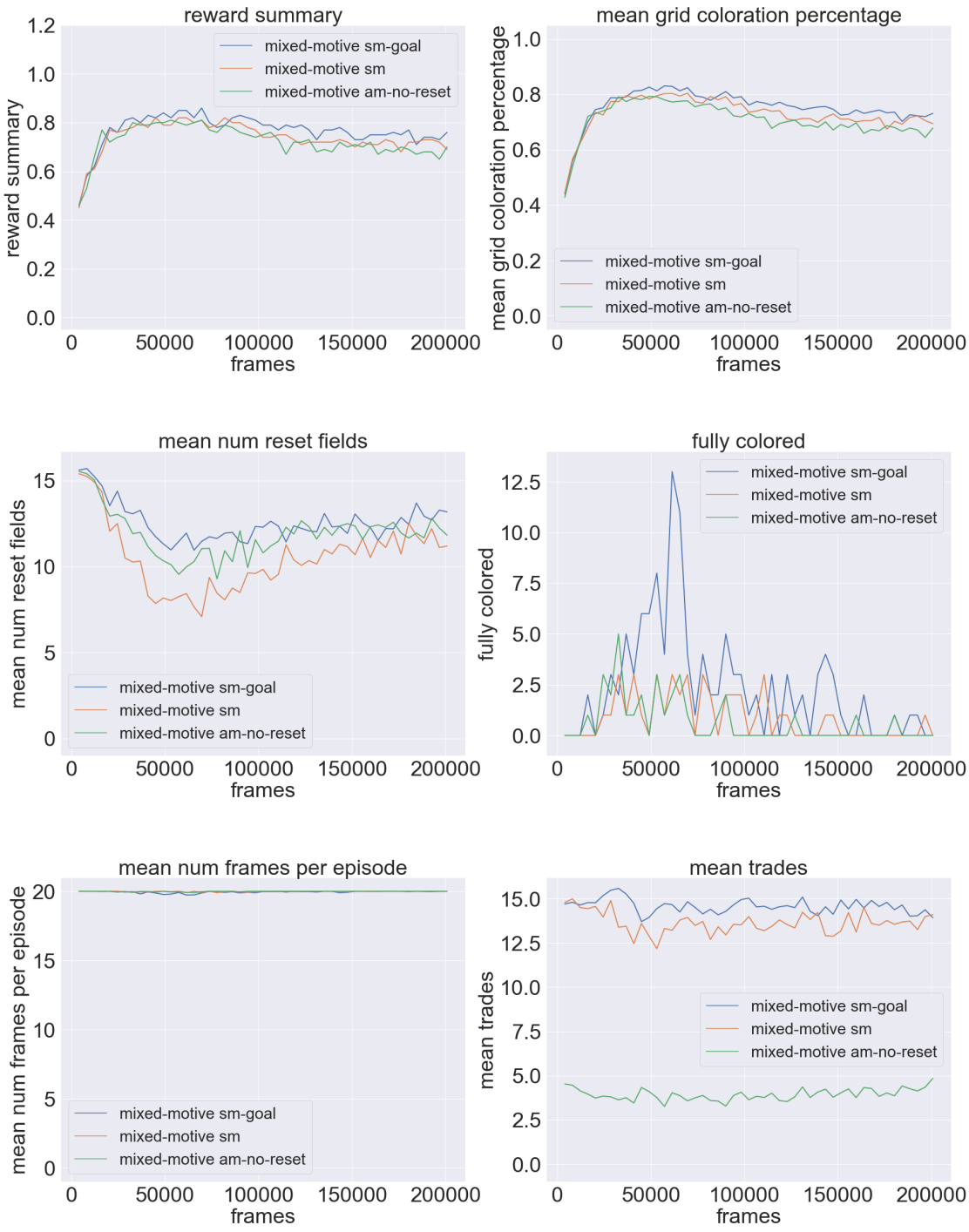


Figure B.12.: Top mixed-motive score details of three DQN agents in a 7x7 Environment

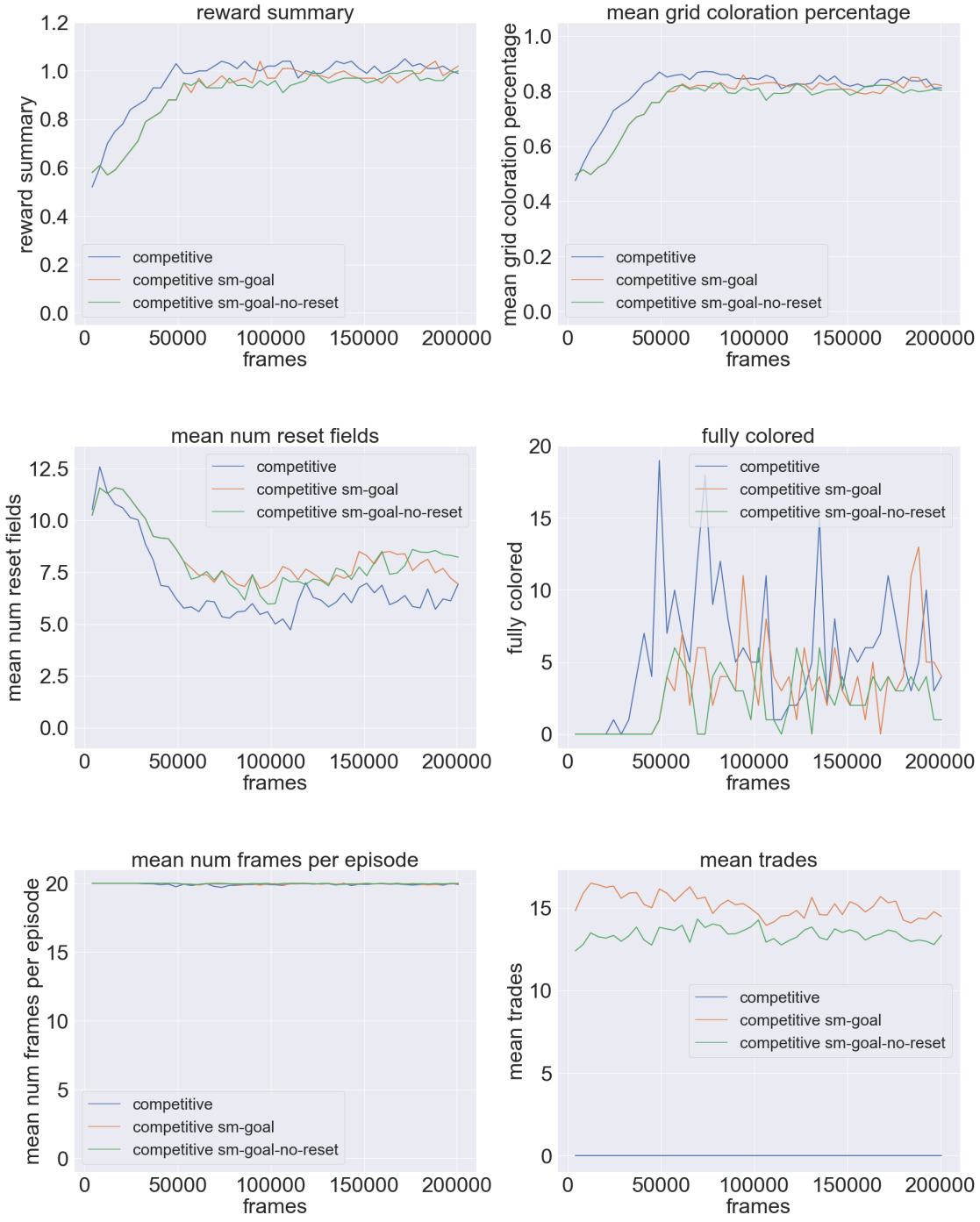


Figure B.13.: Top competitive score details of three PPO agents in a 7x7 Environment

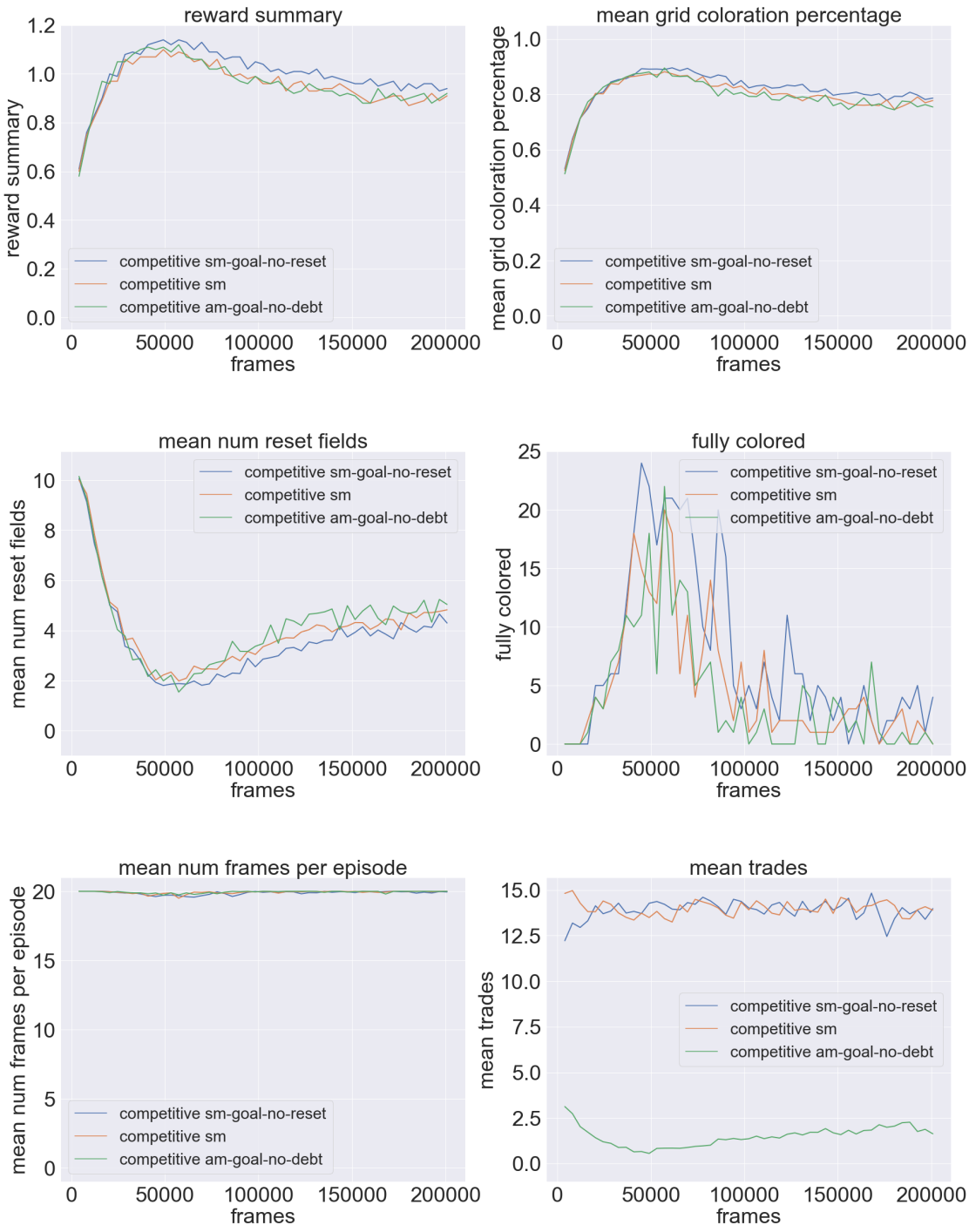


Figure B.14.: Top competitive score details of three DQN agents in a 7x7 Environment

B.0.3. Rooms Environment

An example to run a training process in a nine by nine room divided environment is shown below.

```
$ python -m scripts.train
    --algo ppo
    --agents 3
    --model ppo-rooms
    --env FourRooms-Grid-v0
    --grid-size 9
    --max-steps 30
    --frames-per-proc 256
    --frames 200000
```

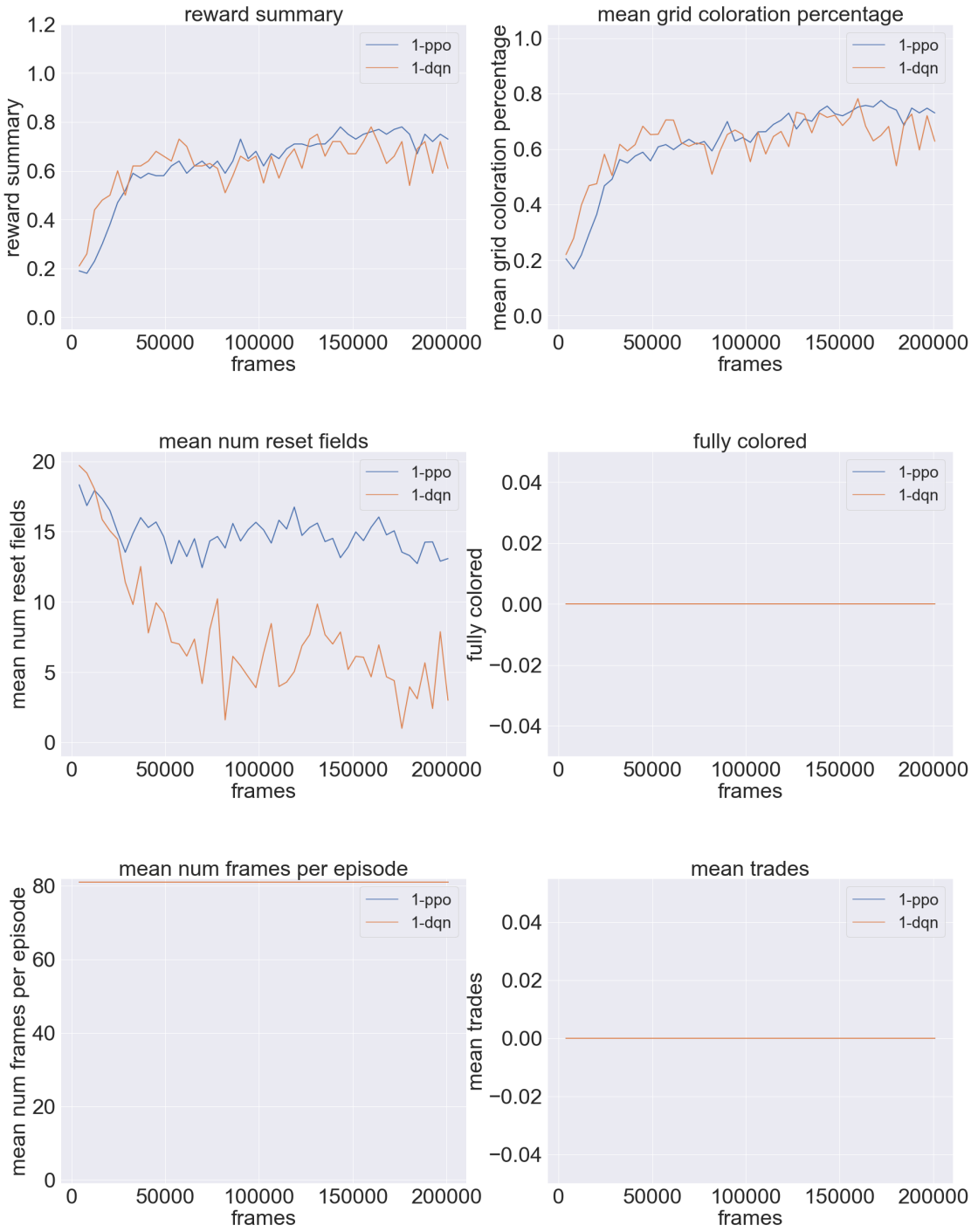


Figure B.15.: Details of the training in a 9x9 Rooms Environment with one agent using PPO and DQN

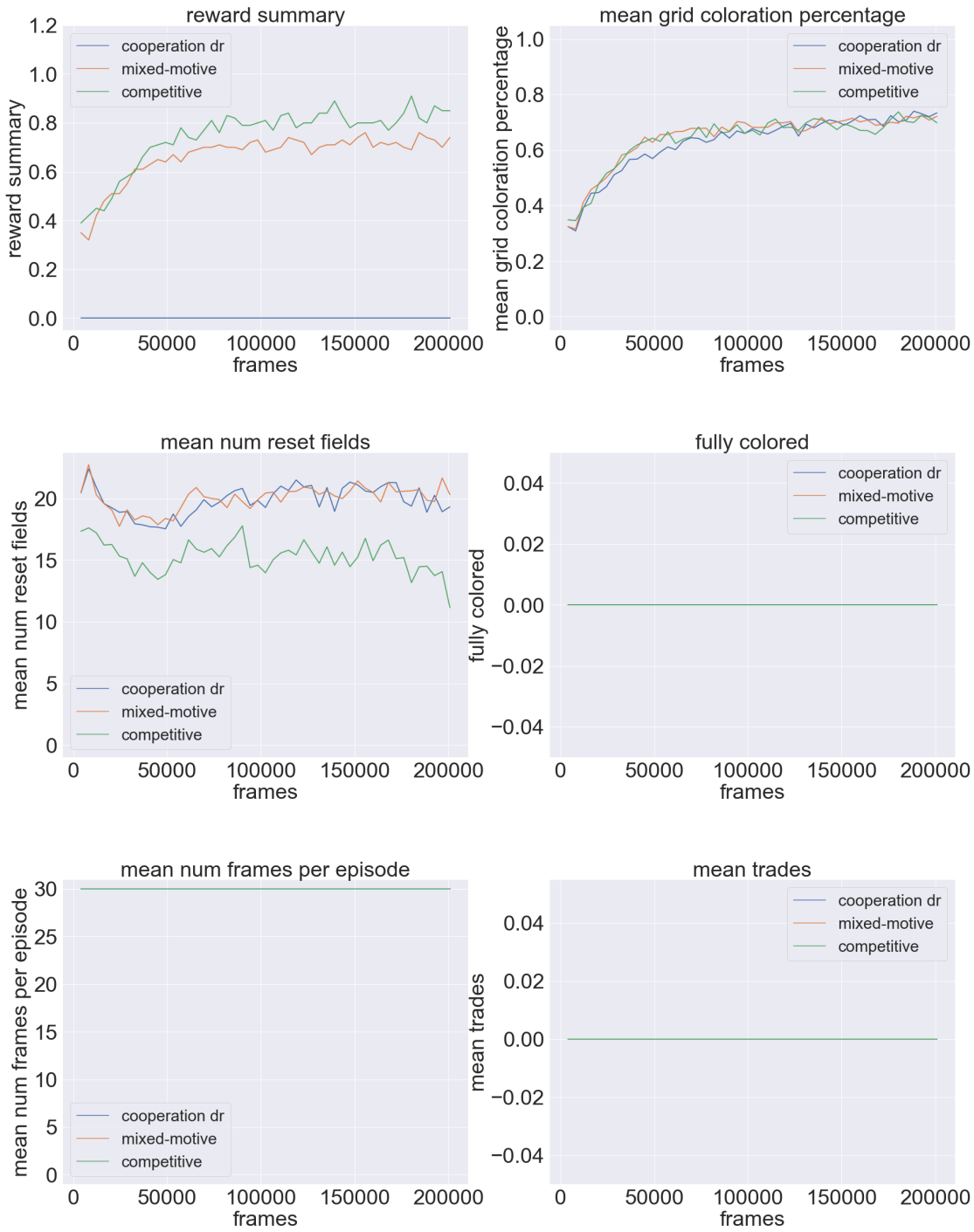


Figure B.16.: Top score details of three PPO agents in a 9x9 Rooms Environment

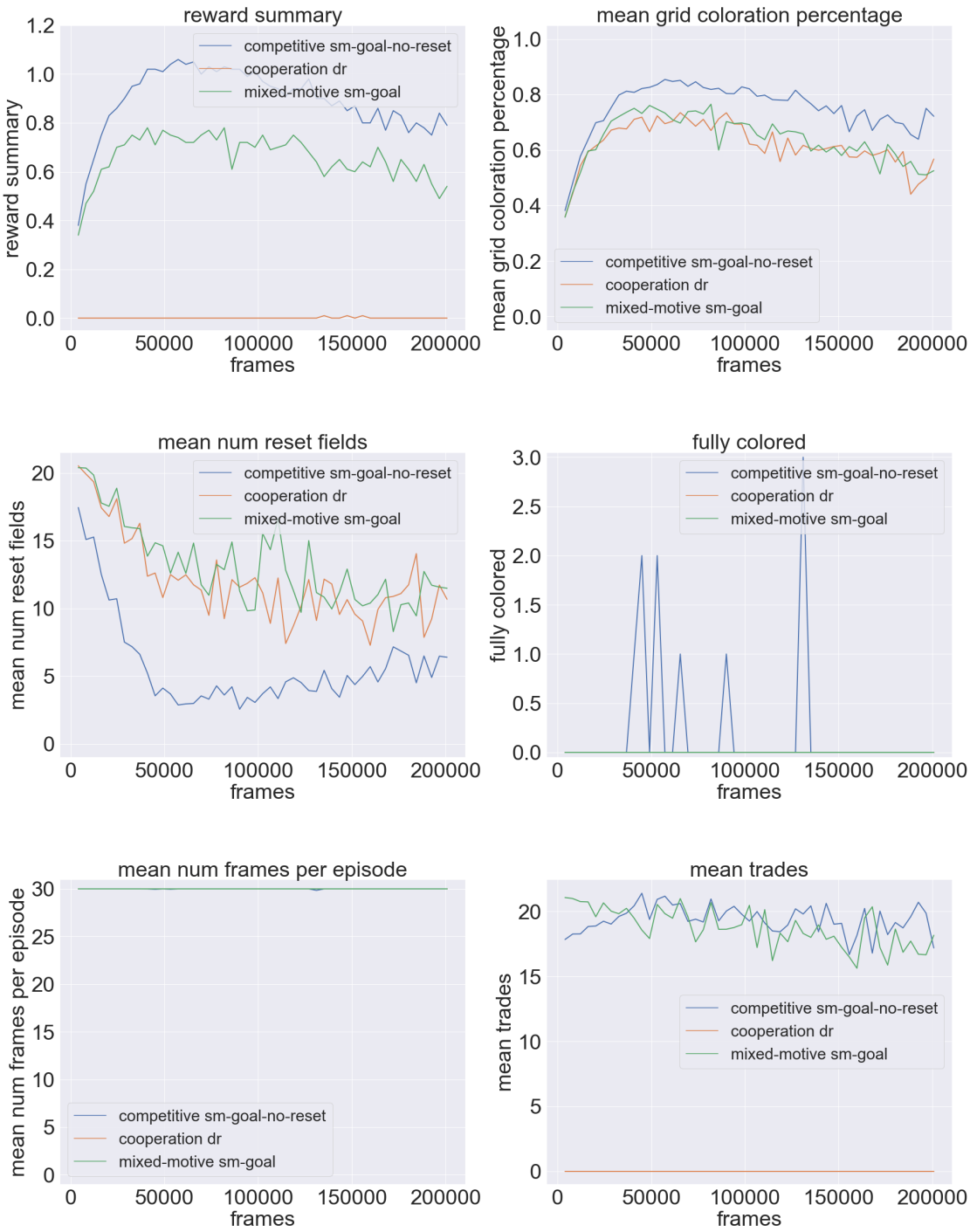


Figure B.17.: Top score details of three DQN agents in a 9x9 Rooms Environment

List of Figures

List of Tables

Code Listings

Bibliography

[AT04] Adrian K Agogino and Kagan Tumer. Unifying temporal and structural credit assignment problems. In *AAMAS*, volume 4, pages 980–987, 2004.

[BBDS10] Lucian Buşoniu, Robert Babuška, and Bart De Schutter. Multi-agent reinforcement learning: An overview. *Innovations in multi-agent systems and applications-1*, pages 183–221, 2010.

[CBWP18] Maxime Chevalier-Boisvert, Lucas Willems, and Suman Pal. Minimalistic gridworld environment for openai gym. `https://github.com/maximecb/gym-minigrid`, 2018. Accessed on September 2021.

[KT03] Vijay R Konda and John N Tsitsiklis. Onactor-critic algorithms. *SIAM journal on Control and Optimization*, 42(4):1143–1166, 2003.

[MBM+16] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[Min61] Marvin Minsky. Steps toward artificial intelligence. *Proceedings of the IRE*, 49(1):8–30, 1961.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[NKL18] Duc Thien Nguyen, Akshat Kumar, and Hoong Chuin Lau. Credit assignment for collective multiagent rl with global rewards. 2018.

[RB09] Zahra Rahaie and Hamid Beigy. Toward a solution to multi-agent credit assignment problem. In *2009 International Conference of Soft Computing and Pattern Recognition*, pages 563–568. IEEE, 2009.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[SBM+21] Kyrill Schmid, Lenz Belzner, Robert Müller, Johannes Tochtermann, and Claudia Linnhoff-Popien. Stochastic market games. In Zhi-Hua Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pages 384–390. International Joint Conferences on Artificial Intelligence Organization, 8 2021. Main Track.

[SLA+15] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[Sut84] Richard S Sutton. *Temporal credit assignment in reinforcement learning*. PhD thesis, University of Massachusetts Amherst, 1984.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[VG96] Cristina Versino and Luca Maria Gambardella. Learning real team solutions. In *Distributed Artificial Intelligence Meets Machine Learning Learning in Multi-Agent Environments*, pages 40–61. Springer, 1996.

[YT14] Logan Yliniemi and Kagan Tumer. Multi-objective multiagent credit assignment through difference rewards in reinforcement learning. In *Asia-Pacific Conference on Simulated Evolution and Learning*, pages 407–418. Springer, 2014.

[ZLS+20] Meng Zhou, Ziyu Liu, Pengwei Sui, Yixuan Li, and Yuk Ying Chung. Learning implicit credit assignment for cooperative multi-agent reinforcement learning. *arXiv preprint arXiv:2007.02529*, 2020.