

MASTER THESIS

Exploring the impact of markets on the credit assignment problem in a multiagent environment

Zarah Zahreddin

Entwurf vom November 6, 2021



MASTER THESIS

Exploring the impact of markets on the credit assignment problem in a multiagent environment

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

[illegible]

Contents

1. Introduction	1
2. Background	3
2.1. Reinforcement Learning	3
2.2. Proximal Policy Optimization	4
2.3. Deep Q-Learning	6
3. Related Work	9
3.1. Credit Assignment Problem	9
3.2. Markets	10
4. Approach	13
4.1. Coloring Environment	13
4.1.1. Compositions	14
4.1.2. Observation	14
4.2. Reward Calculations	15
4.3. Learning Process	18
4.3.1. DQN	20
4.3.2. PPO	20
4.4. Market Settings	20
4.4.1. Shareholder Market	21
4.4.2. Action Market	23
4.4.3. Reward Calculations	23
4.4.4. Additional Conditions	25
5. Results	27
6. Discussion	29
7. Conclusion	31
A. Training Parameters	33
List of Figures	37
List of Tables	39
Code Listings	41

Contents

Bibliography

43

1. Introduction

- Motivation
- Goal
- Research Question
- Structure

2. Background

- Describe the technical basis of your work
- Do not tell a historical story - make it short

2.1. Reinforcement Learning

Sutton and Barto wrote in “Reinforcement learning: An introduction” [SB18] that Reinforcement learning (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 finding the goal state is described as the Markov Decision Process (MDP). When multiple agents act in the same environment the Markov decision process is called a stochastic game [BBDS10].

rl components

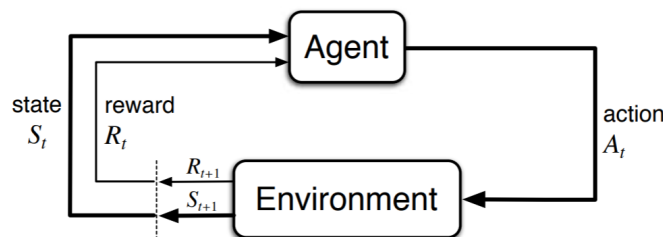


Figure 2.1.: The cycle of agent-environment interaction as shown in “Reinforcement learning: An introduction” [SB18]

One environment state S_t is part of a set S containing all possible states. During each point in time t the agent can interact with the environment by executing an action A_t , which in turn changes the environment state. Since it is possible that not all actions are valid in each state the agents action selection is based on a restricted set $A_t \in A(S_t)$. In a multiagent environment, every agent chooses its action and adds it into a joint action set, which is executed collectively during t [BBDS10].

sets and values

The reward R_t is element of a set of possible rewards R , which is a subset of real numbers $R \subset \mathbb{R}$. 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. The general concept of RL, as defined by Sutton and Barto, is for agents to maximize rewards. Unlike machine learning approaches, the agent starts with no knowledge about good or bad actions and enhances the decision-making over time.

policy

2. Background

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. In most cases however, policies 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 then can be used to update the policy accordingly. For a negative or low reward as an example, the probability of policy $\pi(a | s)$ decreases, reducing the chances of executing that same action in that specific state again.

While rewards only rate the immediate situation, a value function, i.e. the state-value function $v_\pi(s)$ for a policy π , can be used to estimate the long-term value of a state s . The result is the total accumulated reward an agent could get following that state and choosing actions based on the current policy. 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. On one hand, an agent has to explore the options given 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 Proximal Policy Optimization (PPO) in the article "Proximal Policy Optimization Algorithms" [SWD⁺17]. This section is solely based on that article in order to explain the Algorithm. 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 that data through several epochs.

The origin of PPO lies in a similar approach called Trust Region Policy Optimization (TRPO). TRPO strives to maximize the following function:

$$\underset{\theta}{\text{maximize}} \mathbb{E}_t \left[\frac{\pi_\theta(a_t | s_t)}{\pi_{\theta_{old}}(a_t | s_t)} \hat{A}_t - \beta KL[\pi_{\theta_{old}}(\cdot | s_t), \pi_\theta(\cdot | s_t)] \right] \quad (2.1)$$

with \hat{A}_t as an estimator of the advantage function. The advantage function is often calculated with the state-value function $V(s)$, a cumulative return $R_t = \sum_{k=0}^{\infty} \lambda^k r_{t+k}$ [MBM⁺16] and a discount coefficient $\lambda \in (0, 1]$ [MBM⁺16] over a period of Time t . The factor λ in the context of RL is mostly used to increase the importance of the near future and reduce the relevance of unsure future rewards or values. A positive result of the advantage function shows that the executed action is profitable.

2.2. Proximal Policy Optimization

The fraction $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ in the Minuend of (2.1) can be replaced by $r(\theta)$ and represents the probability ratio of an action in the current policy in comparison to the old policy. 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. further describe that TRPO maximizes the "surrogate" objective

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t | s_t)}{\pi_{\theta_{old}}(a_t | s_t)} \hat{A}_t \right] = \hat{\mathbb{E}}_t [r(\theta) \hat{A}_t] \quad (2.2)$$

However, maximized on its own without a penalty, this results in a large outcome and leads to drastic policy updates.

problem
TRPO

In order to stay in a trust region, as the name suggests, a penalty is subtracted from the surrogate function (2.2). The penalty is the Subtrahend of equation (2.1) 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. Even during a TRPO run it could be necessary to adapt the coefficient, due to changes.

PPO

Therefore Schulman et al. introduced

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r(\theta) \hat{A}_t, \text{clip}(r(\theta), 1 - \epsilon, 1 + \epsilon) \hat{A}_t \right) \right] \quad (2.3)$$

which is very similar to equation (2.1) but does not require coefficients. The first min entry contains L^{CPI} (2.2). 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.

PPO Algo

Finally, PPO is introduced with the following equation

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right] \quad (2.4)$$

with c_1 and c_2 as coefficients. The authors point out that the loss function $L_t^{VF} = (V_\theta(s_t) - V_t^{tag})^2$ 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.

AC

Schulman et al. also showed an example of the Algorithm using PPO with an actor-critic approach, see Fig. 2.2. According to Konda and Tsitsiklis [KT03], 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.

PPO pseudo

Here, N denotes 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 $L_t^{CLIP+VF+S}(\theta)$ (2.4) is optimized during K epochs. For the optimization process a small random batch of the previous time steps is used.

2. Background

Algorithm 1 PPO, Actor-Critic Style

```

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

```

Figure 2.2.: Exemplary use of PPO, as shown in “Proximal Policy Optimization Algorithms” [SWD⁺17]

2.3. Deep Q-Learning

q value function

Another algorithm often compared with PPO is the training of a deep Q-Network (DQN) with Q-learning. This Algorithm originated from a simple equation called Q-value [JJR19], expanding the common value function $v_\pi(s)$ of chapter 2.1 with an action. The overall goal here is to maximize the possible future rewards.

The expected reward is calculated starting at a state s , executing a specific action a and following the next states by using a policy π . Mnih et al. [MKS⁺13] define the optimal action-value with the following:

$$Q^*(s, a) = \max_{\pi} \mathbb{E} \left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots \mid s_t = s, a_t = a, \pi \right] \quad (2.5)$$

Bellmann

Here, the maximum sum of rewards is calculated. By using a γ coefficient, future rewards are discounted, putting greater importance on the next few values. They proceed by explaining that if all $Q^*(s', a')$ values of the next time step are known then the following Bellman equation can be calculated:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \epsilon} \left[r + \gamma \max_{a'} Q^*(s', a') \mid s, a \right] \quad (2.6)$$

However, in reality the optimal $Q^*(s', a')$ values are not known, thus Mnih et al. continue with the basic idea of RL - iteratively updating this Q function:

$$Q_{i+1}(s, a) = \mathbb{E} \left[r + \gamma \max_{a'} Q_i(s', a') \mid s, a \right] \quad (2.7)$$

Q network

They conclude that an increasing $i \rightarrow \infty$ will allow the functions to converge, resulting in $Q_i \rightarrow Q^*$.

Nonetheless, Mnih et al. argue, that this iterative approach is impractical and suggest using a neural network to approximate the action-value function instead. Hence, the new Q function is $Q(s, a, \theta) \approx Q^*(s, a)$, with θ representing all network weights [JJR19]. They refer to this network as a Q-network, which can be trained using the following loss

function:

$$L_i(\theta_i) = \mathbb{E}_{s,a \sim p(\cdot)} \left[(y_i - Q(s, a; \theta_i))^2 \right] \quad (2.8)$$

$$y_i = \mathbb{E}_{s' \sim \varepsilon} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a \right] \quad (2.9)$$

Jafari et al. [JJR19] explain that the loss function shows how much the estimated values deviate from the target values y_i (2.9). Mnih et al. emphasize that y_i holds the network parameters of the previous iteration, due to θ_{i-1} . Finally, the differentiation of the loss function with respect to θ results in the gradient function below.

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a \sim p(\cdot); s' \sim \varepsilon} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \right) \nabla_{\theta_i} Q(s, a; \theta_i) \right] \quad (2.10)$$

Nevertheless, in most cases it is sufficient to optimize the loss function through stochastic gradient descent.

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 across multiple episodes.

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. Otherwise, the best action according to the Q-value is selected.

experience re-
play

action selec-
tion

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory \mathcal{D} to capacity N

Initialize action-value function Q with random weights

for episode = 1, M **do**

 Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

for $t = 1, T$ **do**

 With probability ϵ select a random action a_t

 otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

 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 $(\phi_t, a_t, r_t, \phi_{t+1})$ in \mathcal{D}

 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from \mathcal{D}

 Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3

end for

end for

Figure 2.3.: DQN with Experience Replay, as shown in “Playing atari with deep reinforcement learning” [MKS⁺13]

Executing the selected action results in a memory entry in form of the earlier described

action selec-
tion

2. Background

minibatch advantages

quadruple. Afterwards a minibatch of the replay memory is randomly sampled and used to perform a gradient descent step. The reference to “equation 3” in the figure 2.3 points to formula 2.10 here.

The suggested process offers several advantages. The replay memory for instance, leads to a smaller deviation or fluctuation in the parameters. The random samples of minibatches can prove 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.

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 independence problem 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 actions in a single-time-step. Since the actions are executed all at once, the problem 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. In order

intro and
comp. prob-
lems

coop problems

coop and prob-
lem

CAP defi-
nition and
kinds

CAP multi

cap solution dr

3. Related Work

to find the DR $D_i(z)$ for an agent i , the following function applies:

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

The first part of the equation $G(z)$ represents the overall reward. This reward is the result of the collective action set of one time step, containing the actions of all agents. 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 difference between the two rewards results in the difference reward $D_i(z)$ of agent i . A high DR indicates a lucrative action of the analyzing agent, since excluding its action or picking something else leads to a small subtrahend. With this method each agent has the opportunity to learn how their actions contribute to the global reward, enabling individual learning.

3.2. Markets

intro mixed
motive

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.

SMG details

Schmid et al. introduced in “Stochastic Market Games” [SBM⁺21] concepts adding 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. 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.

sm

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.

sm transaction

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 transaction agent by a fixed dividend d . Schmid et al. mention that an optional price p can be defined as a price a seller receives from the buyer upon each share purchase. They claim however, 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. am

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.

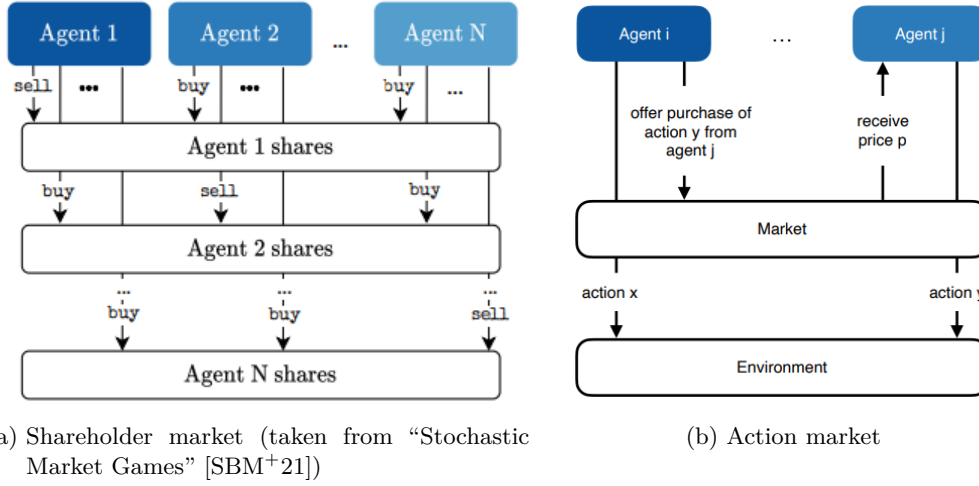


Figure 3.1.: Illustrated Markets as defined in “Stochastic Market Games” [SBM⁺21]

In both market settings a purchase is established if the specified agent happens to execute a selling action that matches with a buyer. It is important to emphasize that the matching is performed each time step, leaving it to chance, whether purchases take place. Hence, in case of an action market 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.

transactions
by chance

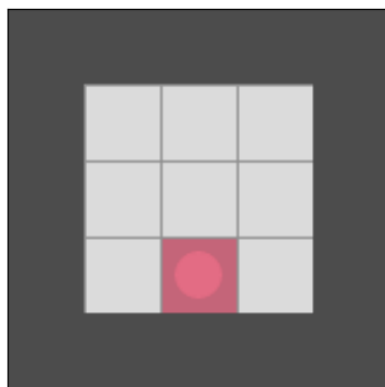
4. Approach

- What is your plan?
- How do you proof that it worked? -> Metric and Experiments

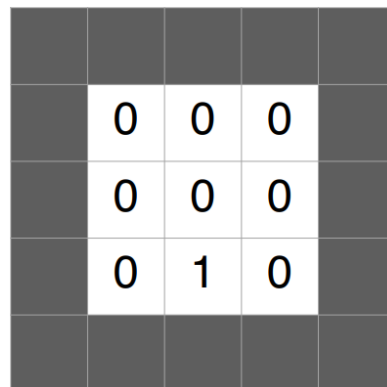
4.1. Coloring Environment

A RL environment is a versatile and unbiased instance, that can be used to visualize agent behavior and environmental changes. 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. For the purpose of this research however, the environment is changed heavily, becoming the “Coloring Environment”. Multiple agents can act in the new instance to try and achieve a new goal - to color all walkable cells.

origin and intro



(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 1 represents a colored cell.

Figure 4.1.: Representations of the coloring environment

Figure 4.1b shows a simplified environment observation an agent processes each time step. Every environment cell holds information about the object 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.

cell objects

4. Approach

4.1.1. Compositions

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.

floor col-
oration

Floor cells manage the coloration state in binary form, as displayed in 4.1b, with a 1 signaling that the cell is colored. The environment reacts to agent movements by coloring the cells they visit. Agents successfully solve the environment once all fields are colored. Otherwise agents lose by using up a limited amount of steps.

bit switching

If a cell is already in coloration state 1 and an agent walks over it again the bit is switched, and the cell is reset to 0, removing the color. Besides moving up, down, left and right an agent can also execute the action wait, to stay in place.

cooperative
multiagents

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. All agents always receive the same reward, regardless of the coloration. In an extreme example one agent could single-handedly color the whole grid, and all others would still profit from the same high reward.

mixed-motive
multiagents

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 the red agent 1 colored 60% of the grid red the reward for that agent would be 0.60. All other agents receive their individual color percentages.

competitive
multiagents

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 captured cell contains the same color as the agent the cell is reset instead. Hence, taking over the cells of the opponents is beneficial, since it increases the color presence of the own color, leading to a higher reward.

comparison
multiagents

In comparison, the basic mixed-motive composition shows no advantages resetting colored cells. In this implementation resetting cells is punished with a small negative reward. Here, the cell is not captured, and the agent won't receive more reward. Rather the opposite applies, and the agent would be punished for doing that. Hence, it is not likely that the agents work against each other yielding in an agent composition with neutral behavior.

what the agent
sees

4.1.2. Observation

The observations agents receive from the environment are always generated from their individual point of view and only contain a restricted area around them. In large environments this feature increases the difficulty but at the same time reflects the reality. An example for a proper observation of an agent is shown in 4.2. This observation represents the internal state of the human representation of image 4.1a.

description


```
[
  [ [2 1 0] [2 1 0] [2 1 0] [2 1 0] [2 1 0] [2 1 0] [2 1 0] ]
  [ [2 1 0] [2 1 0] [2 1 0] [2 1 0] [2 1 0] [2 1 0] [2 1 0] ]
  [ [2 1 0] [3 0 1] [3 0 1] [3 0 1] [2 1 0] [2 1 0] [2 1 0] ]
  [ [2 1 0] [3 0 1] [3 0 1] [4 1 2] [2 1 0] [2 1 0] [2 1 0] ]
  [ [2 1 0] [3 0 1] [3 0 1] [3 0 1] [2 1 0] [2 1 0] [2 1 0] ]
  [ [2 1 0] [2 1 0] [2 1 0] [2 1 0] [2 1 0] [2 1 0] [2 1 0] ]
  [ [2 1 0] [2 1 0] [2 1 0] [2 1 0] [2 1 0] [2 1 0] [2 1 0] ]
]
```

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 bold entries are part of the grid that is shown in Figure 4.1a and the colored array is the agent position. All remaining grayed out elements are additions to the observation in order to standardize the observation size.

dimensions

The first dimension of the array contains the whole observation. The second array dimension represents each grid column of 4.1a, starting from the top left and the inner dimension encodes each cell. The encoding first defines the object type with 1 being just an empty cell, 2 shows a wall, 3 a floor tile and 4 an agent. Second, the coloration status is visible. Since agents can not walk onto walls, that object type has a coloration state of 1 per default.

cell encoding
colors

The third encoding signalizes the color of a cell. To better distinguish the types in the human representation, walls are 0 for the color black and floors are initially white with encoding 1. Each agent is assigned a number from 2 upwards which in turn stands for a randomly generated color. The Floor color encoding is overwritten with the agents color when the cell is captured.

4.2. Reward Calculations

activation line

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.

command algo
and model

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, includ-

4. Approach

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

```
1 $ python -m Coloring.scripts.train
2     --algo ppo
3     --model ppo-training
4     --env Empty-Grid-v0
5     --grid-size 7
6     --agents 3
7     --max-steps 350
8     --setting mixed-motive
```

ing those not listed in 4.1, have default values and are discussed in the sections 4.3 and 4.4. An overview of all training parameters and their default values is listed in Appendix A.

command env

With `--model` the destination path is defined, in which all logs, recordings and status updates are stored. Line 4 and 5 configures the environment. Alternatively 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 7 the agents can only move in a 5 by 5 field due to the surrounding walls.

settings

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”.

environment
reward

Regardless of the composition, agents initially get separate rewards in each step based on their environment change contribution. For instance, agents that color a field receive a positive reward of 0.1, whereas agents that reset a field get a penalty of negative 0.1. Agents that just wait get a reward of 0. 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.

All rewards are written into a list and returned by the environment, see algorithm 1, line 1. The position in the list indicates the receiving agent. In algorithm 1 the process of adapting the initial environment rewards each step is summarized. Here, the specified training arguments are taken into account, leading to the four conditions below.

Algorithm 1: Reward calculation each step

```

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

```

equal rewards

First, the reward is changed in a cooperative setting. This setting requires a new homogeneous reward, that is calculated out of the initial environment values. Settings that contain “mixed-motive” skip this step, since in this case each agent keeps their individual value. The calculation for cooperation is summing up all array entries and checking if they exceed an upper or lower bound.

bounds

If that is the case then the reward is set to the corresponding limit. Otherwise, the sum is taken as is. Then, all rewards of the array are changed to the calculated value. 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.

market actions

Second, the market transactions change the rewards, if a SM or AM is specified. The market details are discussed in Section 4.4. One thing to note here is that agents can execute transactions in each step, spending their current reward on actions or shares that are for sale or receive the purchase price from buyers. Therefore, the rewards can change in this step too.

difference
reward intro

The third and fourth condition depend on the done flag which signalizes the end of an episode. The environment sets done to true, once either the goal is reached or the maximum step amount is executed. However, if the episode is not done yet and the setting specifies DR calculation then the corresponding function is called. The last condition needs to take the done value into account, since for the final rewards the DR could also be applied and this condition prevents the function to be called twice.

—————TODO!!!!

For the DR calculation the environment saves some additional information in the **info** variable. Namely, what the initial environment reward for each agent would be, if that agent had executed the waiting action and whether the goal would have been reached if the agent had waited. The DR function itself reassigns the reward of each agent with the difference between the cooperation reward subtracted with the waiting action reward of **info**. —————TODO!!!!

difference re-
ward function

4. Approach

done calculations

Lastly, the final reward is calculated when done is set to true. Algorithm 2 shows the executed calculations of that case. Again the first thing to check is the setting. If the agents do not work in cooperation, each agents' grid coloration percentage, based on their color presence, is added to the individual reward. Otherwise, the general grid coloration, regardless of the colors is looked up and added to the agent rewards. Finally, the last market calculations are included into the rewards, see chapter 4.4 for details.

Algorithm 2: Final reward calculation

```

1 if mixed 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 if market specified then
13   | rewards = final market adjustments executed on rewards
14 end
15 return rewards

```

4.3. Learning Process

Independent learning

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, observations still connect the agent experiences, by including the positions of all agents on the grid and reacting to their joint actions in each step.

process structure

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.

training setup

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 loop until that

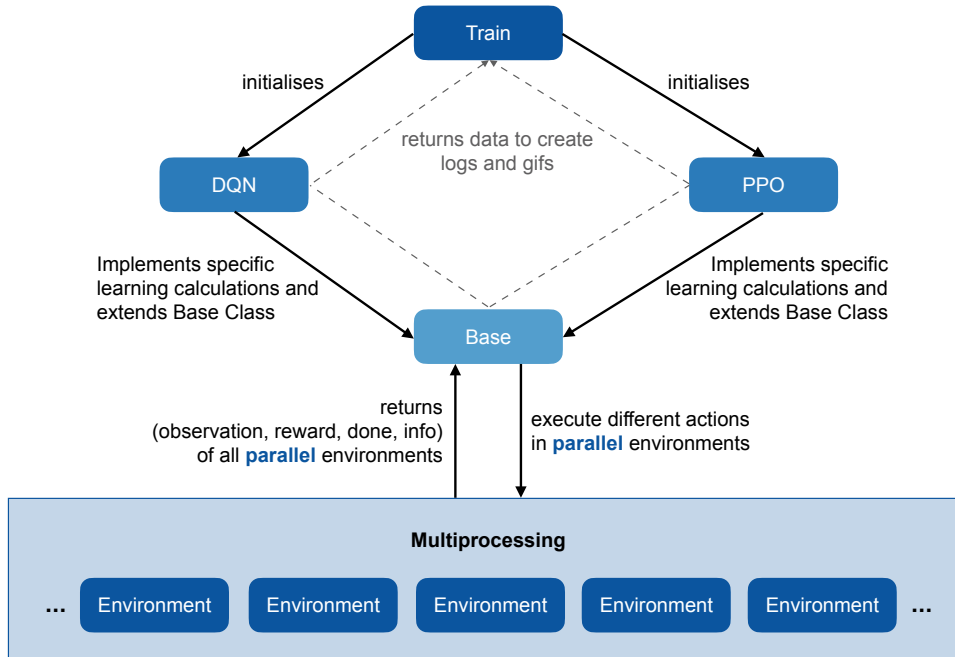


Figure 4.3.: The training structure

number is reached. In this case frames are equivalent to agent steps. During the loop, the defined training algorithm `--algo` is executed.

base class

Since both algorithms have similar procedures, they share the base class. In it experiences are gathered through a certain amount of actions which are executed on parallel environments. The action amount is set with `--frames-per-procs`. During that period details like gained rewards, observations, actions and more are stored in base class variables that are accessible by the learning algorithms. When all experience actions are executed, the base variables are reset and log values are returned, as shown in Figure 4.3. Afterwards the frame counter in the training script is updated with the amount of `--frames-per-procs`. The training loop ends when the frames counter is greater or equal `--frames`. Otherwise more experience batches are gathered.

action selection

Both learning algorithms include their own action selection methods. The PPO implementation relies on an actor-critic neural network, letting the actor part calculate a probability distribution of the action space. In case of the DQN implementation a linear neural 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.

4. Approach

4.3.1. DQN

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 enhanced 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.

4.3.2. PPO

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. 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 experience 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 pseudo code of 2.2.

4.4. Market Settings

TODO: SHAREHOLDER CALCULATIONS CHANGED TO INCLUDE SHARE IN EVERY STEP!

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 have the option to conduct market transactions.

Per default, the environment action space is discrete and only contains five elements: 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 signaling that the agent wants to sell its share. An abstract representation of a shareholder action array is: `[environment_action, agent_index, sell_share]`.

sm - action
space

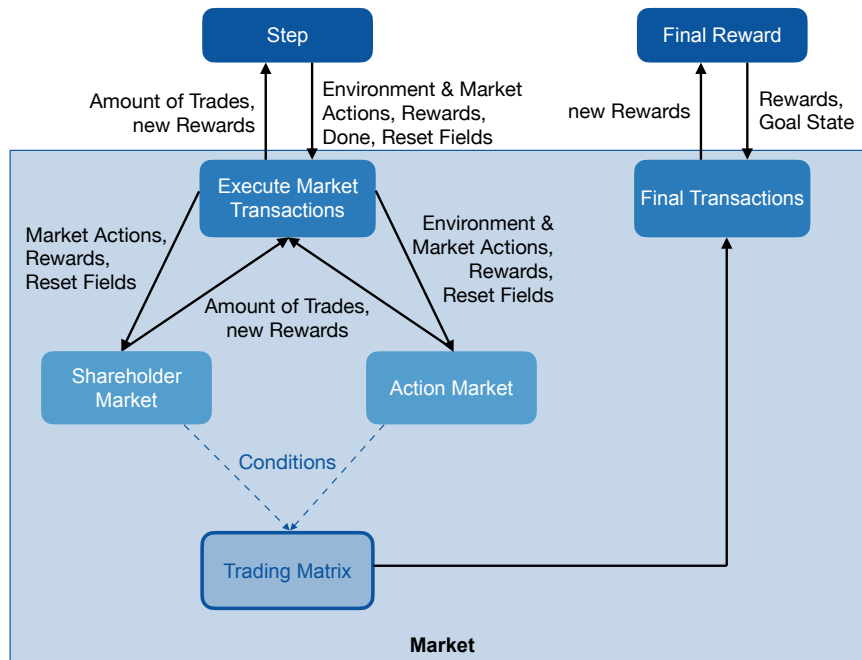


Figure 4.4.: The market elements

In Figure 4.4 market elements are visualized. On the left, the process that happens in each step is shown, as mentioned in Algorithm 1. 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 the buying and selling information in this case.

shareholder
market step

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

trading matrix

4. Approach

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

$$trading_matrix = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

execute mar-
ket transac-
tions

The first thing to check in a market step execution is the market type. If the SM matches, the corresponding function is called. Inside the shareholder function two additional matrices are created from the market actions, a buying matrix and a selling matrix.

buying and
selling matrix

Both matrices are initially filled with zeros, are quadratic, and their dimensions depend on the amount of agents, similar to the trading matrix. The buying matrix contains 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 the rows contain at maximum one entry. The selling matrix may contain ones only on the diagonal and only for the agent that wants to sell according to the market actions.

iteration

After setting up the two matrices they are iterated, extracting buyer, seller and the corresponding matrix rows. 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`

transaction

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.

example

An example for a transaction could be two agents acting in an environment. If agent 2 buys a share from agent 1, the trading matrix is updated. The second row stores the shares of agent two, which increase by 0.1 on the first position. This signalizes that agent 2 is owner of some shares of agent 1 and still has 100% of its own shares. In response, the shares in the first row and columns of agent 1 decreases to 0.9.

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

rewards

Additionally, the rewards would be updated if a price for the shares are set. In this implementation however, the shares have no price since agents are willing to sell anyway. Otherwise, the price would be subtracted from the buying agents' reward and that value would be added to the reward of the selling agent. Nonetheless, in this implementation the SM triggers a reward redistribution according to the trading matrix in each step.

The details of this calculation will be discussed in 4.4.3. Lastly, the transaction count is documented for evaluation purposes. At this point the market execution is done and the number of executed trades and the updated rewards are returned.

4.4.2. Action Market

action space

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 will be 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]`.

transaction

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 in this market setting the following conditions must be met:

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

success

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 2 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 2 this means that the negative value was spent. The first row shows that agent 1 still has a neutral balance and gained the `--trading-fee` of 0.1 from agent 2. To conclude, the market returns the number of transactions that took place in this step and the new rewards.

4.4.3. Reward Calculations

intro

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 is shown, in which a transaction is executed between two agents. Equal to the previous examples the red agent 2 buys a share or action from the blue agent 1. 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.

sm matrix

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 1 since agents start with complete ownership over their shares. Since in the example a transaction between agent 2 and agent 1 was established, the trading matrix is configured accordingly. The first matrix row implies

4. Approach

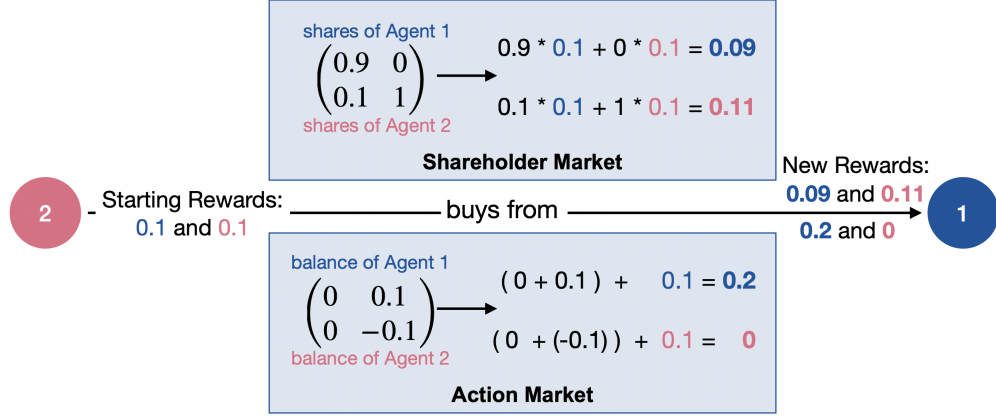


Figure 4.5.: Exemplary reward calculations of both market types

that Agent 1 is owner of 90% of its own shares and is not owner of any shares from agent 2. Whereas the second row shows that agent 2 claims 10% of the shares of agent 1 and has full ownership over its own.

sm
calcula-
tions

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 1 gets a new reward value of 0.09 and Agent 2 claims 0.01 from agent 1 and adds that to its full value of 0.1, resulting in a new reward of 0.11.

sm
calcula-
tions

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.

sm
calcula-
tions

Another difference to AMs is that the SM calculations are executed in each step, and it is irrelevant whether market action 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.

am
calcula-
tions

AMs, in most cases, update the rewards directly and only once during a transaction execution. The trading fee is directly subtracted from one agents' reward and added to the other during the trade. If an agent can not afford the fee the process is completed anyway and the agent goes into debt.

am
trading
matrix

Normally this market type makes no use of the trading matrix. In the specific scenario, see chapter 4.4.4, where the trading matrix is needed however, the agents market balance is summed up and added to the current reward. The balance is stored in the trading matrix rows. In this example the fee of 0.1 is subtracted from agent 2 and added to the reward of agent 1, leading to new rewards of 0.2 for agent 1 and zero for agent 2.

4.4.4. Additional Conditions

The AM or SM `--market` string 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 AMs with 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. 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.

debt

goal

conditions

trading matrix

returns

5. Results

- Result presentation
- Description of images and charts
 - One Agent Environment vs MARL
 - Best cases of reward, trades, grid coloration, field resets
 - Worst cases of above
 - influence of markets

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
 - challenges of markets (i.e. agents didn't need to sell shares/buy actions)
 - final reward is not necessary easy to interpret (did agent do good actions or did he just pick good market actions that sold)?
 - maybe markets need to be more specific (let agents know what others want to buy before choosing action!) and less based on chance - and/or more dynamic, i.e. instead of fixed prices agents can decide what to pay for actions/shares, so that they can for them self decide how important the trade is, and in case of shareholder market, maybe enable multi share purchase?

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 random seed (default: 1)

--agents AGENTS amount of agents

--model MODEL Name of the (trained) model, if none is given then a name is generated. (default: None)

--capture CAPTURE Boolean to enable capturing of environment and save as gif (default: True)

--env ENV name of the environment to train on (default: empty grid)

--agent-view-size AGENT_VIEW_SIZE grid size the agent can see, while standing in the middle (default: 5, so agent sees the 5x5 grid around him)

--grid-size GRID_SIZE size of the playing area (default: 9)

--max-steps MAX_STEPS max steps in environment to reach a goal

--setting SETTING If set to mixed-motive the reward is not shared which enables a competitive environment (one vs. all). Another setting is percentage-reward, where the reward is shared (coop) and is based on the percentage of the grid coloration. The last option is mixed-motive-competitive which extends the normal mixed-motive setting by removing the field reset option. When agents run over already colored fields the field immediately change the color to the one of the agent instead of resetting the color. (default: empty string - coop reward of one if the whole grid is colored)

--market MARKET There are three options 'sm', 'am' and '' for none. SM = Shareholder Market where agents can auction actions similar to stocks. AM = Action Market where agents can buy specific actions from others. (Default = '')

--trading-fee TRADING_FEE If a trade is executed, this value determines the price (market type am) / share (market type sm) the agents exchange (Default: 0.05)

--frames FRAMES number of frames of training (default: 1.000.000)

--frames-per-proc FRAMES_PER_PROC

Appendix A. Training Parameters

```

                                number of frames per process before update (default: 1024)
--procs PROCS                  Number of processes/environments running parallel (default: 16)
--recurrence RECURRENCE
                                number of time-steps gradient is backpropagated (default: 1). If >
                                1, a LSTM is added to the model to have memory.
--batch-size BATCH_SIZE
                                batch size for dqn (default: ppo 256, dqn 128)
--gamma GAMMA                  discount factor (default: 0.99)
--log-interval LOG_INTERVAL
                                number of frames between two logs (default: 1)
--save-interval SAVE_INTERVAL
                                number of updates between two saves (default: 10, 0 means no saving)
--capture-interval CAPTURE_INTERVAL
                                number of gif captures of episodes (default: 10, 0 means no
                                capturing)
--capture-frames CAPTURE_FRAMES
                                number of frames in capture (default: 50, 0 means no capturing)
--lr LR                        learning rate (default: 0.001)
--optim-eps OPTIM_EPS
                                Adam and RMSprop optimizer epsilon (default: 1e-8)
--epochs EPOCHS               number of epochs for PPO (default: 4)
--gae-lambda GAE_LAMBDA
                                lambda coefficient in GAE formula (default: 0.95, 1 means no gae)
--entropy-coef ENTROPY_COEF
                                entropy term coefficient (default: 0.01)
--value-loss-coef VALUE_LOSS_COEF
                                value loss term coefficient (default: 0.5)
--max-grad-norm MAX_GRAD_NORM
                                maximum norm of gradient (default: 0.5)
--clip-eps CLIP_EPS           clipping epsilon for PPO (default: 0.2)
--epsilon-start EPSILON_START
                                starting value of epsilon, used for action selection (default: 0.9
                                -> high exploration)
--epsilon-end EPSILON_END
                                ending value of epsilon, used for action selection (default: 0.05
                                -> high exploitation)
--epsilon-decay EPSILON_DECAY
                                Controls the rate of the epsilon decay in order to shift from
                                exploration to exploitation. The higher the value the slower
                                epsilon decays. (default: 1000)
--replay-size REPLAY_SIZE
                                Size of the replay memory (default: 100000)
--initial-target-update INITIAL_TARGET_UPDATE
                                Frames until the target network is updated, Needs to be smaller
                                than target update! (default: 10000)
--target-update TARGET_UPDATE
                                Frames between updating the target network, Needs to be smaller or
                                equal to frames-per-proc and bigger than initial target update!
                                (default: 100000 - 10 times the initial memory!)
```


List of Figures

2.1. Reinforcement Learning Cycle	3
2.2. Exemplary Use Of PPO	6
2.3. DQN with Experience Replay	7
3.1. Illustrated Markets	11
4.1. Coloring Environment	13
4.2. Agent Observation	15
4.3. The Training Structure	19
4.4. Market Elements	21
4.5. Exemplary Reward Calculation Of Markets	24

List of Tables

Code Listings

4.1. Exemplary command to execute training with three agents in a coloring environment using PPO as algorithm	16
---	----

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.
- [JJR19] Reza Jafari, Mohammad Masoud Javidi, and Marjan Kuchaki Rafsanjani. Using deep reinforcement learning approach for solving the multiple sequence alignment problem. *SN Applied Sciences*, 1(6):1–12, 2019.
- [KT03] Vijay R Konda and John N Tsitsiklis. On actor-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.
- [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.
- [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.

Bibliography

- [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.